Zephyr Project Documentation

Release 4.5.0-rc1

The Zephyr Project Contributors

Oct 12, 2026

Table of contents

1 Introduction **1**
1.1 Licensing . . . 1
1.2 Distinguishing Features . . . 2
1.3 Community Support . . . 4
1.4 Resources . . . 4
1.4.1 Getting Started . . . 4
1.4.2 Code and Development . . . 4
1.4.3 Community and Support . . . 4
1.4.4 Issue Tracking and Security . . . 4
1.4.5 Additional Resources . . . 4
1.5 Fundamental Terms and Concepts . . . 4

2 Developing with Zephyr **5**
2.1 Getting Started Guide . . . 5
2.1.1 Select and Update OS . . . 5
2.1.2 Install dependencies . . . 5
2.1.3 Get Zephyr and install Python dependencies . . . 7
2.1.4 Install the Zephyr SDK . . . 10
2.1.5 Build the Blinky Sample . . . 10
2.1.6 Flash the Sample . . . 11
2.1.7 Next Steps . . . 12
2.1.8 Asking for Help . . . 12
2.2 Beyond the Getting Started Guide . . . 13
2.2.1 Python and pip . . . 13
2.2.2 Advanced Platform Setup . . . 13
2.2.3 Install a Toolchain . . . 17
2.2.4 Updating the Zephyr SDK toolchain . . . 17
2.2.5 Cloning the Zephyr Repositories . . . 17
2.2.6 Export Zephyr CMake package . . . 18
2.2.7 Board Aliases . . . 18
2.2.8 Build and Run an Application . . . 19
2.3 Environment Variables . . . 21
2.3.1 Setting Variables . . . 21
2.3.2 Zephyr Environment Scripts . . . 23
2.3.3 Important Environment Variables . . . 23
2.4 Application Development . . . 24
2.4.1 Overview . . . 24
2.4.2 Application types . . . 25
2.4.3 Creating an Application . . . 26
2.4.4 Important Build System Variables . . . 29
2.4.5 Application CMakeLists.txt . . . 30
2.4.6 CMakeCache.txt . . . 32
2.4.7 Application Configuration . . . 32
2.4.8 Application-Specific Code . . . 33
2.4.9 Building an Application . . . 34
2.4.10 Run an Application . . . 37

2.4.11 Custom Board, Devicetree and SOC Definitions . . . 38
2.5 Debugging . . . 42
2.5.1 Application Debugging . . . 42
2.5.2 Debug with Eclipse . . . 44
2.5.3 Debugging I2C communication . . . 46
2.6 API Status and Guidelines . . . 47
2.6.1 API Overview . . . 47
2.6.2 API Lifecycle . . . 92
2.6.3 Vendor-Specific API Placement Policy . . . 96
2.6.4 API Design Guidelines . . . 97
2.6.5 API Terminology . . . 99
2.7 Language Support . . . 101
2.7.1 C Language Support . . . 101
2.7.2 C++ Language Support . . . 107
2.7.3 Rust Language Support . . . 110
2.8 Optimizations . . . 112
2.8.1 Optimizing for Footprint . . . 112
2.8.2 Optimization Tools . . . 113
2.9 Flashing and Hardware Debugging . . . 122
2.9.1 Flash & Debug Host Tools . . . 122
2.9.2 Debug Probes . . . 133
2.10 Modules (External projects) . . . 140
2.10.1 Modules vs west projects . . . 141
2.10.2 Module Repositories . . . 141
2.10.3 Contributing to Zephyr modules . . . 142
2.10.4 Licensing requirements and policies . . . 144
2.10.5 Documentation requirements . . . 145
2.10.6 Testing requirements . . . 145
2.10.7 Deprecating and removing modules . . . 145
2.10.8 Integrate modules in Zephyr build system . . . 146
2.10.9 Module yaml file description . . . 146
2.10.10 Submitting changes to modules . . . 155
2.11 West (Zephyr's meta-tool) . . . 157
2.11.1 Installing west . . . 157
2.11.2 West Release Notes . . . 158
2.11.3 Troubleshooting West . . . 171
2.11.4 Basics . . . 174
2.11.5 Built-in commands . . . 176
2.11.6 Workspaces . . . 180
2.11.7 West Manifests . . . 185
2.11.8 Configuration . . . 216
2.11.9 West aliases . . . 220
2.11.10 Extensions . . . 220
2.11.11 Building, Flashing and Debugging . . . 224
2.11.12 Signing Binaries . . . 242
2.11.13 Additional Zephyr extension commands . . . 243
2.11.14 History and Motivation . . . 252
2.11.15 Using Zephyr without west . . . 254
2.12 Test Runner (Twister) . . . 255
2.12.1 Command Line Options . . . 256
2.12.2 Integration with pytest test framework . . . 266
2.12.3 Twister Status . . . 274
2.12.4 Twister blackbox tests . . . 276
2.12.5 Board Configuration . . . 280
2.12.6 Toolchain Selection . . . 282
2.12.7 Tests . . . 283
2.12.8 Harnesses . . . 296
2.12.9 Sidecars . . . 307

2.12.10 Selecting platform scope . . . 308
2.12.11 Running in Integration Mode . . . 308
2.12.12 Running tests on custom emulator . . . 308
2.12.13 Running Tests in Random Order . . . 309
2.12.14 Running Tests on Hardware . . . 309
2.12.15 Test Output and Reports . . . 317
2.12.16 Live Run Monitoring . . . 318
2.12.17 Twister Configuration File . . . 318
2.13 Testing . . . 320
2.13.1 Test Framework . . . 321
2.13.2 Benchmarking Framework . . . 335
2.13.3 Generating coverage reports . . . 339
2.13.4 BabbleSim . . . 342
2.13.5 CI Test Plan Selector (test_plan_v2) . . . 345
2.14 Static Code Analysis (SCA) . . . 350
2.14.1 SCA Tool infrastructure . . . 351
2.14.2 Native SCA Tool support . . . 351
2.15 Toolchains . . . 361
2.15.1 Zephyr SDK . . . 361
2.15.2 Arm Compiler 6 . . . 365
2.15.3 Arm Toolchain for Embedded (ATfE) . . . 365
2.15.4 Cadence Tensilica Xtensa C/C++ Compiler (XCC) . . . 367
2.15.5 DesignWare ARC MetaWare Development Toolkit (MWDT) . . . 368
2.15.6 GNU Arm Embedded . . . 369
2.15.7 Qualcomm Hexagon LLVM Toolchain . . . 369
2.15.8 IAR Arm Toolchain . . . 370
2.15.9 Intel oneAPI Toolkit . . . 371
2.15.10 MPLAB XC32 . . . 371
2.15.11 Host Toolchains . . . 372
2.15.12 Other Cross Compilers . . . 372
2.15.13 Custom CMake Toolchains . . . 373
2.15.14 Embedded Linker (ELD) . . . 373
2.16 Tools and IDEs . . . 374
2.16.1 CLion . . . 374
2.16.2 Coccinelle . . . 379
2.16.3 IDE for Zephyr (VS Code extension) . . . 387
2.16.4 Kapa.ai Documentation Assistant . . . 390
2.16.5 STM32CubeIDE . . . 392
2.16.6 Visual Studio Code . . . 397
2.16.7 Workbench for Zephyr . . . 399

3 Kernel **405**
3.1 Kernel Services . . . 405
3.1.1 Scheduling, Interrupts, and Synchronization . . . 405
3.1.2 Data Passing . . . 471
3.1.3 Memory Management . . . 493
3.1.4 Timing . . . 493
3.1.5 Other . . . 505
3.2 Device Driver Model . . . 518
3.2.1 Introduction . . . 518
3.2.2 Standard Drivers . . . 519
3.2.3 Synchronous Calls . . . 519
3.2.4 Driver APIs . . . 519
3.2.5 Driver Data Structures . . . 519
3.2.6 Subsystems and API Structures . . . 520
3.2.7 API Class Inheritance . . . 521
3.2.8 Device-Specific API Extensions . . . 522
3.2.9 Single Driver, Multiple Instances . . . 523

3.2.10 Initialization Levels . . . 524
3.2.11 Deferred initialization . . . 524
3.2.12 System Drivers . . . 525
3.2.13 Inspecting the initialization sequence . . . 525
3.2.14 Error handling . . . 525
3.2.15 Memory Mapping . . . 525
3.2.16 API Reference . . . 528
3.3 User Mode . . . 528
3.3.1 Overview . . . 528
3.3.2 Memory Protection Design . . . 531
3.3.3 Kernel Objects . . . 537
3.3.4 System Calls . . . 541
3.3.5 MPU Stack Objects . . . 550
3.3.6 MPU Backed Userspace . . . 551
3.4 Memory Management . . . 551
3.4.1 Memory Heaps . . . 551
3.4.2 Shared Multi Heap . . . 555
3.4.3 Memory Slabs . . . 556
3.4.4 Memory Blocks Allocator . . . 558
3.4.5 Demand Paging . . . 561
3.4.6 Virtual Memory . . . 565
3.5 Data Structures . . . 567
3.5.1 Single-linked List . . . 567
3.5.2 Double-linked List . . . 569
3.5.3 Multi Producer Single Consumer Packet Buffer . . . 571
3.5.4 Single Producer Single Consumer Packet Buffer . . . 573
3.5.5 Balanced Red/Black Tree . . . 573
3.5.6 Ring Buffers . . . 576
3.5.7 Multi Producer Single Consumer Lock Free Queue . . . 579
3.5.8 Single Producer Single Consumer Lock Free Queue . . . 579
3.5.9 Min-Heap Data Structure . . . 580
3.5.10 sys_ringq Data Structure . . . 581
3.6 Executing Time Functions . . . 583
3.6.1 Configuration . . . 583
3.6.2 Usage . . . 583
3.6.3 API documentation . . . 584
3.7 Object Cores . . . 584
3.7.1 Object Core Concepts . . . 584
3.7.2 Object Core Statistics Concepts . . . 585
3.7.3 Implementation . . . 586
3.7.4 Configuration Options . . . 588
3.7.5 API Reference . . . 589
3.8 Time Utilities . . . 589
3.8.1 Overview . . . 589
3.8.2 Time Utility APIs . . . 589
3.8.3 Concepts Underlying Time Support in Zephyr . . . 592
3.9 Precision Timing . . . 594
3.9.1 Precision time . . . 594
3.9.2 Precision clock . . . 594
3.9.3 PI controller . . . 594
3.9.4 Protocol integration . . . 595
3.9.5 Sample . . . 595
3.9.6 API reference . . . 595
3.10 Utilities . . . 595
3.11 Iterable Sections . . . 596
3.11.1 Overview . . . 596
3.11.2 Step 1: Define the data in C . . . 596
3.11.3 Step 2: Declare the section in CMake . . . 597

3.11.4 Step 3: Provide a linker-script . . . 597
3.11.5 Iterating over the entries . . . 597
3.11.6 API Reference . . . 598
3.12 Code And Data Relocation . . . 598
3.12.1 Overview . . . 598
3.12.2 Details . . . 598
3.13 Scope-based Cleanup Helpers . . . 600
3.13.1 Overview . . . 601
3.13.2 Core Concepts . . . 601
3.13.3 Defining Scoped Types . . . 602
3.13.4 Usage Notes . . . 604
3.13.5 API Reference . . . 605

4 Services **607**
4.1 Connectivity . . . 607
4.1.1 Networking . . . 607
4.1.2 Bluetooth . . . 821
4.1.3 USB . . . 1115
4.1.4 Controller Area Network (CAN) Bus Protocols . . . 1147
4.1.5 LoRa and LoRaWAN . . . 1147
4.1.6 Modbus . . . 1149
4.1.7 Modem modules . . . 1150
4.2 Input / Output . . . 1158
4.2.1 Console . . . 1158
4.2.2 Input . . . 1158
4.2.3 Real Time I/O (RTIO) . . . 1165
4.2.4 Formatted Output . . . 1169
4.3 Inter-Process Communication . . . 1171
4.3.1 Interprocessor Communication (IPC) . . . 1171
4.3.2 Zephyr bus (zbus) . . . 1182
4.3.3 Asynchronous Notifications . . . 1204
4.3.4 Virtualization . . . 1205
4.4 Logging, Tracing & Debugging . . . 1206
4.4.1 Logging . . . 1206
4.4.2 Tracing . . . 1227
4.4.3 Debugging . . . 1239
4.4.4 Profiling . . . 1255
4.4.5 Instrumentation . . . 1255
4.4.6 CPU load . . . 1260
4.4.7 Shell . . . 1260
4.4.8 Task Watchdog . . . 1279
4.5 Storage & Configuration . . . 1280
4.5.1 Disk Access . . . 1280
4.5.2 Flash Circular Buffer (FCB) . . . 1285
4.5.3 File Systems . . . 1286
4.5.4 Flash map . . . 1287
4.5.5 Non-Volatile Memory (NVMEM) . . . 1289
4.5.6 Non-Volatile Storage (NVS) . . . 1291
4.5.7 Retention System . . . 1293
4.5.8 Secure Storage . . . 1298
4.5.9 Settings . . . 1301
4.5.10 Stream Flash . . . 1307
4.5.11 Zephyr Memory Storage (ZMS) . . . 1307
4.6 Memory Management . . . 1317
4.6.1 Memory Attributes . . . 1317
4.6.2 Network Buffers . . . 1320
4.7 Power Management . . . 1323
4.7.1 Power Management . . . 1323

4.7.2 CPU Frequency Scaling . . . 1348
4.7.3 Power off . . . 1352
4.7.4 Resource Management . . . 1352
4.8 Security . . . 1354
4.8.1 Cryptography . . . 1354
4.8.2 Authentication . . . 1359
4.8.3 JSON Web Token (JWT) . . . 1361
4.8.4 Trusted Firmware-M (TF-M) . . . 1362
4.8.5 UUID . . . 1372
4.9 Device Management . . . 1373
4.9.1 MCUmgr . . . 1374
4.9.2 MCUmgr handlers . . . 1377
4.9.3 MCUmgr Callbacks . . . 1386
4.9.4 SMP Protocol Specification . . . 1391
4.9.5 SMP Transport Specification . . . 1443
4.9.6 Device Firmware Upgrade . . . 1446
4.9.7 Over-the-Air Update . . . 1447
4.9.8 Firmware Over-the-Air over HTTP . . . 1448
4.9.9 EC Host Command . . . 1450
4.9.10 Binary Descriptors . . . 1454
4.10 Algorithms & Data . . . 1456
4.10.1 CRC . . . 1456
4.10.2 Digital Signal Processing (DSP) . . . 1458
4.10.3 Serialization . . . 1458
4.11 Frameworks . . . 1461
4.11.1 Linkable Loadable Extensions (LLEXT) . . . 1461
4.11.2 Multimedia Pipeline (mpipe) . . . 1478
4.11.3 Sensing Subsystem . . . 1484
4.11.4 State Machine Framework . . . 1490
4.12 OS Abstraction . . . 1502
4.12.1 POSIX . . . 1502
4.12.2 CMSIS RTOS v1 . . . 1530
4.12.3 CMSIS RTOS v2 . . . 1530

5 Build and Configuration Systems 1533
5.1 Build System (CMake) . . . 1533
5.1.1 Build and Configuration Phases . . . 1533
5.1.2 Supporting Scripts and Tools . . . 1539
5.2 CMake Reference . . . 1543
5.2.1 Core Build System & Hardware Modules . . . 1543
5.2.2 Sysbuild Modules . . . 1581
5.2.3 Utility & Tooling Modules . . . 1586
5.2.4 Target properties . . . 1591
5.2.5 Variables . . . 1591
5.3 Devicetree . . . 1596
5.3.1 Devicetree Guide . . . 1596
5.3.2 Devicetree Reference . . . 1666
5.4 Configuration System (Kconfig) . . . 1906
5.4.1 Interactive Kconfig interfaces . . . 1906
5.4.2 Tracing values to their source . . . 1911
5.4.3 Setting Kconfig configuration values . . . 1912
5.4.4 Kconfig - Tips and Best Practices . . . 1917
5.4.5 Custom Kconfig Preprocessor Functions . . . 1932
5.4.6 Kconfig extensions . . . 1935
5.5 Snippets . . . 1937
5.5.1 Using Snippets . . . 1937
5.5.2 Built-in snippets . . . 1938
5.5.3 Writing Snippets . . . 1950

5.5.4 Snippets Design . . . 1954
5.6 Zephyr CMake Package . . . 1955
5.6.1 Zephyr CMake package export (west) . . . 1955
5.6.2 Zephyr CMake package export (without west) . . . 1956
5.6.3 Zephyr Base Environment Setting . . . 1956
5.6.4 Zephyr CMake Package Search Order . . . 1956
5.6.5 Zephyr CMake Package Version . . . 1957
5.6.6 Multiple Zephyr Installations (Zephyr workspace) . . . 1958
5.6.7 Zephyr Build Configuration CMake packages . . . 1959
5.6.8 Zephyr CMake package source code . . . 1961
5.7 Sysbuild (System build) . . . 1961
5.7.1 Definitions . . . 1961
5.7.2 Architectural Overview . . . 1962
5.7.3 Building with sysbuild . . . 1963
5.7.4 Configuration namespacing . . . 1963
5.7.5 Sysbuild flashing using `west flash` . . . 1965
5.7.6 Sysbuild debugging using `west debug` . . . 1965
5.7.7 Building a sample with MCUboot . . . 1965
5.7.8 Sysbuild Kconfig file . . . 1966
5.7.9 Sysbuild targets . . . 1966
5.7.10 Dedicated image build targets . . . 1967
5.7.11 Adding Zephyr applications to sysbuild . . . 1967
5.7.12 Merged hex files . . . 1971
5.7.13 Adding non-Zephyr applications to sysbuild . . . 1972
5.7.14 Configuring sysbuild internal state . . . 1972
5.7.15 Extending sysbuild . . . 1972
5.7.16 Sysbuild and CMake presets . . . 1979
5.8 Application version management . . . 1979
5.8.1 VERSION file . . . 1980
5.8.2 Application build version . . . 1980
5.8.3 Use in code . . . 1981
5.8.4 Use in Kconfig . . . 1981
5.8.5 Use in CMake . . . 1982
5.8.6 Use in MCUboot-supported applications . . . 1983
5.8.7 Use in Shell . . . 1983
5.9 Flashing . . . 1983
5.9.1 Flashing configuration . . . 1984
5.10 Signing Binaries . . . 1985
5.10.1 MCUboot / imgtool . . . 1985
5.10.2 Signing key files . . . 1986
5.10.3 Extending signing externally . . . 1987

6 Hardware Support **1989**
6.1 Architecture-related Guides . . . 1989
6.1.1 Zephyr support status on ARC processors . . . 1989
6.1.2 Arm Cortex-M Developer Guide . . . 1991
6.1.3 ARM64 Developer Guide . . . 2003
6.1.4 Zephyr support status on RISC-V processors . . . 2004
6.1.5 Semihosting Guide . . . 2006
6.1.6 x86 Developer Guide . . . 2007
6.1.7 Xtensa Developer Guide . . . 2008
6.2 Barriers API . . . 2010
6.3 Caching . . . 2010
6.3.1 Cache Control Configuration . . . 2010
6.3.2 Caching Basics . . . 2011
6.4 Zephyr's device emulators/simulators . . . 2013
6.4.1 Overview . . . 2013
6.4.2 Available Emulators . . . 2014

6.5 External Bus and Bus Connected Peripherals Emulators . . . 2015
6.5.1 Overview . . . 2015
6.5.2 Concept . . . 2015
6.5.3 Creating a Device Driver Emulator . . . 2016
6.5.4 Available Emulators . . . 2018
6.5.5 Samples . . . 2018
6.5.6 API Reference . . . 2019
6.6 Firmware . . . 2019
6.6.1 Overview . . . 2019
6.6.2 API Reference . . . 2019
6.6.3 Samples . . . 2019
6.6.4 Resources . . . 2019
6.7 Peripherals . . . 2020
6.7.1 Analog . . . 2020
6.7.2 Audio . . . 2023
6.7.3 Communication . . . 2026
6.7.4 Display . . . 2054
6.7.5 GPIO & Input . . . 2060
6.7.6 Memory & Storage . . . 2063
6.7.7 Power . . . 2068
6.7.8 Sensors . . . 2074
6.7.9 Motion & Actuation . . . 2091
6.7.10 System & Diagnostics . . . 2099
6.7.11 Timing . . . 2105
6.8 Pin Control . . . 2110
6.8.1 Introduction . . . 2110
6.8.2 State model . . . 2111
6.8.3 Dynamic pin control . . . 2113
6.8.4 Devicetree representation . . . 2113
6.8.5 Implementation guidelines . . . 2116
6.8.6 Pin Control API . . . 2117
6.8.7 Other reference material . . . 2118
6.9 Porting . . . 2118
6.9.1 Architecture Porting Guide . . . 2118
6.9.2 SoC Porting Guide . . . 2133
6.9.3 Board Porting Guide . . . 2137
6.9.4 Shields . . . 2151
6.10 Virtualization . . . 2163
6.10.1 Virtual I/O (VIRTIO) . . . 2163

7 Contributing to Zephyr 2167
7.1 General Guidelines . . . 2167
7.1.1 Contribution Guidelines . . . 2167
7.1.2 Contributor Expectations . . . 2195
7.1.3 Reviewer Expectations . . . 2199
7.1.4 Coding Guidelines . . . 2200
7.1.5 Coding Style Guidelines . . . 2214
7.1.6 Proposals and RFCs . . . 2231
7.1.7 Modifying Contributions made by other developers . . . 2232
7.1.8 Pull Request Lifecycle Policy . . . 2233
7.2 Documentation . . . 2235
7.2.1 Documentation Guidelines . . . 2235
7.2.2 Documentation Generation . . . 2257
7.3 Dealing with external components . . . 2262
7.3.1 Contributing External Components . . . 2262
7.3.2 Binary Blobs . . . 2266
7.4 Need help along the way? . . . 2270

8 Project and Governance **2271**
8.1 Technical Steering Committee (TSC) . . . 2271
8.1.1 TSC Member Role . . . 2271
8.2 TSC Project Roles . . . 2274
8.2.1 Project Roles . . . 2274
8.2.2 Teams and Supporting Activities . . . 2276
8.2.3 Role Retirement . . . 2278
8.2.4 MAINTAINERS File . . . 2278
8.3 TSC Working Groups . . . 2279
8.3.1 Overview . . . 2279
8.3.2 Membership . . . 2279
8.3.3 Advisory role . . . 2280
8.3.4 TSC Working Group Lifecycle . . . 2281
8.4 Release Process . . . 2281
8.4.1 Release Phases Overview . . . 2281
8.4.2 Issue Tracking during Feature Freeze . . . 2283
8.4.3 Pull Request Merge Criteria . . . 2284
8.4.4 Release Quality Criteria . . . 2285
8.4.5 Release Milestones . . . 2286
8.4.6 Long Term Support (LTS) Releases . . . 2287
8.4.7 Host Tools Support Policy . . . 2289
8.4.8 Hardware Support Tiers . . . 2289
8.4.9 Release Procedure . . . 2290
8.5 Feature Tracking . . . 2292
8.5.1 Roadmap and Release Plans . . . 2292
8.6 Code Flow and Branches . . . 2293
8.6.1 Introduction . . . 2293
8.6.2 Roles and Responsibilities . . . 2294
8.7 Development Environment and Tools . . . 2294
8.7.1 Code Review . . . 2294
8.7.2 Continuous Integration . . . 2298
8.7.3 Labeling issues and pull requests in GitHub . . . 2298
8.8 Bug Reporting . . . 2300
8.8.1 Reporting a regression issue . . . 2300
8.9 Communication and Collaboration . . . 2301
8.10 Code Documentation . . . 2301
8.10.1 API Documentation . . . 2301
8.10.2 Reference to Requirements . . . 2301
8.10.3 Test Documentation . . . 2301
8.10.4 Documentation Guidelines . . . 2301
8.11 Terminology . . . 2303

9 Security **2305**
9.1 Zephyr Security Overview . . . 2305
9.1.1 Introduction . . . 2305
9.1.2 Current Security Definition . . . 2306
9.1.3 Secure Development Process . . . 2308
9.1.4 Secure Design . . . 2312
9.1.5 Security Certification . . . 2314
9.2 Security Vulnerability Reporting . . . 2315
9.2.1 Introduction . . . 2315
9.2.2 Vulnerability Reporting Guidelines . . . 2315
9.2.3 Security Issue Management . . . 2316
9.2.4 Vulnerability Notification . . . 2318
9.2.5 Backporting of Security Vulnerabilities . . . 2319
9.2.6 Need to Know . . . 2319
9.3 Secure Coding . . . 2319
9.3.1 Introduction and Scope . . . 2320

9.3.2 Secure Coding . . . 2320
9.3.3 Secure development knowledge . . . 2321
9.3.4 Code Review . . . 2322
9.3.5 Issues and Bug Tracking . . . 2322
9.3.6 Modifications to This Document . . . 2322
9.4 Sensor Device Threat Model . . . 2323
9.4.1 Assets . . . 2323
9.4.2 Communication . . . 2325
9.4.3 Other Considerations . . . 2327
9.4.4 Threats . . . 2327
9.4.5 Notes . . . 2327
9.5 Hardening Tool . . . 2327
9.5.1 Usage . . . 2328
9.5.2 Profiles . . . 2328
9.5.3 Configuration options . . . 2329
9.5.4 The hardening database . . . 2329
9.5.5 Extending the database out of tree . . . 2330
9.6 Control Flow Integrity . . . 2331
9.6.1 Forward and return edges . . . 2331
9.6.2 Hardware support . . . 2332
9.7 Vulnerabilities . . . 2333
9.7.1 CVE-2017 . . . 2333
9.7.2 CVE-2019 . . . 2334
9.7.3 CVE-2020 . . . 2334
9.7.4 CVE-2021 . . . 2341
9.7.5 CVE-2022 . . . 2345
9.7.6 CVE-2023 . . . 2346
9.7.7 CVE-2024 . . . 2351
9.7.8 CVE-2025 . . . 2354
9.7.9 CVE-2026 . . . 2357
9.8 Security standards and Zephyr . . . 2440
9.8.1 EU Cyber Resilience Act (CRA) . . . 2440
9.8.2 ETSI 303-645 . . . 2447

10 Safety **2457**
10.1 Zephyr Safety Overview . . . 2457
10.1.1 Introduction . . . 2457
10.1.2 Overview . . . 2457
10.1.3 Safety Document update . . . 2457
10.1.4 General safety scope . . . 2457
10.1.5 Quality . . . 2458
10.1.6 Processes and workflow . . . 2461
10.2 Safety Requirements Management . . . 2463
10.2.1 Safety Requirements . . . 2463
10.2.2 Safety Requirements Checklist . . . 2467
10.2.3 Getting Started with Requirements . . . 2470
10.2.4 Requirements Catalog . . . 2473

11 Glossary of Terms **2547**

Bibliography **2551**

Python Module Index **2553**

Index **2555**

Chapter 1

Introduction

The Zephyr OS is based on a small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications.

The Zephyr kernel supports multiple architectures, including:

- ARCv2 (EM and HS) and ARCv3 (HS6X)
- ARMv6-M, ARMv7-M, and ARMv8-M (Cortex-M)
- ARMv7-A and ARMv8-A (Cortex-A, 32- and 64-bit)
- ARMv7-R, ARMv8-R (Cortex-R, 32- and 64-bit)
- Qualcomm Hexagon
- Intel x86 (32- and 64-bit)
- MIPS (MIPS32 Release 1 specification)
- OpenRISC (32-bit)
- Renesas RX
- RISC-V (32- and 64-bit)
- SPARC V8
- Tensilica Xtensa
- TriCore (TC1.6.2 and TC1.8)

The full list of supported boards based on these architectures can be found here.

In the context of the Zephyr OS, a *subsystem* refers to a logically distinct part of the operating system that handles specific functionality or provides certain services. Subsystems can include components such as networking, file systems, device driver classes, power management, and communication protocols, among others. Each subsystem is designed to be modular and can be configured, customized, and extended to meet the requirements of different embedded applications.

1.1 Licensing

Zephyr is permissively licensed using the Apache 2.0 license (as found in the `LICENSE` file in the project's GitHub repo). There are some imported or reused components of the Zephyr project that use other licensing, as described in *Licensing of Zephyr Project components*.

1.2 Distinguishing Features

Zephyr offers a large and ever growing number of features including:

Extensive suite of Kernel services
Zephyr offers a number of familiar services for development:

- *Multi-threading Services* for cooperative, priority-based, non-preemptive, and preemptive threads with optional round robin time-slicing. Includes POSIX pthreads compatible API support.
- *Interrupt Services* for compile-time registration of interrupt handlers.
- *Memory Allocation Services* for dynamic allocation and freeing of fixed-size or variable-size memory blocks.
- *Inter-thread Synchronization Services* for binary semaphores, counting semaphores, and mutex semaphores.
- *Inter-thread Data Passing Services* for basic message queues, enhanced message queues, and byte streams.
- *Power Management Services* such as overarching, application or policy-defined, System Power Management and fine-grained, driver-defined, Device Power Management.

Multiple Scheduling Algorithms
Zephyr provides a comprehensive set of thread scheduling choices:

- Cooperative and Preemptive Scheduling
- Earliest Deadline First (EDF)
- Meta IRQ scheduling implementing "interrupt bottom half" or "tasklet" behavior
- Timeslicing: Enables time slicing between preemptible threads of equal priority
- Multiple queuing strategies:
  - Simple linked-list ready queue
  - Red/black tree ready queue
  - Traditional multi-queue ready queue

Highly configurable / Modular for flexibility
Allows an application to incorporate *only* the capabilities it needs as it needs them, and to specify their quantity and size.

Cross Architecture
Supports a wide variety of supported boards with different CPU architectures and developer tools. Contributions have added support for an increasing number of SoCs, platforms, and drivers.

Memory Protection
Implements configurable architecture-specific stack-overflow protection, kernel object and device driver permission tracking, and thread isolation with thread-level memory protection on x86, ARC, and ARM architectures, userspace, and memory domains.

For platforms without MMU/MPU and memory constrained devices, supports combining application-specific code with a custom kernel to create a monolithic image that gets loaded and executed on a system's hardware. Both the application code and kernel code execute in a single shared address space.

Compile-time resource definition
Allows system resources to be defined at compile-time, which reduces code size and increases performance for resource-limited systems.

Optimized Device Driver Model
Provides a consistent device model for configuring the drivers that are part of the platform/system and a consistent model for initializing all the drivers configured into the system and allows the reuse of drivers across platforms that have common devices/IP blocks.

Devicetree Support
Use of *devicetree* to describe hardware. Information from devicetree is used to create the application image.

Native Networking Stack supporting multiple protocols
Networking support is fully featured and optimized, including LwM2M and BSD sockets compatible support. OpenThread support (on Nordic chipsets) is also provided - a mesh network designed to securely and reliably connect hundreds of products around the home.

Bluetooth Low Energy 5.0 support
Bluetooth 5.0 compliant (ESR10) and Bluetooth Low Energy Controller support (LE Link Layer). Includes Bluetooth Mesh and a Bluetooth qualification-ready Bluetooth controller.

- Generic Access Profile (GAP) with all possible LE roles
- Generic Attribute Profile (GATT)
- Pairing support, including the Secure Connections feature from Bluetooth 4.2
- Clean HCI driver abstraction
- Raw HCI interface to run Zephyr as a Controller instead of a full Host stack
- Verified with multiple popular controllers
- Highly configurable

Mesh Support:

- Relay, Friend Node, Low-Power Node (LPN) and GATT Proxy features
- Both Provisioning bearers supported (PB-ADV & PB-GATT)
- Highly configurable, fitting in devices with at least 16k RAM

Native Linux, macOS, and Windows Development
A command-line CMake build environment runs on popular developer OS systems. A native port (native_sim) lets you build and run Zephyr as a native application on Linux, aiding development and testing.

Virtual File System Interface with ext2, FatFs, and LittleFS Support
ext2, LittleFS and FatFS support; FCB (Flash Circular Buffer) for memory constrained applications.

Powerful multi-backend logging Framework
Support for log filtering, object dumping, panic mode, multiple backends (memory, networking, filesystem, console, ...) and integration with the shell subsystem.

User friendly and full-featured Shell interface
A multi-instance shell subsystem with user-friendly features such as autocompletion, wildcards, coloring, metakeys (arrows, backspace, ctrl+u, etc.) and history. Support for static commands and dynamic sub-commands.

Settings on non-volatile storage
The settings subsystem gives modules a way to store persistent per-device configuration and runtime state. Settings items are stored as key-value pair strings.

Non-volatile storage (NVS)
NVS allows storage of binary blobs, strings, integers, longs, and any combination of these.

Native port
Native sim allows running Zephyr as a Linux application with support for various subsystems and networking.

1.3 Community Support

Community support is provided via mailing lists and Discord; see the Resources below for details.

1.4 Resources

Here's a quick summary of resources to help you find your way around:

1.4.1 Getting Started

- Zephyr Documentation
- Getting Started Guide
- Tips when asking for help
- Code samples

1.4.2 Code and Development

- Source Code Repository
- Releases
- Contribution Guide

1.4.3 Community and Support

- Discord Server for real-time community discussions
- User mailing list (users@lists.zephyrproject.org)
- Developer mailing list (devel@lists.zephyrproject.org)
- Other project mailing lists
- Project Wiki

1.4.4 Issue Tracking and Security

- GitHub Issues
- Security documentation
- Security Advisories Repository
- Report security vulnerabilities at vulnerabilities@zephyrproject.org

1.4.5 Additional Resources

- Zephyr Project Website
- Zephyr Tech Talks

1.5 Fundamental Terms and Concepts

See *Glossary of Terms*

Chapter 2

Developing with Zephyr

2.1 Getting Started Guide

Follow this guide to:

- Set up a command-line Zephyr development environment on Ubuntu, macOS, or Windows (instructions for other Linux distributions are discussed in *Install Linux Host Dependencies*)
- Get the source code
- Build, flash, and run a sample application

2.1.1 Select and Update OS

Click the operating system you are using.

Ubuntu

This guide covers Ubuntu version 24.04 LTS and later. If you are using a different Linux distribution see *Install Linux Host Dependencies*.

```
sudo apt update
sudo apt upgrade
```

macOS

Select *System Settings* ‣ *General* ‣ *Software Update* and install any available updates. See this Apple support topic for more details.

> **Note**
>
> x86-64 macOS is not supported.

Windows

Select *Start* ‣ *Settings* ‣ *Update & Security* ‣ *Windows Update*. Click *Check for updates* and install any that are available.

2.1.2 Install dependencies

Next, install the host tools Zephyr needs to configure and build applications. The instructions below use the recommended package manager for each operating system so the tools are available from your terminal.

The current minimum required versions for the main dependencies are:

Tool	Min. Version
CMake	3.28.0
Python	3.12
Devicetree compiler	1.4.6

Note

Python 3.12 is strongly recommended. Using a newer Python release may fail on some systems, for example when installing the required packages on Windows.

Ubuntu

1. Use `apt` to install the required dependencies:

```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget python3-dev python3-venv␣
↪python3-tk \
  xz-utils file make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1
```

Note

Due to the unavailability of `gcc-multilib` and `g++-multilib` on AArch64 (ARM64) systems, you may need to omit them from the list of packages to install.

2. Verify the versions of the main dependencies installed on your system by entering:

```
cmake --version
python3 --version
dtc --version
```

Check those against the versions in the table in the beginning of this section. Refer to the *Install Linux Host Dependencies* page for additional information on updating the dependencies manually.

macOS

1. Install Homebrew:

```
/bin/bash -c "$(curl -fsSL␣
↪https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. After the Homebrew installation script completes, follow the on-screen instructions to add the Homebrew installation to the path.

```
(echo; echo 'eval "$(/opt/homebrew/bin/brew shellenv)"') >> ~/.zprofile
source ~/.zprofile
```

3. Use `brew` to install the required dependencies:

```
brew install cmake ninja gperf python3 python-tk ccache qemu dtc libmagic␣
↪wget openocd
```

4. Add the Homebrew Python folder to the path so you can execute `python` and `pip` as well as `python3` and `pip3`.

```
(echo; echo 'export PATH="'$(brew --prefix)'/
↪opt/python/libexec/bin:$PATH"') >> ~/.zprofile
source ~/.zprofile
```

Windows

> **Note**
>
> These instructions cover the native Windows environment. You may also use the Windows Subsystem for Linux (WSL) by following the Ubuntu instructions in this guide instead. In that case, note that flashing and debugging hardware from within WSL requires first making the USB device visible to WSL, for example using usbipd-win.

On modern versions of Windows (10 and later), install Windows Terminal from the Microsoft Store. The instructions below work in either `cmd.exe` or PowerShell.

These instructions use Windows' official package manager, winget. If winget isn't an option, install the dependencies from their respective websites and make sure their command line tools are on your *`PATH` environment variable*.

1. In modern Windows versions, winget is already pre-installed by default. You can verify that this is the case by typing `winget` in a terminal window. If that fails, you can then install winget.
2. Open a Command Prompt (`cmd.exe`) or PowerShell terminal window. To do so, press the Windows key, type `cmd.exe` or PowerShell and click on the result.
3. Use `winget` to install the required dependencies:

   ```
   winget install Kitware.CMake Ninja-build.Ninja oss-winget.gperf␣
   ↪Python.Python.3.12 Git.Git oss-winget.dtc wget 7zip.7zip
   ```

4. Close the terminal window.

> **Note**
>
> You may need to add the 7zip installation folder to your `PATH`.

2.1.3 Get Zephyr and install Python dependencies

Next, use *west* to create a workspace and fetch Zephyr together with its *modules*.

These commands use `zephyrproject` as the workspace name; you can choose another name and location. You will also install Zephyr's Python dependencies in a Python virtual environment so they stay separate from your system Python installation.

1. Create a new virtual environment:

   Ubuntu

   ```
   python3 -m venv ~/zephyrproject/.venv
   ```

   macOS

   ```
   python3 -m venv ~/zephyrproject/.venv
   ```

   Windows

Open a `cmd.exe` or PowerShell terminal window **as a regular user**.

Batchfile

```
cd %HOMEPATH%
py -3.12 -m venv zephyrproject\.venv
```

PowerShell

```
cd $Env:HOMEPATH
py -3.12 -m venv zephyrproject\.venv
```

2. Activate the virtual environment:

Ubuntu

```
source ~/zephyrproject/.venv/bin/activate
```

macOS

```
source ~/zephyrproject/.venv/bin/activate
```

Windows

> **Note**
>
> Python's virtual environment activation in PowerShell requires running a script itself, which needs to be allowed.
>
> ```
> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
> ```

Batchfile

```
zephyrproject\.venv\Scripts\activate.bat
```

PowerShell

```
zephyrproject\.venv\Scripts\Activate.ps1
```

Once activated your shell will be prefixed with `(.venv)`. The virtual environment can be deactivated at any time by running `deactivate`.

> **Note**
>
> Remember to activate the virtual environment every time you start a new terminal session before working with Zephyr. If you don't, commands such as `west` will not be found, or may run against a different Python environment, leading to confusing errors.

3. Install west:

West is Zephyr's workspace manager; the next commands use it to create and update the workspace.

```
pip install west
```

4. Get the Zephyr source code:

`west init` creates a *west workspace* and clones `https://github.com/zephyrproject-rtos/zephyr` as its *manifest repository*.

`west update` then fetches the various *west projects* (modules) listed in Zephyr's *west manifest* (hardware abstraction layers (HALs), libraries, etc.).

Ubuntu

```
west init -m https://github.com/zephyrproject-rtos/zephyr ~/zephyrproject
cd ~/zephyrproject
west update
```

macOS

```
west init -m https://github.com/zephyrproject-rtos/zephyr ~/zephyrproject
cd ~/zephyrproject
west update
```

Windows

```
west init -m https://github.com/zephyrproject-rtos/zephyr zephyrproject
cd zephyrproject
west update
```

> **Tip**
>
> To reduce disk space usage and avoid downloading unnecessary modules or vendor HALs during setup, you may configure *Project Groups* before running `west update`.

5. Install Zephyr's Python dependencies:

   `west packages` reads the Python requirements from the checked-out Zephyr workspace (including its modules), so the installed packages match the Zephyr version you fetched.

   Ubuntu

```
west packages pip --install
```

   macOS

```
west packages pip --install
```

   Windows

   Batchfile

```
cmd /c zephyr\scripts\utils\west-packages-pip-install.cmd
```

   PowerShell

```
python -m pip install @((west packages pip) -split ' ')
```

> **Note**
>
> Installing these dependencies can downgrade or upgrade west itself.

6. Export a *Zephyr CMake package*. This registers your current Zephyr checkout in CMake's user package registry so `find_package(Zephyr)` can locate it automatically when building applications.

```
west zephyr-export
```

2.1.4 Install the Zephyr SDK

The *Zephyr Software Development Kit (SDK)* contains toolchains for each of Zephyr's supported architectures. Those toolchains include the compiler, assembler, linker, and other programs required to build Zephyr applications for your target hardware.

It also contains additional host tools, such as custom QEMU and OpenOCD builds that are used to emulate, flash and debug Zephyr applications.

Install the Zephyr SDK with `west sdk install` from the Zephyr repository:

Ubuntu

```
cd ~/zephyrproject/zephyr
west sdk install
```

macOS

```
cd ~/zephyrproject/zephyr
west sdk install
```

Windows

Batchfile

```
cd %HOMEPATH%\zephyrproject\zephyr
west sdk install
```

PowerShell

```
cd $Env:HOMEPATH\zephyrproject\zephyr
west sdk install
```

Tip

Use command options to choose the SDK installation destination or install only selected architecture toolchains. See `west sdk install --help` for details.

Note

If you want to install Zephyr SDK without using the `west sdk` command, please see *Zephyr SDK installation*.

2.1.5 Build the Blinky Sample

Note

blinky is compatible with most, but not all, boards. If your board does not meet Blinky's blinky-sample-requirements, then hello_world is a good alternative.

If you are unsure what name west uses for your board, use `west boards` to list all boards Zephyr supports. Your board's documentation page also shows the exact board target name to pass to `west build`.

Build the blinky with *west build*. Replace `<your-board-name>` with the name of your board:

Ubuntu

```
cd ~/zephyrproject/zephyr
west build -p always -b <your-board-name> samples/basic/blinky
```

macOS

```
cd ~/zephyrproject/zephyr
west build -p always -b <your-board-name> samples/basic/blinky
```

Windows

Batchfile

```
cd %HOMEPATH%\zephyrproject\zephyr
west build -p always -b <your-board-name> samples\basic\blinky
```

PowerShell

```
cd $Env:HOMEPATH\zephyrproject\zephyr
west build -p always -b <your-board-name> samples\basic\blinky
```

The `-p always` option forces a pristine build, which removes build output from any previous configuration. This avoids stale files when you are getting started. Later, you can use `-p auto` to let `west build` heuristics decide when a pristine build may be needed. See `west build -h` for details.

Note

A board may contain one or multiple SoCs and each SoC may contain one or more CPU clusters. When building for such boards, specify the SoC or CPU cluster for which the sample must be built. For example to build blinky for the cpuapp core on the nrf5340dk the board must be provided as: `nrf5340dk/nrf5340/cpuapp`. See also *Board terminology* for more details.

2.1.6 Flash the Sample

Connect your board, usually via USB, and turn it on if there's a power switch. If in doubt about what to do, check your board's page in boards, as some boards require a specific setup or procedure to flash them.

Flash the sample with *west flash*. This programs the application you just built onto the connected board:

```
west flash
```

Note

You may need to install additional *host tools* required by your board. The `west flash` command will print an error if any required dependencies are missing.

Note

On Linux, you may need to configure udev rules before flashing with a debug probe for the first time. See *Setting udev rules*.

If you're using blinky, the LED will start to blink as shown in this figure:

Fig. 1: Phytec reel_board running blinky

2.1.7 Next Steps

Here are some next steps for exploring Zephyr:

- Try other samples
- Learn about *Application Development* and the *west* tool
- Find out about west's *flashing and debugging* features, or more about *Flashing and Hardware Debugging* in general
- Check out *Beyond the Getting Started Guide* for additional setup alternatives and ideas
- Discover *Resources* for getting help from the Zephyr community

2.1.8 Asking for Help

Before asking for help, search this documentation, the Zephyr project's GitHub discussions and issues, and Discord chat history. Your question may already have an answer there. You can also ask the *chatbot* available from every page of this documentation.

- **Mailing Lists**: users@lists.zephyrproject.org is usually the right list to ask for help. Search archives and sign up here.
- **GitHub**: Use GitHub discussions for questions and GitHub issues for bugs and feature requests.
- **Discord**: You can join with this Discord invite.

When asking for help, include:

1. What you want to do
2. What you tried, including the commands you ran
3. What happened, including the full text output

Copy and paste text instead of sharing screenshots. For more than 5 lines of terminal output, source code, or logs on Discord or GitHub, create a snippet using three backticks.

2.2 Beyond the Getting Started Guide

The *Getting Started Guide* gives a straight-forward path to set up your Linux, macOS, or Windows environment for Zephyr development. In this document, we delve deeper into Zephyr development setup issues and alternatives.

2.2.1 Python and pip

Python 3 and its package manager, pip[1], are used extensively by Zephyr to install and run scripts required to compile and run Zephyr applications, set up and maintain the Zephyr development environment, and build project documentation.

Depending on your operating system, you may need to provide the `--user` flag to the `pip3` command when installing new packages. This is documented throughout the instructions. See Installing Packages in the Python Packaging User Guide for more information about pip[1], including information on -\-user.

- On Linux, make sure `~/.local/bin` is at the front of your *PATH environment variable*, or programs installed with `--user` won't be found. Installing with `--user` avoids conflicts between pip and the system package manager, and is the default on Debian-based distributions.
- On macOS, Homebrew disables -\-user.
- On Windows, see the Installing Packages information on `--user` if you require using this option.

On all operating systems, pip's `-U` flag installs or updates the package if the package is already installed locally but a more recent version is available. It is good practice to use this flag if the latest version of a package is required. (Check the scripts/requirements.txt file to see if a specific Python package version is expected.)

2.2.2 Advanced Platform Setup

Here are some alternative instructions for more advanced platform setup configurations for supported development platforms:

Install Linux Host Dependencies

Documentation is available for these Linux distributions:

- Ubuntu
- Fedora
- Clear Linux
- Arch Linux

For distributions that are not based on rolling releases, some of the requirements and dependencies may not be met by your package manager. In that case please follow the additional instructions that are provided to find software from sources other than the package manager.

> **Note**
>
> If you're working behind a corporate firewall, you'll likely need to configure a proxy for accessing the internet, if you haven't done so already. While some tools use the environment

[1] pip is Python's package installer. Its `install` command first tries to reuse packages and package dependencies already installed on your computer. If that is not possible, `pip install` downloads them from the Python Package Index (PyPI) on the Internet.

The package versions requested by Zephyr's `requirements.txt` may conflict with other requirements on your system, in which case you may want to set up a virtualenv for Zephyr development.

variables `http_proxy` and `https_proxy` to get their proxy settings, some use their own configuration files, most notably `apt` and `git`.

Update Your Operating System Ensure your host system is up to date.

Ubuntu

```
sudo apt-get update
sudo apt-get upgrade
```

Fedora

```
sudo dnf upgrade
```

Clear Linux

```
sudo swupd update
```

Arch Linux

```
sudo pacman -Syu
```

Install Requirements and Dependencies Note that both Ninja and Make are installed with these instructions; you only need one.

Ubuntu

```
sudo apt-get install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-utils␣
↪file \
  make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1
```

Fedora

```
sudo dnf group install development-tools c-development
sudo dnf install cmake ninja-build gperf dfu-util dtc wget which \
  python3-pip python3-tkinter xz file python3-devel SDL2-devel \
  libusb1-devel
```

Clear Linux

```
sudo swupd bundle-add c-basic dev-utils dfu-util dtc \
  os-core-dev python-basic python3-basic python3-tcl
```

The Clear Linux focus is on *native* performance and security and not cross-compilation. For that reason it uniquely exports by default to the *environment* of all users a list of compiler and linker flags. Zephyr's CMake build system will either warn or fail because of these. To clear the C/C++ flags among these and fix the Zephyr build, run the following command as root then log out and back in:

```
echo 'unset CFLAGS CXXFLAGS' >> /etc/profile.d/unset_cflags.sh
```

Note this command unsets the C/C++ flags for *all users on the system*. Each Linux distribution has a unique, relatively complex and potentially evolving sequence of bash initialization files sourcing each other and Clear Linux is no exception. If you need a more flexible solution, start by looking at the logic in `/usr/share/defaults/etc/profile`.

Arch Linux

```
sudo pacman -S git cmake ninja gperf ccache dfu-util dtc wget \
    python-pip python-setuptools python-wheel tk xz file make which
```

CMake A *recent CMake version* is required. Check what version you have by using `cmake --version`. If you have an older version, there are several ways of obtaining a more recent one:

- On Ubuntu, you can follow the instructions for adding the kitware third-party apt repository to get an updated version of cmake using apt.
- Download and install a packaged cmake from the CMake project site. (Note this won't uninstall the previous version of cmake.)

  ```
  cd ~
  wget␣
  ↪https://github.com/Kitware/CMake/releases/download/v3.21.1/cmake-3.21.1-Linux-x86_64.sh
  chmod +x cmake-3.21.1-Linux-x86_64.sh
  sudo ./cmake-3.21.1-Linux-x86_64.sh --skip-license --prefix=/usr/local
  hash -r
  ```

  The `hash -r` command may be necessary if the installation script put cmake into a new location on your PATH.
- Download and install from the pre-built binaries provided by the CMake project itself in the CMake Downloads page. For example, to install version 3.21.1 in `~/bin/cmake`:

  ```
  mkdir $HOME/bin/cmake && cd $HOME/bin/cmake
  wget␣
  ↪https://github.com/Kitware/CMake/releases/download/v3.21.1/cmake-3.21.1-Linux-x86_64.sh
  yes | sh cmake-3.21.1-Linux-x86_64.sh | cat
  echo "export PATH=$PWD/cmake-3.21.1-Linux-x86_64/bin:\$PATH" >>
  ↪$HOME/.zephyrrc
  ```

- Use `pip3`:

  ```
  pip3 install --user cmake
  ```

  Note this won't uninstall the previous version of cmake and will install the new cmake into your ~/.local/bin folder so you'll need to add ~/.local/bin to your PATH. (See *Python and pip* for details.)
- Check your distribution's beta or unstable release package library for an update.
- On Ubuntu you can also use snap to get the latest version available:

  ```
  sudo snap install cmake
  ```

After updating cmake, verify that the newly installed cmake is found using `cmake --version`. You might also want to uninstall the CMake provided by your package manager to avoid conflicts. (Use `whereis cmake` to find other installed versions.)

DTC (Device Tree Compiler) A *recent DTC version* is required. Check what version you have by using `dtc --version`. If you have an older version, either install a more recent one by building from source, or use the one that is bundled in the *Zephyr SDK* by installing it.

Python A *modern Python 3 version* is required. Check what version you have by using `python3 --version`.

If you have an older version, you will need to install a more recent Python 3. You can build from source, or use a backport from your distribution's package manager channels if one is available.

Isolating this Python in a virtual environment is recommended to avoid interfering with your system Python.

Install the Zephyr Software Development Kit (SDK) The Zephyr Software Development Kit (SDK) contains toolchains for each of Zephyr's supported architectures. It also includes additional host tools, such as custom QEMU and OpenOCD.

Use of the Zephyr SDK is highly recommended and may even be required under certain conditions (for example, running tests in QEMU for some architectures).

To install the SDK, follow the Linux steps from the Zephyr SDK installation guide.

Building on Linux without the Zephyr SDK The Zephyr SDK is provided for convenience and ease of use. It provides toolchains for all Zephyr target architectures, and does not require any extra flags when building applications or running tests. In addition to cross-compilers, the Zephyr SDK also provides prebuilt host tools. It is, however, possible to build without the SDK's toolchain by using another toolchain as described in the *Toolchains* section.

As already noted above, the SDK also includes prebuilt host tools. To use the SDK's prebuilt host tools with a toolchain from another source, you must set the `ZEPHYR_SDK_INSTALL_DIR` environment variable to the Zephyr SDK installation directory. To build without the Zephyr SDK's prebuilt host tools, the `ZEPHYR_SDK_INSTALL_DIR` environment variable must be unset.

To make sure this variable is unset, run:

```
unset ZEPHYR_SDK_INSTALL_DIR
```

macOS alternative setup instructions

Important note about Gatekeeper Starting with macOS 10.15 Catalina, applications launched from the macOS Terminal application (or any other terminal emulator) are subject to the same system security policies that are applied to applications launched from the Dock. This means that if you download executable binaries using a web browser, macOS will not let you execute those from the Terminal by default. In order to get around this issue you can take two different approaches:

- Run `xattr -r -d com.apple.quarantine /path/to/folder` where `path/to/folder` is the path to the enclosing folder where the executables you want to run are located.
- Open *System Preferences* ▸ *Security and Privacy* ▸ *Privacy* and then scroll down to "Developer Tools". Then unlock the lock to be able to make changes and check the checkbox corresponding to your terminal emulator of choice. This will apply to any executable being launched from such terminal program.

Note that this section does **not** apply to executables installed with Homebrew, since those are automatically un-quarantined by `brew` itself. This is however relevant for most *Toolchains*.

Additional notes for MacPorts users While MacPorts is not officially supported in this guide, it is possible to use MacPorts instead of Homebrew to get all the required dependencies on macOS. Note also that you may need to install `rust` and `cargo` for the Python dependencies to install correctly.

Windows alternative setup instructions

Windows 10 WSL (Windows Subsystem for Linux) If you are running a recent version of Windows 10 you can make use of the built-in functionality to natively run Ubuntu binaries directly on a standard command-prompt. This allows you to use software such as the *Zephyr SDK* without setting up a virtual machine.

Warning

Windows 10 version 1803 has an issue that will cause CMake to not work properly and is fixed in version 1809 (and later). More information can be found in Zephyr Issue 10420.

1. Install the Windows Subsystem for Linux (WSL).

   Note

   For the Zephyr SDK to function properly you will need Windows 10 build 15002 or greater. You can check which Windows 10 build you are running in the "About your PC" section of the System Settings. If you are running an older Windows 10 build you might need to install the Creator's Update.

2. Follow the Ubuntu instructions in the *Install Linux Host Dependencies* document.

2.2.3 Install a Toolchain

Zephyr binaries are compiled and linked by a *toolchain* comprised of a cross-compiler and related tools which are different from the compiler and tools used for developing software that runs natively on your host operating system.

You can install the *Zephyr SDK* to get toolchains for all supported architectures, or install an *alternate toolchain* recommended by the SoC vendor or a specific board (check your specific board-level documentation).

You can configure the Zephyr build system to use a specific toolchain by setting *environment variables* such as `ZEPHYR_TOOLCHAIN_VARIANT` to a supported value, along with additional variable(s) specific to the toolchain variant.

2.2.4 Updating the Zephyr SDK toolchain

When updating Zephyr SDK, check whether the `ZEPHYR_TOOLCHAIN_VARIANT` or `ZEPHYR_SDK_INSTALL_DIR` environment variables are already set.

- If the variables are not set, the latest compatible version of Zephyr SDK will be selected by default. Proceed to next step without making any changes.
- If `ZEPHYR_TOOLCHAIN_VARIANT` is set, the corresponding toolchain will be selected at build time. Zephyr SDK is identified by the value `zephyr`. If the `ZEPHYR_TOOLCHAIN_VARIANT` environment variable is not `zephyr`, then either unset it or change its value to `zephyr` to make sure Zephyr SDK is selected.
- If the `ZEPHYR_SDK_INSTALL_DIR` environment variable is set, it will override the default lookup location for Zephyr SDK. If you install Zephyr SDK to one of the *recommended locations*, you can unset this variable. Otherwise, set it to your chosen install location.

For more information about these environment variables in Zephyr, see *Important Environment Variables*.

2.2.5 Cloning the Zephyr Repositories

The Zephyr project source is maintained in the GitHub zephyr repo. External modules used by Zephyr are found in the parent GitHub Zephyr project. Because of these dependencies, it's convenient to use the Zephyr-created *west* tool to fetch and manage the Zephyr and external module source code. See *Basics* for more details.

Once your development tools are installed, use *West (Zephyr's meta-tool)* to create, initialize, and download sources from the zephyr and external module repos. We'll use the name `zephyrproject`, but you can choose any name that does not contain a space anywhere in the path.

```
west init zephyrproject
cd zephyrproject
west update
```

The `west update` command fetches and keeps *Modules (External projects)* in the `zephyrproject` folder in sync with the code in the local zephyr repo.

Warning

You must run `west update` any time the `zephyr/west.yml` changes, caused, for example, when you pull the `zephyr` repository, switch branches in it, or perform a `git bisect` inside of it.

Keeping Zephyr updated

To update the Zephyr project source code, you need to get the latest changes via `git`. Afterwards, run `west update` as mentioned in the previous paragraph. Additionally, check for updated or added Python dependencies.

Linux/macOS

```
# replace zephyrproject with the path you gave west init
cd zephyrproject/zephyr
git pull
west update
west packages pip --install
```

Windows

Batchfile

```
:: replace zephyrproject with the path you gave west init
cd zephyrproject\zephyr
git pull
west update
cmd /c scripts\utils\west-packages-pip-install.cmd
```

PowerShell

```
# replace zephyrproject with the path you gave west init
cd zephyrproject\zephyr
git pull
west update
python -m pip install @((west packages pip) -split ' ')
```

2.2.6 Export Zephyr CMake package

The *Zephyr CMake Package* can be exported to CMake's user package registry if it has not already been done as part of *Getting Started Guide*.

2.2.7 Board Aliases

Developers who work with multiple boards may find explicit board names cumbersome and want to use aliases for common targets. This is supported by a CMake file with content like this:

```
# Variable foo_BOARD_ALIAS=bar replaces BOARD=foo with BOARD=bar and
# sets BOARD_ALIAS=foo in the CMake cache.
set(pca10028_BOARD_ALIAS nrf51dk/nrf51822)
set(pca10056_BOARD_ALIAS nrf52840dk/nrf52840)
set(k64f_BOARD_ALIAS frdm_k64f)
set(sltb004a_BOARD_ALIAS efr32mg_sltb004a)
```

and specifying its location in *ZEPHYR_BOARD_ALIASES*. This enables use of aliases `pca10028` in contexts like `cmake -DBOARD=pca10028` and `west -b pca10028`.

2.2.8 Build and Run an Application

You can build, flash, and run Zephyr applications on real hardware using a supported host system. Depending on your operating system, you can also run it in emulation with QEMU, or as a native application with native_sim. Additional information about building applications can be found in the *Building an Application* section.

Build Blinky

Let's build the blinky sample application.

Zephyr applications are built to run on specific hardware, called a "board"[2]. We'll use the Phytec reel_board here, but you can change the `reel_board` build target to another value if you have a different board. See boards or run `west boards` from anywhere inside the `zephyrproject` directory for a list of supported boards.

1. Go to the zephyr repository:

   ```
   cd zephyrproject/zephyr
   ```

2. Build the blinky sample for the `reel_board`:

   ```
   # From the root of the zephyr repository
   west build -b reel_board samples/basic/blinky
   ```

The main build products will be in `build/zephyr`; `build/zephyr/zephyr.elf` is the blinky application binary in ELF format. Other binary formats, disassembly, and map files may be present depending on your board.

The other sample applications in the samples folder are documented in samples.

> **Note**
>
> If you want to reuse an existing build directory for another board or application, you need to add the parameter `-p=auto` to `west build` to clean out settings and artifacts from the previous build.

Run the Application by Flashing to a Board

Most hardware boards supported by Zephyr can be flashed by running `west flash`. This may require board-specific tool installation and configuration to work properly.

See *Run an Application* and your specific board's documentation in boards for additional details.

[2] This has become something of a misnomer over time. While the target can be, and often is, a microprocessor running on its own dedicated hardware board, Zephyr also supports using QEMU to run targets built for other architectures in emulation, targets which produce native host system binaries that implement Zephyr's driver interfaces with POSIX APIs, and even running different Zephyr-based binaries on CPU cores of differing architectures on the same physical chip. Each of these hardware configurations is called a "board," even though that doesn't always make perfect sense in context.

Setting udev rules

Flashing a board requires permission to directly access the board hardware, usually managed by installation of the flashing tools. On Linux systems, if the `west flash` command fails, you likely need to define udev rules to grant the needed access permission.

Udev is a device manager for the Linux kernel and the udev daemon handles all user space events raised when a hardware device is added (or removed) from the system. We can add a rules file to grant access permission by non-root users to certain USB-connected devices.

The OpenOCD (On-Chip Debugger) project conveniently provides a rules file that defined board-specific rules for most Zephyr-supported arm-based boards, so we recommend installing this rules file by downloading it from their sourceforge repo, or if you've installed the Zephyr SDK there is a copy of this rules file in the SDK folder:

- Either download the OpenOCD rules file and copy it to the right location:

```
wget -O 60-openocd.rules␣
↪https://sf.net/p/openocd/code/ci/master/tree/contrib/60-openocd.rules?format=raw
sudo cp 60-openocd.rules /etc/udev/rules.d
```

- or copy the rules file from the Zephyr SDK folder:

```
sudo cp $
↪{ZEPHYR_SDK_INSTALL_DIR}/sysroots/x86_64-pokysdk-linux/usr/share/openocd/contrib/60-openoc
↪/etc/udev/rules.d
```

Then, in either case, ask the udev daemon to reload these rules:

```
sudo udevadm control --reload
```

Unplug and plug in the USB connection to your board, and you should have permission to access the board hardware for flashing. Check your board-specific documentation (boards) for further information if needed.

Run the Application in QEMU

You can run Zephyr applications via emulation on your host system using QEMU when targeting either the x86 or ARM Cortex-M3 architectures. QEMU is included with the Zephyr SDK.

If using a manual QEMU installation, ensure it is available in the system `PATH` environment variable.

`QEMU_BIN_PATH` can be used as an optional override to specify the location of QEMU binaries used by Twister. When provided, Twister validates that the specified path exists. Otherwise, Twister relies on SDK or other QEMU discovery mechanisms.

For example, you can build and run the hello_world sample using the x86 emulation board configuration (`qemu_x86`), with:

```
# From the root of the zephyr repository
west build -b qemu_x86 samples/hello_world
west build -t run
```

To exit QEMU, type `Ctrl-a`, then `x`.

Use `qemu_cortex_m3` to target an emulated Arm Cortex-M3 sample.

Run a Sample Application natively (Linux)

You can compile some samples to run as host programs on Linux. See native_sim for more information. On 64-bit host operating systems, you need to install a 32-bit C library, or build targeting native_sim/native/64.

First, build Hello World for `native_sim`.

```
# From the root of the zephyr repository
west build -b native_sim samples/hello_world
```

Next, run the application.

```
west build -t run
# or just run zephyr.exe directly:
./build/zephyr/zephyr.exe
```

Press `Ctrl-C` to exit.

You can run `./build/zephyr/zephyr.exe --help` to get a list of available options.

This executable can be instrumented using standard tools, such as gdb or valgrind.

2.3 Environment Variables

Various pages in this documentation refer to setting Zephyr-specific environment variables. This page describes how.

2.3.1 Setting Variables

Option 1: Just Once

To set the environment variable `MY_VARIABLE` to `foo` for the lifetime of your current terminal window:

Linux/macOS

```
export MY_VARIABLE=foo
```

Windows

```
set MY_VARIABLE=foo
```

Warning

This is best for experimentation. If you close your terminal window, use another terminal window or tab, restart your computer, etc., this setting will be lost forever.

Using options 2 or 3 is recommended if you want to keep using the setting.

Option 2: In all Terminals

Linux/macOS

Add the `export MY_VARIABLE=foo` line to your shell's startup script in your home directory. For Bash, this is usually `~/.bashrc` on Linux or `~/.bash_profile` on macOS. Changes in these startup scripts don't affect shell instances already started; try opening a new terminal window to get the new settings.

Windows

You can use the `setx` program in `cmd.exe` or the third-party RapidEE program.

To use `setx`, type this command, then close the terminal window. Any new `cmd.exe` windows will have `MY_VARIABLE` set to `foo`.

```
setx MY_VARIABLE foo
```

To install RapidEE, a freeware graphical environment variable editor, using Chocolatey in an Administrator command prompt:

```
choco install rapidee
```

You can then run `rapidee` from your terminal to launch the program and set environment variables. Make sure to use the "User" environment variables area – otherwise, you have to run RapidEE as administrator. Also make sure to save your changes by clicking the Save button at top left before exiting. Settings you make in RapidEE will be available whenever you open a new terminal window.

Option 3: Using zephyrrc files

Choose this option if you don't want to make the variable's setting available to all of your terminals, but still want to save the value for loading into your environment when you are using Zephyr.

Linux/macOS

Zephyr supports multiple locations for your `zephyrrc` file, following the XDG Base Directory Specification when possible. Create a zephyrrc file in one of the following locations (they will be checked in order):

1. `$XDG_CONFIG_HOME/zephyr/zephyrrc`
2. `$HOME/.config/zephyr/zephyrrc`
3. `$HOME/.zephyrrc`

Add this line to the file in your preferred location:

```
export MY_VARIABLE=foo
```

To get this value back into your current terminal environment, **you must run** `source zephyr-env.sh` from the main `zephyr` repository. Among other things, this script sources your `zephyrrc` (the first one it finds from the list of locations above).

The value will be lost if you close the window, etc.; run `source zephyr-env.sh` again to get it back.

Windows

Add the line `set MY_VARIABLE=foo` to the file `%userprofile%\zephyrrc.cmd` using a text editor such as Notepad to save the value.

To get this value back into your current terminal environment, **you must run** `zephyr-env.cmd` in a `cmd.exe` window after changing directory to the main `zephyr` repository. Among other things, this script runs `%userprofile%\zephyrrc.cmd`.

The value will be lost if you close the window, etc.; run `zephyr-env.cmd` again to get it back.

These scripts:

- set *ZEPHYR_BASE* to the location of the zephyr repository
- adds some Zephyr-specific locations (such as zephyr's `scripts` directory) to your *PATH* environment variable
- loads any settings from the `zephyrrc` files described above in *Option 3: Using zephyrrc files*.

You can thus use them any time you need any of these settings.

2.3.2 Zephyr Environment Scripts

You can use the zephyr repository scripts `zephyr-env.sh` (for macOS and Linux) and `zephyr-env.cmd` (for Windows) to load Zephyr-specific settings into your current terminal's environment. To do so, run this command from the zephyr repository:

Linux/macOS

```
source zephyr-env.sh
```

Windows

```
zephyr-env.cmd
```

These scripts:

- set *ZEPHYR_BASE* to the location of the zephyr repository
- adds some Zephyr-specific locations (such as zephyr's `scripts` directory) to your `PATH` environment variable
- loads any settings from the `zephyrrc` files described above in *Option 3: Using zephyrrc files*.

You can thus use them any time you need any of these settings.

2.3.3 Important Environment Variables

Some *Important Build System Variables* can also be set in the environment. Here is a description of some of these important environment variables. This is not a comprehensive list.

BOARD
: See *Important Build System Variables*.

CONF_FILE
: See *Important Build System Variables*.

SHIELD
: See *Shields*.

ZEPHYR_BASE
: See *Important Build System Variables*.

EXTRA_ZEPHYR_MODULES
: See *Important Build System Variables*.

ZEPHYR_MODULES
: See *Important Build System Variables*.

ZEPHYR_BOARD_ALIASES
: See *Board Aliases*

The following additional environment variables are significant when configuring the *toolchain* used to build Zephyr applications.

ZEPHYR_SDK_INSTALL_DIR
: Path where Zephyr SDK is installed.

ZEPHYR_TOOLCHAIN_VARIANT
: The name of the toolchain to use.

{TOOLCHAIN}_TOOLCHAIN_PATH
: Path to the toolchain specified by *ZEPHYR_TOOLCHAIN_VARIANT*. For example, if `ZEPHYR_TOOLCHAIN_VARIANT=host/llvm`, use `LLVM_TOOLCHAIN_PATH`. (Note the capitalization when forming the environment variable name.)

You might need to update some of these variables when you *update the Zephyr SDK toolchain*.

Emulators and boards may also depend on additional programs. The build system will try to locate those programs automatically, but may rely on additional CMake or environment variables to do so. Please consult your emulator's or board's documentation for more information. The following environment variables may be useful in such situations:

`PATH`

`PATH` is an environment variable used on Unix-like or Microsoft Windows operating systems to specify a set of directories where executable programs are located.

2.4 Application Development

Note

In this document, we'll assume:

- your **application directory**, `<app>`, is something like `<home>/zephyrproject/app`
- its **build directory** is `<app>/build`

These terms are defined below. On Linux/macOS, `<home>` is equivalent to `~`. On Windows, it's `%userprofile%`.

Keeping your application inside the workspace (`<home>/zephyrproject`) makes it easier to use `west build` and other commands with it. (You can put your application anywhere as long as *ZEPHYR_BASE* is set appropriately, though.)

2.4.1 Overview

Zephyr's build system is based on CMake.

The build system is application-centric, and requires Zephyr-based applications to initiate building the Zephyr source code. The application build controls the configuration and build process of both the application and Zephyr itself, compiling them into a single binary.

The main zephyr repository contains Zephyr's source code, configuration files, and build system. You also likely have installed various *Modules (External projects)* alongside the zephyr repository, which provide third party source code integration.

The files in the **application directory** link Zephyr and any modules with the application. This directory contains all application-specific files, such as application-specific configuration files and source code.

Here are the files in a simple Zephyr application:

```
<app>
├── CMakeLists.txt
├── app.overlay
├── prj.conf
├── VERSION
└── src
    └── main.c
```

These contents are:

- **CMakeLists.txt**: This file tells the build system where to find the other application files, and links the application directory with Zephyr's CMake build system. This link provides features supported by Zephyr's build system, such as board-specific configuration files, the ability to run and debug compiled binaries on real or emulated hardware, and more.

- **app.overlay**: This is a devicetree overlay file that specifies application-specific changes which should be applied to the base devicetree for any board you build for. The purpose of devicetree overlays is usually to configure something about the hardware used by the application.

  The build system looks for `app.overlay` by default, but you can add more devicetree overlays, and other default files are also searched for.

  See *Devicetree* for more information about devicetree.

- **prj.conf**: This is a Kconfig fragment that specifies application-specific values for one or more Kconfig options. These application settings are merged with other settings to produce the final configuration. The purpose of Kconfig fragments is usually to configure the software features used by the application.

  The build system looks for `prj.conf` by default, but you can add more Kconfig fragments, and other default files are also searched for.

  See *Kconfig Configuration* below for more information.

- **VERSION**: A text file that contains several version information fields. These fields let you manage the lifecycle of the application and automate providing the application version when signing application images.

  See *Application version management* for more information about this file and how to use it.

- **main.c**: A source code file. Applications typically contain source files written in C, C++, or assembly language. The Zephyr convention is to place them in a subdirectory of `<app>` named `src`.

Once an application has been defined, you will use CMake to generate a **build directory**, which contains the files you need to build the application and Zephyr, then link them together into a final binary you can run on your board. The easiest way to do this is with *west build*, but you can use CMake directly also. Application build artifacts are always generated in a separate build directory: Zephyr does not support "in-tree" builds.

The following sections describe how to create, build, and run Zephyr applications, followed by more detailed reference material.

2.4.2 Application types

We distinguish three basic types of Zephyr application based on where `<app>` is located:

Application type	`<app>` location
repository	zephyr repository
workspace	west workspace where Zephyr is installed
freestanding	other locations

We'll discuss these more below. To learn how the build system supports each type, see *Zephyr CMake Package*.

Zephyr repository application

An application located within the `zephyr` source code repository in a Zephyr *west workspace* is referred to as a Zephyr repository application. In the following example, the hello_world sample is a Zephyr repository application:

```
zephyrproject/
├── .west/
│   └── config
└── zephyr/
```

(continues on next page)

(continued from previous page)

```
     ├── arch/
     ├── boards/
     ├── cmake/
     ├── samples/
     │    ├── hello_world/
     │    └── ...
     ├── tests/
     └── ...
```

Zephyr workspace application

An application located within a *workspace*, but outside the zephyr repository itself, is referred to as a Zephyr workspace application. In the following example, `app` is a Zephyr workspace application:

```
zephyrproject/
├── .west/
│    └── config
├── zephyr/
├── bootloader/
├── modules/
├── tools/
├── <vendor/private-repositories>/
└── applications/
     └── app/
```

Zephyr freestanding application

A Zephyr application located outside of a Zephyr *workspace* is referred to as a Zephyr freestanding application. In the following example, `app` is a Zephyr freestanding application:

```
<home>/
├── zephyrproject/
│     ├── .west/
│     │    └── config
│     ├── zephyr/
│     ├── bootloader/
│     ├── modules/
│     └── ...
│
└── app/
     ├── CMakeLists.txt
     ├── prj.conf
     └── src/
         └── main.c
```

2.4.3 Creating an Application

In Zephyr, you can either use a reference workspace application or create your application by hand.

Using a Reference Workspace Application

The example-application Git repository contains a reference *workspace application*. It is recommended to use it as a reference when creating your own application as described in the following sections.

The example-application repository demonstrates how to use several commonly-used features, such as:

- Custom *board ports*
- Custom *devicetree bindings*
- Custom *device drivers*
- Continuous Integration (CI) setup, including using *twister*
- A custom west *extension command*

Basic example-application Usage The easiest way to get started with the example-application repository within an existing Zephyr workspace is to follow these steps:

```
cd <home>/zephyrproject
git clone https://github.com/zephyrproject-rtos/example-application my-app
```

The directory name `my-app` above is arbitrary: change it as needed. You can now go into this directory and adapt its contents to suit your needs. Since you are using an existing Zephyr workspace, you can use `west build` or any other west commands to build, flash, and debug.

Advanced example-application Usage You can also use the example-application repository as a starting point for building your own customized Zephyr-based software distribution. This lets you do things like:

- remove Zephyr modules you don't need
- add additional custom repositories of your own
- override repositories provided by Zephyr with your own versions
- share the results with others and collaborate further

The example-application repository contains a `west.yml` file and is therefore also a west *manifest repository*. Use this to create a new, customized workspace by following these steps:

```
cd <home>
mkdir my-workspace
cd my-workspace
git clone https://github.com/zephyrproject-rtos/example-application␣
↪my-manifest-repo
west init -l my-manifest-repo
```

This will create a new workspace with the *T2 topology*, with `my-manifest-repo` as the manifest repository. The `my-workspace` and `my-manifest-repo` names are arbitrary: change them as needed.

Next, customize the manifest repository. The initial contents of this repository will match the example-application's contents when you clone it. You can then edit `my-manifest-repo/west.yml` to your liking, changing the set of repositories in it as you wish. See *Manifest Imports* for many examples of how to add or remove different repositories from your workspace as needed. Make any other changes you need to other files.

When you are satisfied, you can run:

```
west update
```

and your workspace will be ready for use.

If you push the resulting `my-manifest-repo` repository somewhere else, you can share your work with others. For example, let's say you push the repository to `https://git.example.com/my-manifest-repo`. Other people can then set up a matching workspace by running:

```
west init -m https://git.example.com/my-manifest-repo my-workspace
cd my-workspace
west update
```

From now on, you can collaborate on the shared software by pushing changes to the repositories you are using and updating `my-manifest-repo/west.yml` as needed to add and remove repositories, or change their contents.

Creating an Application by Hand

You can follow these steps to create a basic application directory from scratch. However, using the example-application repository or one of Zephyr's samples as a starting point is likely to be easier.

1. Create an application directory.

   For example, in a Unix shell or Windows `cmd.exe` prompt:

   ```
   mkdir app
   ```

   Warning

   Building Zephyr or creating an application in a directory with spaces anywhere on the path is not supported. So the Windows path `C:\Users\YourName\app` will work, but `C:\Users\Your Name\app` will not.

2. Create your source code files.

   It's recommended to place all application source code in a subdirectory named `src`. This makes it easier to distinguish between project files and sources.

   Continuing the previous example, enter:

   ```
   cd app
   mkdir src
   ```

3. Place your application source code in the `src` sub-directory. For this example, we'll assume you created a file named `src/main.c`.

4. Create a file named `CMakeLists.txt` in the `app` directory with the following contents:

   ```
   cmake_minimum_required(VERSION 3.28.0)

   find_package(Zephyr)
   project(my_zephyr_app)

   target_sources(app PRIVATE src/main.c)
   ```

   Notes:

   - The `cmake_minimum_required()` call is required by CMake. It is also invoked by the Zephyr package on the next line. CMake will error out if its version is older than either the version in your `CMakeLists.txt` or the version number in the Zephyr package.
   - `find_package(Zephyr)` pulls in the Zephyr build system, which creates a CMake target named `app` (see *Zephyr CMake Package*). Adding sources to this target is how you include them in the build. The Zephyr package will define `Zephyr-Kernel` as a CMake project and enable support for the `C`, `CXX`, `ASM` languages.

- `project(my_zephyr_app)` defines your application's CMake project. This must be called after `find_package(Zephyr)` to avoid interference with Zephyr's `project(Zephyr-Kernel)`.
- `target_sources(app PRIVATE src/main.c)` is to add your source file to the `app` target. This must come after `find_package(Zephyr)` which defines the target. You can add as many files as you want with `target_sources()`.

5. Create at least one Kconfig fragment for your application (usually named `prj.conf`) and set Kconfig option values needed by your application there. See *Kconfig Configuration*. If no Kconfig options need to be set, create an empty file.
6. Configure any devicetree overlays needed by your application, usually in a file named `app.overlay`. See *Set devicetree overlays*.
7. Set up any other files you may need, such as *twister* configuration files, continuous integration files, documentation, etc.

2.4.4 Important Build System Variables

You can control the Zephyr build system using many variables. This section describes the most important ones that every Zephyr developer should know about.

Note

The variables **BOARD**, **CONF_FILE**, and **DTC_OVERLAY_FILE** can be supplied to the build system in 3 ways (in order of precedence):

- As a parameter to the `west build` or `cmake` invocation via the `-D` command-line switch. If you have multiple overlay files, you should use quotations, `"file1.overlay;file2.overlay"`
- As *Environment Variables*.
- As a `set(<VARIABLE> <VALUE>)` statement in your `CMakeLists.txt`

- **ZEPHYR_BASE**: Zephyr base variable used by the build system. `find_package(Zephyr)` will automatically set this as a cached CMake variable. But `ZEPHYR_BASE` can also be set as an environment variable in order to force CMake to use a specific Zephyr installation.
- **BOARD**: Selects the board that the application's build will use for the default configuration. See boards for built-in boards, and *Board Porting Guide* for information on adding board support.
- **CONF_FILE**: Indicates the name of one or more Kconfig configuration fragment files. Multiple filenames can be separated with either spaces or semicolons. Each file includes Kconfig configuration values that override the default configuration values.

  See *The Initial Configuration* for more information.
- **EXTRA_CONF_FILE**: Additional Kconfig configuration fragment files. Multiple filenames can be separated with either spaces or semicolons. This can be useful in order to leave **CONF_FILE** at its default value, but "mix in" some additional configuration options.
- **DTC_OVERLAY_FILE**: One or more devicetree overlay files to use. Multiple files can be separated with semicolons. See *Set devicetree overlays* for examples and *Introduction to devicetree* for information about devicetree and Zephyr.
- **EXTRA_DTC_OVERLAY_FILE**: Additional devicetree overlay files to use. Multiple files can be separated with semicolons. This can be useful to leave **DTC_OVERLAY_FILE** at its default value, but "mix in" some additional overlay files.
- **SHIELD**: see *Shields*

- **ZEPHYR_MODULES**: A CMake list containing absolute paths of additional directories with source code, Kconfig, etc. that should be used in the application build. See *Modules (External projects)* for details. If you set this variable, it must be a complete list of all modules to use, as the build system will not automatically pick up any modules from west.
- **EXTRA_ZEPHYR_MODULES**: Like **ZEPHYR_MODULES**, except these will be added to the list of modules found via west, instead of replacing it.
- **FILE_SUFFIX**: Optional suffix for filenames that will be added to Kconfig fragments and devicetree overlays (if these files exists, otherwise will fallback to the name without the prefix). See *File Suffixes* for details.
- **KCONFIG_WARNING_AS_ERROR**: Treat every Kconfig warning as an error, including the ones which are only printed by default. See *Treating Kconfig warnings as errors* for details.

Note

You can use a *Zephyr Build Configuration CMake packages* to share common settings for these variables.

2.4.5 Application CMakeLists.txt

Every application must have a `CMakeLists.txt` file. This file is the entry point, or top level, of the build system. The final `zephyr.elf` image contains both the application and the kernel libraries.

This section describes some of what you can do in your `CMakeLists.txt`. Make sure to follow these steps in order.

1. If you only want to build for one board, add the name of the board configuration for your application on a new line. For example:

   ```
   set(BOARD qemu_x86)
   ```

   Refer to boards for more information on available boards.

   The Zephyr build system determines a value for **BOARD** by checking the following, in order (when a BOARD value is found, CMake stops looking further down the list):

   - Any previously used value as determined by the CMake cache takes highest precedence. This ensures you don't try to run a build with a different **BOARD** value than you set during the build configuration step.
   - Any value given on the CMake command line (directly or indirectly via `west build`) using `-DBOARD=YOUR_BOARD` will be checked for and used next.
   - If an *environment variable* `BOARD` is set, its value will then be used.
   - Finally, if you set `BOARD` in your application `CMakeLists.txt` as described in this step, this value will be used.

2. If your application uses a configuration file or files other than the usual `prj.conf`, add lines setting the **CONF_FILE** variable to these files appropriately. If multiple filenames are given, separate them by a single space or semicolon. CMake lists can be used to build up configuration fragment files in a modular way when you want to avoid setting **CONF_FILE** in a single place. For example:

   ```
   set(CONF_FILE "fragment_file1.conf")
   list(APPEND CONF_FILE "fragment_file2.conf")
   ```

   See *The Initial Configuration* for more information.

3. If your application uses devicetree overlays, you may need to set *DTC_OVERLAY_FILE*. See *Set devicetree overlays*.

4. If your application has its own kernel configuration options, create a `Kconfig` file in the same directory as your application's `CMakeLists.txt`.

   See *the Kconfig section of the manual* for detailed Kconfig documentation.

   An (unlikely) advanced use case would be if your application has its own unique configuration **options** that are set differently depending on the build configuration.

   If you just want to set application specific **values** for existing Zephyr configuration options, refer to the `CONF_FILE` description above.

   Structure your `Kconfig` file like this:

```
# SPDX-License-Identifier: Apache-2.0

mainmenu "Your Application Name"

# Your application configuration options go here

# Sources Kconfig.zephyr in the Zephyr root directory.
#
# Note: All 'source' statements work relative to the Zephyr root directory␣
↪(due
# to the $srctree environment variable being set to $ZEPHYR_BASE). If you want
# to 'source' relative to the current Kconfig file instead, use 'rsource' (or␣
↪a
# path relative to the Zephyr root).
source "Kconfig.zephyr"
```

   Note

   Environment variables in `source` statements are expanded directly, so you do not need to define an `option env="ZEPHYR_BASE"` Kconfig "bounce" symbol. If you use such a symbol, it must have the same name as the environment variable.

   See *Kconfig extensions* for more information.

   The `Kconfig` file is automatically detected when placed in the application directory, but it is also possible for it to be found elsewhere if the CMake variable `KCONFIG_ROOT` is set with an absolute path.

5. Specify that the application requires Zephyr on a new line, **after any lines added from the steps above**:

```
find_package(Zephyr)
project(my_zephyr_app)
```

   Note

   `find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})` can be used if enforcing a specific Zephyr installation by explicitly setting the `ZEPHYR_BASE` environment variable should be supported. All samples in Zephyr supports the `ZEPHYR_BASE` environment variable.

6. Now add any application source files to the 'app' target library, each on their own line, like so:

```
target_sources(app PRIVATE src/main.c)
```

Below is a simple example `CMakeList.txt`:

```
set(BOARD qemu_x86)

find_package(Zephyr)
project(my_zephyr_app)

target_sources(app PRIVATE src/main.c)
```

The Cmake property `HEX_FILES_TO_MERGE` leverages the application configuration provided by Kconfig and CMake to let you merge externally built hex files with the hex file generated when building the Zephyr application. For example:

```
set_property(GLOBAL APPEND PROPERTY HEX_FILES_TO_MERGE
    ${app_bootloader_hex}
    ${PROJECT_BINARY_DIR}/${KERNEL_HEX_NAME}
    ${app_provision_hex})
```

2.4.6 CMakeCache.txt

CMake uses a CMakeCache.txt file as persistent key/value string storage used to cache values between runs, including compile and build options and paths to library dependencies. This cache file is created when CMake is run in an empty build folder.

For more details about the CMakeCache.txt file see the official CMake Cache documentation.

2.4.7 Application Configuration

Application Configuration Directory

Zephyr will use configuration files from the application's configuration directory except for files with an absolute path provided by the arguments described earlier, for example `CONF_FILE`, `EXTRA_CONF_FILE`, `DTC_OVERLAY_FILE`, and `EXTRA_DTC_OVERLAY_FILE`.

The application configuration directory is defined by the `APPLICATION_CONFIG_DIR` variable.

`APPLICATION_CONFIG_DIR` will be set by one of the sources below with the highest priority listed first.

1. If `APPLICATION_CONFIG_DIR` is specified by the user with `-DAPPLICATION_CONFIG_DIR=<path>` or in a CMake file before `find_package(Zephyr)` then this folder is used a the application's configuration directory.
2. The application's source directory.

Kconfig Configuration

Application configuration options are usually set in `prj.conf` in the application directory. For example, C++ support could be enabled with this assignment:

```
CONFIG_CPP=y
```

Looking at existing samples is a good way to get started.

See *Setting Kconfig configuration values* for detailed documentation on setting Kconfig configuration values. The *The Initial Configuration* section on the same page explains how the initial configuration is derived. See kconfig-search for a complete list of configuration options. See *Hardening Tool* for security information related with Kconfig options.

The other pages in the *Kconfig section of the manual* are also worth going through, especially if you planning to add new configuration options.

Experimental features Zephyr is a project under constant development and thus there are features that are still in early stages of their development cycle. Such features will be marked `[EXPERIMENTAL]` in their Kconfig title.

The `CONFIG_WARN_EXPERIMENTAL` setting can be used to enable warnings at CMake configure time if any experimental feature is enabled.

```
CONFIG_WARN_EXPERIMENTAL=y
```

For example, if option `CONFIG_FOO` is experimental, then enabling it and `CONFIG_WARN_EXPERIMENTAL` will print the following warning at CMake configure time when you build an application:

```
warning: Experimental symbol FOO is enabled.
```

Devicetree Overlays

See *Set devicetree overlays*.

File Suffixes

Zephyr applications might want to have a single code base with multiple configurations for different build/product variants which would necessitate different Kconfig options and devicetree configuration. In order to better configure this, Zephyr provides a **`FILE_SUFFIX`** option when configuring applications that can be automatically appended to filenames. This is applied to Kconfig fragments and board overlays but with a fallback so that if such files do not exist, the files without these suffixes will be used instead.

Given the following example project layout:

```
<app>
├── CMakeLists.txt
├── prj.conf
├── prj_mouse.conf
├── boards
│   ├── native_sim.overlay
│   └── qemu_cortex_m3_mouse.overlay
└── src
    └── main.c
```

- If this is built normally without `FILE_SUFFIX` being defined for `native_sim` then `prj.conf` and `boards/native_sim.overlay` will be used.
- If this is built normally without `FILE_SUFFIX` being defined for `qemu_cortex_m3` then `prj.conf` will be used, no application devicetree overlay will be used.
- If this is built with `FILE_SUFFIX` set to `mouse` for `native_sim` then `prj_mouse.conf` and `boards/native_sim.overlay` will be used (there is no `native_sim_mouse.overlay` file so it falls back to `native_sim.overlay`).
- If this is built with `FILE_SUFFIX` set to `mouse` for `qemu_cortex_m3` then `prj_mouse.conf` will be used and `boards/qemu_cortex_m3_mouse.overlay` will be used.

2.4.8 Application-Specific Code

Application-specific source code files are normally added to the application's `src` directory. If the application adds a large number of files the developer can group them into sub-directories under `src`, to whatever depth is needed.

Application-specific source code should not use symbol name prefixes that have been reserved by the kernel for its own use. For more information, see Naming Conventions.

Third-party Library Code

It is possible to build library code outside the application's `src` directory but it is important that both application and library code targets the same Application Binary Interface (ABI). On most architectures there are compiler flags that control the ABI targeted, making it important that both libraries and applications have certain compiler flags in common. It may also be useful for glue code to have access to Zephyr kernel header files.

To make it easier to integrate third-party components, the Zephyr build system has defined CMake functions that give application build scripts access to the zephyr compiler options. The functions are documented and defined in cmake/modules/extensions.cmake and follow the naming convention `zephyr_get_<type>_<format>`.

The following variables will often need to be exported to the third-party build system.

- `CMAKE_C_COMPILER`, `CMAKE_AR`.
- `ARCH` and `BOARD`, together with several variables that identify the Zephyr kernel version.

samples/application_development/external_lib is a sample project that demonstrates some of these features.

2.4.9 Building an Application

The Zephyr build system compiles and links all components of an application into a single application image that can be run on simulated hardware or real hardware.

Like any other CMake-based system, the build process takes place *in two stages*. First, build files (also known as a buildsystem) are generated using the `cmake` command-line tool while specifying a generator. This generator determines the native build tool the buildsystem will use in the second stage. The second stage runs the native build tool to actually build the source files and generate an image. To learn more about these concepts refer to the CMake introduction in the official CMake documentation.

Although the default build tool in Zephyr is *west*, Zephyr's meta-tool, which invokes `cmake` and the underlying build tool (`ninja` or `make`) behind the scenes, you can also choose to invoke `cmake` directly if you prefer. On Linux and macOS you can choose between the `make` and `ninja` generators (i.e. build tools), whereas on Windows you need to use `ninja`, since `make` is not supported on this platform. For simplicity we will use `ninja` throughout this guide, and if you choose to use `west build` to build your application know that it will default to `ninja` under the hood.

As an example, let's build the Hello World sample for the `reel_board`:

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild
```

On Linux and macOS, you can also build with `make` instead of `ninja`:

Using west:

- to use `make` just once, add `-- -G"Unix Makefiles"` to the west build command line; see the *west build* documentation for an example.
- to use `make` by default from now on, run `west config build.generator "Unix Makefiles"`.

Using CMake directly:

```
# From the root of the zephyr repository
# Use cmake to configure a Make-based buildsystem:
cmake -Bbuild -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
make -Cbuild
```

Basics

1. Navigate to the application directory <app>.
2. Enter the following commands to build the application's `zephyr.elf` image for the board specified in the command-line parameters:

   Using west:

   ```
   west build -b <board>
   ```

   Using CMake and ninja:

   ```
   mkdir build && cd build

   # Use cmake to configure a Ninja-based buildsystem:
   cmake -GNinja -DBOARD=<board> ..

   # Now run the build tool on the generated build system:
   ninja
   ```

   If desired, you can build the application using the configuration settings specified in an alternate `.conf` file using the `CONF_FILE` parameter. These settings will override the settings in the application's `.config` file or its default `.conf` file. For example:

   Using west:

   ```
   west build -b <board> -- -DCONF_FILE=prj.alternate.conf
   ```

   Using CMake and ninja:

   ```
   mkdir build && cd build
   cmake -GNinja -DBOARD=<board> -DCONF_FILE=prj.alternate.conf ..
   ninja
   ```

   As described in the previous section, you can instead choose to permanently set the board and configuration settings by either exporting **BOARD** and **CONF_FILE** environment variables or by setting their values in your `CMakeLists.txt` using `set()` statements. Additionally, `west` allows you to *set a default board*.

Build Directory Contents

When using the Ninja generator a build directory looks like this:

```
<app>/build
├── build.ninja
├── CMakeCache.txt
├── CMakeFiles
├── cmake_install.cmake
├── rules.ninja
└── zephyr
```

The most notable files in the build directory are:

- `build.ninja`, which can be invoked to build the application.
- A `zephyr` directory, which is the working directory of the generated build system, and where most generated files are created and stored.

After running `ninja`, the following build output files will be written to the `zephyr` sub-directory of the build directory. (This is **not the Zephyr base directory**, which contains the Zephyr source code etc. and is described above.)

- `.config`, which contains the configuration settings used to build the application.

> **Note**
>
> The previous version of `.config` is saved to `.config.old` whenever the configuration is updated. This is for convenience, as comparing the old and new versions can be handy.

- Various object files (`.o` files and `.a` files) containing compiled kernel and application code.
- `zephyr.elf`, which contains the final combined application and kernel binary. Other binary output formats, such as `.hex` and `.bin`, are also supported.

Rebuilding an Application

Application development is usually fastest when changes are continually tested. Frequently rebuilding your application makes debugging less painful as the application becomes more complex. It's usually a good idea to rebuild and test after any major changes to the application's source files, CMakeLists.txt files, or configuration settings.

> **Important**
>
> The Zephyr build system rebuilds only the parts of the application image potentially affected by the changes. Consequently, rebuilding an application is often significantly faster than building it the first time.

Sometimes the build system doesn't rebuild the application correctly because it fails to recompile one or more necessary files. You can force the build system to rebuild the entire application from scratch with the following procedure:

1. Open a terminal console on your host computer, and navigate to the build directory `<app>/build`.
2. Enter one of the following commands, depending on whether you want to use `west` or `cmake` directly to delete the application's generated files, except for the `.config` file that contains the application's current configuration information.

   ```
   west build -t clean
   ```

   or

   ```
   ninja clean
   ```

   Alternatively, enter one of the following commands to delete *all* generated files, including the `.config` files that contain the application's current configuration information for those board types.

   ```
   west build -t pristine
   ```

   or

```
ninja pristine
```

If you use west, you can take advantage of its capability to automatically *make the build folder pristine* whenever it is required.

3. Rebuild the application normally following the steps specified in *Building an Application* above.

Building for a board revision

The Zephyr build system has support for specifying multiple hardware revisions of a single board with small variations. Using revisions allows the board support files to make minor adjustments to a board configuration without duplicating all the files described in *Create your board directory* for each revision.

To build for a particular revision, use `<board>@<revision>` or `<board>@<revision>/<qualifiers>` instead of plain `<board>` or `<board>/<qualifiers>`. For example:

Using west:

```
west build -b nrf9160dk@0.14.0/nrf9160/ns
```

Using CMake and ninja:

```
mkdir build && cd build
cmake -GNinja -DBOARD=nrf9160dk@0.14.0/nrf9160/ns ..
ninja
```

Check your board's documentation for details on whether it has multiple revisions, and what revisions are supported.

When targeting a board revision, the active revision will be printed at CMake configure time, like this:

```
-- Board: plank, Revision: 1.5.0
```

2.4.10 Run an Application

An application image can be run on a real board or emulated hardware.

Running on a Board

Most boards supported by Zephyr let you flash a compiled binary using `west flash` to copy the binary to the board and run it. Follow these instructions to flash and run an application on real hardware:

1. Build your application, as described in *Building an Application*.
2. Make sure your board is attached to your host computer. Usually, you'll do this via USB.
3. Run this console command from the build directory, `<app>/build`, to flash the compiled Zephyr image and run it on your board:

```
west flash
```

The Zephyr build system integrates with the board support files to use hardware-specific tools to flash the Zephyr binary to your hardware, then run it.

Each time you run the flash command, your application is rebuilt and flashed again.

In cases where board support is incomplete, flashing via the Zephyr build system may not be supported. If you receive an error message about flash support being unavailable, consult your board's documentation for additional information on how to flash your board.

> **Note**
>
> When developing on Linux, it's common to need to install board-specific udev rules to enable USB device access to your board as a non-root user. If flashing fails, consult your board's documentation to see if this is necessary.

Running in an Emulator

Zephyr has built-in emulator support for QEMU. It allows you to run and test an application virtually, before (or in lieu of) loading and running it on actual target hardware.

Check out *Beyond the Getting Started Guide* for additional steps needed on Windows.

Follow these instructions to run an application via QEMU:

1. Build your application for one of the QEMU boards, as described in *Building an Application*.

   For example, you could set `BOARD` to:

   - `qemu_x86` to emulate running on an x86-based board
   - `qemu_cortex_m3` to emulate running on an ARM Cortex M3-based board

2. Run one of these console commands from the build directory, `<app>/build`, to run the Zephyr binary in QEMU:

   ```
   west build -t run
   ```

   or

   ```
   ninja run
   ```

3. Press `Ctrl A, X` to stop the application from running in QEMU.

   The application stops running and the terminal console prompt redisplays.

Each time you execute the run command, your application is rebuilt and run again.

> **Note**
>
> If the *Zephyr SDK* is installed, the `run` target will use the SDK's QEMU binary by default. To use another version of QEMU, *set the environment variable* `QEMU_BIN_PATH` to the path of the QEMU binary you want to use instead.

> **Note**
>
> You can choose a specific emulator by appending `_<emulator>` to your target name, for example `west build -t run_qemu` or `ninja run_qemu` for QEMU.

2.4.11 Custom Board, Devicetree and SOC Definitions

In cases where the board or platform you are developing for is not yet supported by Zephyr, you can add board, Devicetree and SOC definitions to your application without having to add them to the Zephyr tree.

The structure needed to support out-of-tree board and SOC development is similar to how boards and SOCs are maintained in the Zephyr tree. By using this structure, it will be much easier to upstream your platform related work into the Zephyr tree after your initial development is done.

Add the custom board to your application or a dedicated repository using the following structure:

```
boards/
soc/
CMakeLists.txt
prj.conf
README.rst
src/
```

where the `boards` directory hosts the board you are building for:

```
.
├── boards
│   └── vendor
│       └── my_custom_board
│           ├── doc
│           │   └── img
│           └── support
└── src
```

and the `soc` directory hosts any SOC code. You can also have boards that are supported by a SOC that is available in the Zephyr tree.

Boards

Use the vendor name as the folder name (which must match the vendor prefix in dts/bindings/vendor-prefixes.txt if submitting upstream to Zephyr, or be `others` if it is not a vendor board) under `boards` for `my_custom_board`.

Documentation (under `doc/`) and support files (under `support/`) are optional, but will be needed when submitting to Zephyr.

The contents of `my_custom_board` should follow the same guidelines for any Zephyr board, and provide the following files:

```
board.yml
my_custom_board_defconfig
my_custom_board.dts
my_custom_board.yaml
board.cmake
board.h
CMakeLists.txt
doc/
Kconfig.my_custom_board
Kconfig.defconfig
support/
```

Once the board structure is in place, you can build your application targeting this board by specifying the location of your custom board information with the `-DBOARD_ROOT` parameter to the CMake build system:

Using west:

```
west build -b <board name> -- -DBOARD_ROOT=<path to boards>
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DBOARD_ROOT=<path to boards> .
ninja -Cbuild
```

This will use your custom board configuration and will generate the Zephyr binary into your application directory.

You can also define the `BOARD_ROOT` variable in the application `CMakeLists.txt` file. Make sure to do so **before** pulling in the Zephyr boilerplate with `find_package(Zephyr ...)`.

> **Note**
>
> When specifying `BOARD_ROOT` in a CMakeLists.txt, then an absolute path must be provided, for example `list(APPEND BOARD_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/<extra-board-root>)`. When using `-DBOARD_ROOT=<board-root>` both absolute and relative paths can be used. Relative paths are treated relatively to the application directory.

> **Note**
>
> When using sysbuild, then `BOARD_ROOT` must defined in a module or in the sysbuild `CMakeLists.txt` file, see *Configuring sysbuild internal state* for details.

SOC Definitions

Similar to board support, the structure is similar to how SOCs are maintained in the Zephyr tree, for example:

```
soc
└── st
    └── stm32
        ├── common
        └── stm32l0x
```

The file soc/Kconfig will create the top-level `SoC/CPU/Configuration Selection` menu in Kconfig.

Out of tree SoC definitions can be added to this menu using the `SOC_ROOT` CMake variable. This variable contains a semicolon-separated list of directories which contain SoC support files.

Following the structure above, the following files can be added to load more SoCs into the menu.

```
soc
└── st
    └── stm32
        └── stm32l0x
            ├── Kconfig
            ├── Kconfig.soc
            └── Kconfig.defconfig
```

The Kconfig files above may describe the SoC or load additional SoC Kconfig files.

An example of loading `stm32l0` specific Kconfig files in this structure:

```
soc
└── st
    └── stm32
        ├── Kconfig.soc
        └── stm32l0x
            └── Kconfig.soc
```

can be done with the following content in `st/stm32/Kconfig.soc`:

```
rsource "*/Kconfig.soc"
```

Once the SOC structure is in place, you can build your application targeting this platform by specifying the location of your custom platform information with the `-DSOC_ROOT` parameter to the CMake build system:

Using west:

```
west build -b <board name> -- -DSOC_ROOT=<path to soc> -DBOARD_ROOT=<path to␣
↪boards>
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DSOC_ROOT=<path to soc> -DBOARD_ROOT=
↪<path to boards> .
ninja -Cbuild
```

This will use your custom platform configurations and will generate the Zephyr binary into your application directory.

See *Build settings* for information on setting SOC_ROOT in a module's `zephyr/module.yml` file.

Or you can define the `SOC_ROOT` variable in the application `CMakeLists.txt` file. Make sure to do so **before** pulling in the Zephyr boilerplate with `find_package(Zephyr ...)`.

> **Note**
>
> When specifying `SOC_ROOT` in a CMakeLists.txt, then an absolute path must be provided, for example `list(APPEND SOC_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/<extra-soc-root>)`. When using `-DSOC_ROOT=<soc-root>` both absolute and relative paths can be used. Relative paths are treated relatively to the application directory.

Devicetree Definitions

Devicetree directory trees are found in `APPLICATION_SOURCE_DIR`, `BOARD_DIR`, and `ZEPHYR_BASE`, but additional trees, or DTS_ROOTs, can be added by creating this directory tree:

```
include/
dts/common/
dts/arm/
dts/
dts/bindings/
```

Where 'arm' is changed to the appropriate architecture. Each directory is optional. The binding directory contains bindings and the other directories contain files that can be included from DT sources.

Once the directory structure is in place, you can use it by specifying its location through the `DTS_ROOT` CMake Cache variable:

Using west:

```
west build -b <board name> -- -DDTS_ROOT=<path to dts root>
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DDTS_ROOT=<path to dts root> .
ninja -Cbuild
```

You can also define the variable in the application `CMakeLists.txt` file. Make sure to do so **before** pulling in the Zephyr boilerplate with `find_package(Zephyr ...)`.

> **Note**
>
> When specifying `DTS_ROOT` in a CMakeLists.txt, then an absolute path must be provided, for example `list(APPEND DTS_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/<extra-dts-root>`. When using `-DDTS_ROOT=<dts-root>` both absolute and relative paths can be used. Relative paths are treated relatively to the application directory.

Devicetree source are passed through the C preprocessor, so you can include files that can be located in a `DTS_ROOT` directory. By convention devicetree include files have a `.dtsi` extension.

You can also use the preprocessor to control the content of a devicetree file, by specifying directives through the `DTS_EXTRA_CPPFLAGS` CMake Cache variable:

Using west:

```
west build -b <board name> -- -DDTS_EXTRA_CPPFLAGS=-DTEST_ENABLE_FEATURE
```

Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=<board name> -DDTS_EXTRA_CPPFLAGS=-
↪DTEST_ENABLE_FEATURE .
ninja -Cbuild
```

2.5 Debugging

2.5.1 Application Debugging

This section is a quick hands-on reference to start debugging your application with QEMU. Most content in this section is already covered in QEMU and GNU_Debugger reference manuals.

In this quick reference, you'll find shortcuts, specific environmental variables, and parameters that can help you to quickly set up your debugging environment.

The simplest way to debug an application running in QEMU is using the GNU Debugger and setting a local GDB server in your development system through QEMU.

You will need an ELF (Executable and Linkable Format) binary image for debugging purposes. The build system generates the image in the build directory. By default, the kernel binary name is `zephyr.elf`. The name can be changed using `CONFIG_KERNEL_BIN_NAME`.

GDB server

We will use the standard 1234 TCP port to open a GDB (GNU Debugger) server instance. This port number can be changed for a port that best suits the development environment. There are multiple ways to do this. Each way starts a QEMU instance with the processor halted at startup and with a GDB server instance listening for a connection.

Running QEMU directly You can run QEMU to listen for a "gdb connection" before it starts executing any code to debug it.

```
qemu -s -S <image>
```

will setup Qemu to listen on port 1234 and wait for a GDB connection to it.

The options used above have the following meaning:

- `-S` Do not start CPU at startup; rather, you must type 'c' in the monitor.
- `-s` Shorthand for `-gdb tcp::1234`: open a GDB server on TCP port 1234.

Running QEMU via `ninja` Run the following inside the build directory of an application:

```
ninja debugserver
```

QEMU will write the console output to the path specified in **`${QEMU_PIPE}`** via CMake, typically `qemu-fifo` within the build directory. You may monitor this file during the run with **`tail -f qemu-fifo`**.

Running QEMU via `west` Run the following from your project root:

```
west build -t debugserver_qemu
```

QEMU will write the console output to the terminal from which you invoked **`west`**.

Configuring the `gdbserver` listening device The Kconfig option `CONFIG_QEMU_GDBSERVER_LISTEN_DEV` controls the listening device, which can be a TCP port number or a path to a character device. GDB releases 9.0 and newer also support Unix domain sockets.

If the option is unset, then the QEMU invocation will lack a `-s` or a `-gdb` parameter. You can then use the `QEMU_EXTRA_FLAGS` shell environment variable to pass in your own listen device configuration.

GDB client

Connect to the server by running **gdb** and giving these commands:

```
$ path/to/gdb path/to/zephyr.elf
(gdb) target remote localhost:1234
(gdb) dir ZEPHYR_BASE
```

> **Note**
>
> Substitute the correct *ZEPHYR_BASE* for your system.

You can use a local GDB configuration `.gdbinit` to initialize your GDB instance on every run. Your home directory is a typical location for `.gdbinit`, but you can configure GDB to load from other locations, including the directory from which you invoked **gdb**. This example file performs the same configuration as above:

```
target remote localhost:1234
dir ZEPHYR_BASE
```

Alternate interfaces GDB provides a curses-based interface that runs in the terminal. Pass the `--tui` option when invoking **gdb** or give the `tui enable` command within **gdb**.

> **Note**
>
> The GDB version on your development system might not support the `--tui` option. Please make sure you use the GDB binary from the SDK which corresponds to the toolchain that has been used to build the binary.

Finally, the command below connects to the GDB server using the DDD (Data Display Debugger), a graphical frontend for GDB. The following command loads the symbol table from the ELF binary file, in this instance, `zephyr.elf`.

```
ddd --gdb --debugger "gdb zephyr.elf"
```

Both commands execute **gdb**. The command name might change depending on the toolchain you are using and your cross-development tools.

ddd may not be installed in your development system by default. Follow your system instructions to install it. For example, use **`sudo apt-get install ddd`** on an Ubuntu system.

Debugging

As configured above, when you connect the GDB client, the application will be stopped at system startup. You may set breakpoints, step through code, etc. as when running the application directly within **gdb**.

> **Note**
>
> **gdb** will not print the system console output as the application runs, unlike when you run a native application in GDB directly. If you just **`continue`** after connecting the client, the application will run, but nothing will appear to happen. Check the console output as described above.

2.5.2 Debug with Eclipse

Overview

CMake supports generating a project description file that can be imported into the Eclipse Integrated Development Environment (IDE) and used for graphical debugging.

The GNU MCU Eclipse plug-ins provide a mechanism to debug ARM projects in Eclipse with pyOCD, Segger J-Link, and OpenOCD debugging tools.

The following tutorial demonstrates how to debug a Zephyr application in Eclipse with pyOCD in Windows. It assumes you have already installed the GCC ARM Embedded toolchain and pyOCD.

Set Up the Eclipse Development Environment

1. Download and install Eclipse IDE for C/C++ Developers.
2. In Eclipse, install the GNU MCU Eclipse plug-ins by opening the menu `Window->Eclipse Marketplace...`, searching for `GNU MCU Eclipse`, and clicking `Install` on the matching result.
3. Configure the path to the pyOCD GDB server by opening the menu `Window->Preferences`, navigating to `MCU`, and setting the `Global pyOCD Path`.

Generate and Import an Eclipse Project

1. Set up a GNU Arm Embedded toolchain as described in *GNU Arm Embedded*.

2. Navigate to a folder outside of the Zephyr tree to build your application.

```
# On Windows
cd %userprofile%
```

> **Note**
>
> If the build directory is a subdirectory of the source directory, as is usually done in Zephyr, CMake will warn:
>
> "The build directory is a subdirectory of the source directory.
>
> This is not supported well by Eclipse. It is strongly recommended to use a build directory which is a sibling of the source directory."

3. Configure your application with CMake and build it with ninja. Note the different CMake generator specified by the `-G"Eclipse CDT4 - Ninja"` argument. This will generate an Eclipse project description file, `.project`, in addition to the usual ninja build files.

   Using west:

```
west build -b frdm_k64f samples/synchronization -- -G"Eclipse CDT4 - Ninja"
```

   Using CMake and ninja:

```
cmake -Bbuild -GNinja -DBOARD=frdm_k64f -G"Eclipse CDT4 - Ninja"␣
↪samples/synchronization
ninja -Cbuild
```

4. In Eclipse, import your generated project by opening the menu `File->Import...` and selecting the option `Existing Projects into Workspace`. Browse to your application build directory in the choice, `Select root directory:`. Check the box for your project in the list of projects found and click the `Finish` button.

Create a Debugger Configuration

1. Open the menu `Run->Debug Configurations...`.
2. Select `GDB PyOCD Debugging`, click the `New` button, and configure the following options:
   - In the Main tab:
     - Project: `my_zephyr_app@build`
     - C/C++ Application: `zephyr/zephyr.elf`
   - In the Debugger tab:
     - pyOCD Setup
       - Executable path: `$pyocd_path\$pyocd_executable`
       - Uncheck "Allocate console for semihosting"
     - Board Setup
       - Bus speed: 8000000 Hz
       - Uncheck "Enable semihosting"
     - GDB Client Setup
       - Executable path example (use your `GNUARMEMB_TOOLCHAIN_PATH`): `C:\gcc-arm-none-eabi-6_2017-q2-update\bin\arm-none-eabi-gdb.exe`
   - In the SVD Path tab:

– File path: `<workspace top>\modules\hal\nxp\mcux\devices\MK64F12\MK64F12.xml`

> **Note**
>
> This is optional. It provides the SoC's memory-mapped register addresses and bit-fields to the debugger.

3. Click the `Debug` button to start debugging.

RTOS Awareness

Support for Zephyr RTOS awareness is implemented in pyOCD v0.11.0 and later. It is compatible with GDB PyOCD Debugging in Eclipse, but you must enable CONFIG_DEBUG_THREAD_INFO=y in your application.

2.5.3 Debugging I2C communication

There is a possibility to log all or some of the I2C transactions done by the application. This feature is enabled by the Kconfig option `CONFIG_I2C_DUMP_MESSAGES`, but it uses the `LOG_DBG` function to print the contents so the `CONFIG_I2C_LOG_LEVEL_DBG` option must also be enabled.

The sample output of the dump looks like this:

```
D: I2C msg: io_i2c_ctrl7_port0, addr=50
D:    W      len=01: 00
D:    R Sr P len=08:
D: contents:
D: 43 42 41 00 00 00 00 00 |CBA.....
```

The first line indicates the I2C controller and the target address of the transaction. In above example, the I2C controller is named `io_i2c_ctrl7_port0` and the target device address is `0x50`

> **Note**
>
> the address, length and contents values are in hexadecimal, but lack the `0x` prefix

Next lines contain messages, both sent and received. The contents of write messages is always shown, while the content of read messages is controlled by a parameter to the function `i2c_dump_msgs_rw`. This function is available for use by user, but is also called internally by `i2c_transfer` API function with read content dump enabled. Before the length parameter, the header of the message is printed using abbreviations:

- W - write message
- R - read message
- Sr - restart bit
- P - stop bit

The above example shows one write message with byte `0x00` representing the address of register to read from the I2C target. After that the log shows the length of received message and following that, the bytes read from the target `43 42 41 00 00 00 00 00`. The content dump consist of both the hex and ASCII representation.

Filtering the I2C communication dump

By default, all I2C communication is logged between all I2C controllers and I2C targets. It may litter the log with unrelated devices and make it difficult to effectively debug the communication with a device of interest.

Enable the Kconfig option `CONFIG_I2C_DUMP_MESSAGES_ALLOWLIST` to create an allowlist of I2C targets to log. The allowlist of devices is configured using the devicetree, for example:

```
/ {
    i2c {
        display0: some-display@a {
            ...
        };
        sensor3: some-sensor@b {
            ...
        };
    };

    i2c-dump-allowlist {
        compatible = "zephyr,i2c-dump-allowlist";
        devices = <&display0>, <&sensor3>;
    };
};
```

The filters nodes are identified by the compatible string with `zephyr,i2c-dump-allowlist` value. The devices are selected using the `devices` property with phandles to the devices on the I2C bus.

In the above example, the communication with device `display0` and `sensor3` will be displayed in the log.

2.6 API Status and Guidelines

2.6.1 API Overview

The table lists Zephyr's APIs and information about them, including their current *stability level*. More details about API changes between major releases are available in the zephyr_release_notes.

The version column uses semantic version, and has the following expectations:

- Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.
  - If minor version is up to one (0.1.z), API is considered *experimental*.
  - If minor version is larger than one (0.y.z | y > 1), API is considered *unstable*.
- Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.
  - APIs with major versions equal or larger than one (x.y.z | x >= 1) are considered *stable*.
  - All existing stable APIs in Zephyr will be start with version 1.0.0.
- Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
- Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.

- Major version X (x.Y.z | x > 0) MUST be incremented if a compatibility breaking change was made to the API.

Note

Version for existing APIs are initially set based on the current state of the APIs:

- 0.1.0 denotes an *experimental* API
- 0.8.0 denote an *unstable* API,
- and finally 1.0.0 indicates a *stable* APIs.

Changes to APIs in the future will require adapting the version following the guidelines above.

API	Version	Available in Zephyr Since
Device Drivers		
1-Wire	0.8.0	v3.2.0
1-Wire Driver Backend API		
1-Wire Sensor API		
1-Wire data link layer		
1-Wire network layer		
ADC	1.0.0	v1.0.0
ADC Driver Backend API		
ADS126X ADC driver API		
Device-specific ADC API extensions		
ADS131M02		
ADS1x4s0x		
Current Sense Amplifier		
Current Sense Shunt		
Infineon AutAnalog SAR ADC	1.0.0	v4.5.0
LMP90xxx		
Voltage Divider		
WisBlock ADC		
Emulated ADC		
Renesas RA ADC_B channel selectors		
Audio		
Audio Codec Interface	0.3.0	v1.13.0
Audio Codec Driver Backend API		
Device-specific audio codec API extensions		
CS35L5X		
Digital Microphone Interface	0.2.0	v1.13.0
Digital Microphone Driver Backend API		
MIDI2 Universal MIDI Packet definitions	0.1.0	v4.1.0
MIDI commands		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Message types		
System common and System Real Time message status		
UMP Stream Responder	0.1.0	v4.3.0
UMP Stream specific fields		
UMP Stream Function Block discovery message filter bits		
UMP Stream endpoint discovery message filter bits		
UMP Stream format		
UMP Stream status		
Universal MIDI Packet (UMP) Format and MIDI 2.0 Protocol		
Auxiliary (Text) Display	0.8.0	v3.4.0
Auxiliary Display Driver Backend API		
BBRAM	1.0.0	v2.7.0
BBRAM Driver Backend API		
BBRAM emulator backend API		
BCM283x interrupt controllers		
Battery Charger	1.0.0	v3.5.0
Charger Driver Backend API		
Device-specific Charger API extensions		
nPM10xx Charger interface		
Biometrics	0.1.0	v4.4.0
Biometrics Driver Backend API		
Biometrics emulator test helpers		
Buzzer	0.1.0	v4.5.0
Buzzer Driver Backend API		
CAN		
CAN Controller	1.2.0	v1.12.0
CAN Controller Driver Backend API		
CAN Transceiver	0.1.0	v3.1.0
CAN Transceiver Driver Backend API		
Devicetree CAN API		
Fake CAN controller		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
CRC driver APIs	0.1.0	v4.3.0
CRC Driver Backend API		
Cellular	0.8.0	v3.6.0
Cellular Driver Backend API		
Cellular modem helpers		
Cellular modem driver definition macros		
Device-specific Cellular API extensions		
HL7800		
HL78xx		
SIMCom SIM7080		
ST87Mxx		
GNSS configuration		
ST87Mxx Constants		
WIFI scanning configuration		
Clock Control	1.0.0	v1.0.0
Clock Control Driver Backend API		
Device-specific Clock Control API extensions		
ARM CMSDK		
Ambiq		
Analog Devices MAX32		
Atmel SAM PMC		
Bouffalo Lab		
Core Devices SF32LB		
Espressif ESP32		
GigaDevice GD32		
Infineon CAT1		
Intel ADSP		
LiteX		
Microchip		
Microchip PIC32CMJH		
Microchip PIC32CMPL		
Microchip PIC32CZCA		
Microchip SAM D5x/E5x		
Microchip SAM PMC		
Microchip XEC		
NXP		
NXP LPC11U6x		
NXP clock sources		
Nordic nRF		
Nuvoton NuMaker		
Nuvoton NuMicro		
Raspberry Pi Pico		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Realtek Ameba		
Realtek BEE		
Realtek RTS5912		
Renesas R-Car		
Renesas R-Car CPG/MSSR		
Renesas RA CGC		
Renesas RA BSP Clock Clkout Divider Constants		
Renesas RA BSP Clock Source Constants		
Renesas RA Clock Divider Generators		
Renesas RX CGC		
Renesas RZ CGC		
Renesas SmartBond		
STMicroelectronics STM32		
Silicon Labs		
Silicon Labs SiWx91x		
TI		
Texas Instruments MSPM0		
Texas Instruments TISCI		
Devicetree Clocks API		
Aesc Silicon clock domain identifiers		
FocalTech FT9001 clock identifiers		
Microchip SmartFusion2 clock identifiers		
NXP i.MX952 Clock Sources		
Peripheral Clocks (41-163)		
Nuvoton M48x Clock Controller Devicetree helpers		
Nuvoton NuMaker controller Devicetree helpers		
Nuvoton NuMaker controller Devicetree helpers		
Silicon Labs Series 2 and 3 Devicetree helpers		
Silicon Labs SiWx91x clock identifiers		
Clock Monitor	0.1.0	v4.5.0
CLOCK_MONITOR Driver Backend API		
Comparator	0.8.0	v4.0.0
Comparator Driver Backend API		
Fake comparator		
Coredump pseudo-device	0.1.0	v3.2.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Coredump pseudo-device Driver Backend API		
Counter	1.1.0	v1.14.0
Counter Driver Backend API		
Counter capture APIs		
Device-specific Counter API extensions		
STM32 counter capture		
Devicetree Counter Captures API		
Emulated counter		
DAC	1.1.0	v2.3.0
DAC Driver Backend API		
Device-specific DAC API extensions		
DAC161S997		
DAI	0.9.0	v3.1.0
DAI Driver Backend API		
DALI	0.1.0	v4.4.0
DALI Driver Backend API		
DAP Link	0.1.0	v4.4.0
DMA	1.1.0	v1.5.0
DMA (SF32LB specifics)		
DMA Driver Backend API		
Devicetree DMA API		
Realtek Bee DMA Macros		
Xilinx ADMA specifics		
Disk Access	1.1.0	v1.6.0
Device-specific Disk Access API extensions		
STM32 SDMMC		
Display	0.10.0	v1.14.0
AC057TC1 display controller	0.1.0	v4.4.0
Device-specific Display API extensions		
SSD16xx		
Devicetree Display API		
Display Driver Backend API		
Color dithering		
Display panel devicetree bindings		
ED2208-GCA (EL073TF1) display controller	0.1.0	v4.5.0
MIPI Display interface		
MIPI-DBI	1.0.0	v3.6.0
MIPI-DBI Driver Backend API		
MIPI-DSI	1.0.0	v3.1.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
MIPI-DSI Driver Backend API		
Monochrome Character Framebuffer	0.10.0	v1.14.0
EDAC	0.8.0	v2.5.0
Device-specific EDAC API extensions		
NXP ERM EDAC		
Synopsys DDR controller EDAC		
EEPROM	1.0.0	v2.1.0
EEPROM Driver Backend API		
Fake EEPROM		
ESPI	1.0.0	v2.0.0
eSPI Driver Backend API		
eSPI Emulation Interface		
eSPI SAF	0.8.0	v2.6.0
eSPI SAF Driver Backend API		
Entropy	1.0.0	v1.10.0
Entropy Driver Backend API		
External Cache Controller	1.0.0	v2.6.0
FPGA	0.8.0	v2.7.0
FPGA Driver Backend API		
Flash	1.0.0	v1.2.0
Extended Operations		
Andes QSPI-NOR XIP		
IT51XXX		
Microchip	0.1.0	v4.5.0
Microchip NVMCTRL G1		
NPCX		
Realtek RTS5912		
Renesas RA		
ST M95P32		
STM32		
Flash Driver Backend API		
Flash Internal		
Fuel Gauge	0.9.0	v3.3.0
Devicetree battery helpers		
Fuel Gauge Driver Backend API		
Fuel gauge backend emulator APIs		
GNSS	0.8.0	v3.6.0
GNSS Driver Backend API		
GPIO	1.0.0	v1.0.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Device-specific GPIO API extensions		
Ambiq		
Bouffalolab BL61x GPIO FIFO / Wire Out	0.1.0	v4.4.0
MAX32-specific GPIO Flags		
NCT38XX		
NXP MCUX LPC		
NXP PCA SERIES		
NXP PCAL64XXA		
STM32 GPIO specific flags		
SX1509B		
nPM10xx-specific GPIO Flags		
nPM13xx-specific GPIO Flags		
nPM2100-specific GPIO Flags		
nPM6001-specific GPIO Flags		
nRF-specific GPIO Flags		
Devicetree GPIO API		
GPIO pin headers		
ArduCam FFC 40-pin camera connector.		
Arducam DVP 20-pin connector		
Arduino MKR header		
Arduino Mega header		
Arduino Nano header		
Arduino Uno (R3) header		
M.2 Key A serial memory connector		
Raspberry Pi CSI connector		
ST Morpho header		
ST Zio header		
STMod+ connector		
WisBlock IO slot connector		
WisBlock Sensor Slot connector		
Emulated GPIO		
GPIO Driver Backend API		
Haptics	0.4.0	v4.0.0
Device-specific Haptics API extensions		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
CS40L26/27		
CS40L5x		
DRV2605		
TM6605		
Haptics Driver Backend API		
Hardware Info	1.0.0	v1.14.0
Hardware Spinlock	0.8.0	v3.5.0
Hardware Spinlock Driver Backend API		
I2C	1.0.0	v1.0.0
I2C Driver Backend API		
I2C EEPROM Target	1.0.0	v1.13.0
I2C Emulation Interface		
I2S	1.0.0	v1.9.0
I2S Driver Backend API		
I3C	0.8.0	v3.2.0
I3C Address-related Helper Code		
I3C Common Command Codes		
I3C Devicetree related bits		
I3C HDR DDR API		
I3C In-Band Interrupts		
I3C Target Device API		
I3C Target Device Driver Backend API		
I3C Transfer API		
IPM	1.0.0	v1.0.0
IPM Driver Backend API		
Input	0.1.0	v3.4.0
Analog axis		
Device-specific Input API extensions		
CY8CMBR3xxx		
Nunchuk		
PAT912x		
PAW32xx		
PMW3610		
Renesas RA CTSU		
Renesas RX CTSU		
Input Event Definitions		
Keyboard Matrix		
Keymap utilities		
LVGL keypad keys		
Touchscreen Events	0.1.0	v3.7.0
Inter-VM Shared Memory		
LED	1.0.0	v1.12.0
Fake LED controller		
LED Driver Backend API		
LED Strip	0.8.0	v1.10.0
Device-specific LED Strip API extensions		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
TLC5971		
LED Strip Driver Backend API		
LIN		
LIN Controller	0.1.0	v4.5.0
LIN Transceiver	0.1.0	v4.5.0
LoRa	0.9.0	v2.2.0
LoRa Driver Backend API		
MBOX	0.8.0	v1.0.0
Devicetree MBOX API		
MBOX Driver Backend API		
MDIO	0.8.0	v2.7.0
MDIO Driver Backend API		
MEMC	0.1.0	v4.5.0
MEMC Driver Backend API		
MSPI	0.9.0	v3.7.0
MSPI Configure API		
Cadence MSPI interface		
MSPI Devicetree related macros		
MSPI Driver Backend API		
MSPI Emulation Interface		
MSPI Transfer API		
MSPI callback API		
Mspi_util		
MUX	0.1.0	v4.5.0
Devicetree MUX API		
Miscellaneous Devices		
Coresight STMESP		
Devmux		
FTDI FT8xx		
FT8xx co-processor		
FT8xx common functions		
FT8xx display list		
FT8xx memory map		
FT8xx reference API		
MAX2221X MISC		
NXP FlexIO		
Raspberry Pi Pico PIO		
Raspberry Pi Pico mailbox		
Renesas ELC		
TMC6460 MISC		
Multi-function Devices		
MFD AD559X interface		
MFD ADP5360 Interface	0.1.0	v4.5.0
MFD AXP192 interface		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
MFD BD8LB600FS interface		
MFD Infineon MXCRYPTO interface		
MFD Infineon MXCRYPTOLITE interface		
MFD M5PM1 Interface	0.1.0	v4.5.0
MFD MAX22017 interface		
MFD MAX2221X		
MFD NPM10XX Interface	0.3.0	v4.4.0
MFD NPM13XX Interface		
MFD NPM2100 Interface		
MFD PCA9422 Interface		
MFD TLA2528 Interface	0.1.0	v4.5.0
NVMEM	0.1.0	v4.3.0
NVMEM Devicetree Helpers		
OPAMP	0.1.0	v4.3.0
OPAMP Driver Backend API		
One Time Programmable Memory	0.1.0	v4.4.0
OTP Driver Backend API		
nPM10xx OTP interface	0.1.0	v4.5.0
PCIe		
PCIe Controller		
PCIe Controller Driver Backend API		
PCIe Endpoint Driver Backend API		
PCIe Host		
PCIe Capabilities		
PCIe Host MSI Interface		
PCIe Host PTM Interface		
PCIe Virtual Channel Host Interface		
PECI	1.0.0	v2.1.0
PECI Driver Backend API		
PMU		
PS/2	0.8.0	v2.1.0
PS/2 Driver Backend API		
PSI5	0.1.0	v4.2.0
PSI5 Driver Backend API		
PTP Clock	1.0.0	v1.13.0
PTP Clock Driver Backend API		
PWM	1.0.0	v1.0.0
Device-specific PWM API extensions		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
MAX2221X PWM		
MAX31790		
WisBlock PWM		
Devicetree PWMs API		
Fake PWM		
PWM Driver Backend API		
Pin Control	0.8.0	v3.0.0
Dynamic Pin Control		
Pin control		
Ambiq pin control helpers		
Ambiq Apollo2 pin control helpers		
Ambiq Apollo3 pin control helpers		
Ambiq Apollo4 pin control helpers		
Ambiq Apollo5 pin control helpers		
Espressif pin control helpers		
Espressif ESP32 pin control helpers		
Espressif ESP32C2 pin control helpers		
Espressif ESP32C3 pin control helpers		
Espressif ESP32C5 pin control helpers		
Espressif ESP32C6 pin control helpers		
Espressif ESP32C61 pin control helpers		
Espressif ESP32H2 pin control helpers		
Espressif ESP32P4 pin control helpers		
Espressif ESP32S2 pin control helpers		
Espressif ESP32S3 pin control helpers		
FocalTech pin control helpers		
FocalTech FT9001 pin control helpers		
FT9001 IOCTRL register offsets		
Nuvoton pin control helpers		
Nuvoton NuMaker M46x pin control helpers		
Renesas pin control helpers		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Renesas R-Car pin control helpers		
Renesas RA pin control helpers		
Renesas RA0 pin control helpers		
Renesas RX pin control helpers		
Renesas RZ/A pin control helpers		
Renesas RZ/A2M pin control helpers		
Renesas RZ/G pin control helpers		
Renesas RZ/N pin control helpers		
Renesas RZ/T pin control helpers		
Renesas RZ/V pin control helpers		
SiFli pin control helpers		
SiFli SF32LB52x pin control helpers		
Silicon Labs pin control helpers		
Silicon Labs SiWx91x pin control helpers		
Silicon Labs Six301 pin control helpers		
Silicon Labs xG21 pin control helpers		
Silicon Labs xG22 pin control helpers		
Silicon Labs xG23 pin control helpers		
Silicon Labs xG24 pin control helpers		
Silicon Labs xG26 pin control helpers		
Silicon Labs xG27 pin control helpers		
Silicon Labs xG28 pin control helpers		
Silicon Labs xG29 pin control helpers		
WCH CH32H41x pin control helpers		
Pulse IO	0.1.0	v4.4.0
QEMU fw_cfg	0.1.0	v4.4.0
RTC	0.1.0	v3.4.0
Device-specific RTC API extensions		
Fake RTC		
RTC Driver Backend API		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Regulator	0.1.0	v2.4.0
Devicetree Regulator API		
ADP5360 Devicetree helpers		
AXP192 Devicetree helpers		
MAX20335 Devicetree helpers		
MAX20362 Devicetree helpers		
MSPM0 VREF Devicetree helpers		
NXP VREF Devicetree helpers		
Raspberry Pi Pico regulator Devicetree helpers		
Silicon Labs DCDC Devicetree helpers		
Texas Instruments TPS6287x Devicetree helpers		
nPM10xx Devicetree helpers		
nPM1100 Devicetree helpers		
nPM13xx Devicetree helpers		
nPM2100 Devicetree helpers		
nPM6001 Devicetree helpers		
nRF5X Devicetree helpers		
Fake regulator		
Regulator Driver Backend API		
Regulator Parent Interface		
MAX20362 API		
PCA9420 Utilities.		
Reset Controller	0.2.0	v3.1.0
CH32 reset controller helpers		
Devicetree Reset Controller API		
FocalTech FT9001 reset controller helpers		
FocalTech Reset Control Module Registers		
FocalTech Reset Enable/Disable Definitions		
AHB3RSTCR Reset Control		
ARITHRSTCR Reset Control		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
EPORTRSTR Reset Control		
IPSRSTCR Reset Control		
MULTIRSTCR Reset Control		
SYSRSTCR Reset Control		
GigaDevice reset controller helpers		
GigaDevice GD32A50x reset controller helpers		
GigaDevice GD32E10x reset controller helpers		
GigaDevice GD32E50x reset controller helpers		
GigaDevice GD32F3x0 reset controller helpers		
GigaDevice GD32F403 reset controller helpers		
GigaDevice GD32F4xx reset controller helpers		
GigaDevice GD32L23x reset controller helpers		
GigaDevice GD32VF103 reset controller helpers		
Nuvoton M48X Reset controller Devicetree helpers		
Nuvoton NuMaker reset controller helpers		
Nuvoton NuMaker M031X reset controller helpers		
Nuvoton NuMaker M2L31X reset controller helpers		
Nuvoton NuMaker M333X reset controller helpers		
Nuvoton NuMaker M335X reset controller helpers		
Nuvoton NuMaker M46X reset controller helpers		
Nuvoton NuMaker M55M1X reset controller helpers		
Raspberry Pi RP2040 reset controller helpers		
Raspberry Pi RP2350 reset controller helpers		
Reset Controller Driver Backend API		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
STM32 reset controller helpers		
Retained memory	1.0.0	v3.4.0
Retained Memory Driver Backend API		
SCMI		
Protocols		
Base Protocol		
Clock Protocol		
NXP CPU Domain Protocol		
NXP System Power Protocol		
Pin Control Protocol		
Power Domain Protocol		
Reset domain management Protocol		
System Power Protocol		
Transport		
Shared Memory		
Utilities		
SDHC	0.8.0	v3.1.0
SDHC Driver Backend API		
SENT	0.1.0	v4.2.0
SENT Driver Backend API		
SMBus	0.8.0	v3.4.0
SMBus Driver Backend API		
SPI	1.1.0	v1.0.0
Devicetree SPI API		
SPI Driver Backend API		
SPI Emulation Interface		
SWDP	0.1.0	v3.7.0
Sensor	1.0.0	v1.2.0
Battery APIs		
Device-specific Sensor API extensions and Devicetree constants.		
1-Wire Sensor API		
AMS		
TCS3400		
Analog Devices		
ADLTC2990		
ADT7420		
ADXL313		
ADXL345		
ADXL355		
ADXL362		
ADXL367		
ADXL372		
MAX17055		
MAX30009		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
MAX30210		
MAX31790		
MAX32664C		
Angst+Pfister		
FCX-MLDX5		
Avago Technologies		
APDS9253		
APDS9960		
Avia Semiconductor		
HX711		
Bosch Sensortec		
BMM350		
BMP581		
Broadcom		
AFBR-S50		
EQDC MCUX		
Festo		
VEAA X-3		
Fintek		
F75303		
Gas Sensing Solutions		
ExplorIR-M		
Hangzhou Grow		
Grow R502A		
ITE		
IT51XXX tachometer		
IT8XXX2 tachometer		
IT8XXX2 voltage comparator		
Iclegend		
Iclegend S3KM1110		
Infineon		
TLE9104		
LTC4286		
MEMSIC		
MC3419		
MMC56X3		
Measurement Specialties		
HTU31D		
Melexis		
MLX90394		
Microchip		
MCP9600		
MTCH9010		
PAC194X		
NXP		
QDEC MCUX		
QDEC NXP S32		
National Semiconductor		
LM95234		
Nordic Semiconductor		
QDEC nRF		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
nPM10xx Sensor interface		
Nuvoton		
NPCX tachometer		
PNI		
RM3100		
Phosense		
XBR818		
PixArt		
PAJ7620		
PAT9136		
ROHM		
BD8LB600FS		
Realtek		
Realtek Bee QDEC		
S9706		
STMicroelectronics		
IIS2DLPC		
IIS2ICLX		
IIS3DWB		
ISM330DHCX		
ISM6HG256X		
LIS2DE12		
LIS2DH		
LIS2DS12		
LIS2DU12		
LIS2DUX12		
LIS2DW12		
LPS22HH		
LPS2xDF		
LSM6DSO		
LSM6DSO16IS		
LSM6DSV16X		
LSM6DSV320X		
LSM6DSV32X		
LSM6DSV80X		
LSM6DSVXXX		
LSM9DS1		
QDEC STM32		
STTS22H		
VL53L0X		
ScioSense		
ENS160		
Sensirion		
SCD4X		
SGP40		
SHT4X		
STCC4		
Sensor axis alignment		
TDK InvenSense		
ICM42686		
ICM42688		
ICM4268X		
ICM45686		
ICM566xx		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Texas Instruments		
BQ274XX		
FDC2X1X		
INA226		
INA230		
INA237		
INA2XX		
TMAG5273		
TMP114		
TMP11X		
TMP451		
Vishay		
VEML6031		
VEML6046		
VEML60XX		
Winsen		
MH-Z19B		
Wurth Elektronik		
WSEN-PADS-2511020213301		
WSEN-TIDS-2521020222501		
Zephyr		
Generic Sensor Emulator		
Sensor Driver Backend API		
Sensor emulator backend API		
Stepper		
Device-specific stepper API extensions		
DRV84XX Stepper Controller		
TMCM-3216 Stepper Controller		
Trinamic Stepper Controller	0.9.0	v4.0.0
Stepper Hardware Driver	0.9.0	v4.0.0
Stepper Hardware Driver Driver Backend API		
Stepper Motion Controller	0.9.0	v4.0.0
Stepper Motion Controller Driver Backend API		
System control (SYSCON)	0.8.0	v2.7.0
SYSCON Driver Backend API		
TEE	0.1.0	v3.7.0
TEE Driver Backend API		
Time-aware GPIO	0.1.0	v3.5.0
Time-aware GPIO Driver Backend API		
UART	2.0.0	v1.0.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Async UART API	1.0.0	v1.14.0
Interrupt-driven UART API		
Polling UART API		
UART Emulation Interface		
USB		
Battery Charging (BC1.2)		
BC1.2 backed emulator APIs		
USB Device Controller	0.1.1	v3.3.0
USB Host Controller	0.1.1	v3.3.0
USB Type-C		
USB Power Delivery		
USB Power Delivery Power Data Objects		
USB Type-C Port Controller API	0.1.1	v3.1.0
USB Type-C Port Controller Driver Backend API		
USB Type-C Power Path Controller		
USB Type-C Power Path Controller Driver Backend API		
USB-C VBUS API	0.1.0	v3.3.0
USB-C VBUS Driver Backend API		
VIRTIO	0.1.0	v4.2.0
VIRTIO Driver Backend API		
Virtqueue Interface		
Video Drivers	1.3.0	v2.1.0
Video Controls		
Video Driver Backend API		
Video interface bus types		
WUC	0.1.0	v4.4.0
Devicetree WUC Controller API		
WUC Driver Backend API		
Watchdog	1.0.0	v1.0.0
Watchdog Driver Backend API		
Device Model	1.1.0	v1.0.0
Device memory-mapped IO management		
Named MMIO region macros		
Single MMIO region macros		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Top-level MMIO region macros		
EM32 Clock Gating IDs		
Devicetree	1.2.0	v2.2.0
"For-each" macros		
Bus helpers		
Chosen nodes		
Dependency tracking		
Devicetree CAN API		
Devicetree Clocks API		
Aesc Silicon clock domain identifiers		
FocalTech FT9001 clock identifiers		
Microchip SmartFusion2 clock identifiers		
NXP i.MX952 Clock Sources		
Peripheral Clocks (41-163)		
Devicetree Counter Captures API		
Devicetree DMA API		
Devicetree Display API		
Devicetree Fixed Partition API		
Devicetree GPIO API		
GPIO pin headers		
ArduCam FFC 40-pin camera connector.		
Arducam DVP 20-pin connector		
Arduino MKR header		
Arduino Mega header		
Arduino Nano header		
Arduino Uno (R3) header		
M.2 Key A serial memory connector		
Raspberry Pi CSI connector		
ST Morpho header		
ST Zio header		
STMod+ connector		
WisBlock IO slot connector		
WisBlock Sensor Slot connector		
Devicetree HW spinlock API		
Devicetree IO Channels API		
Devicetree Interrupt CPU API		
Devicetree Interrupt Controller API		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Arm GIC interrupt controller		
Espressif ESP32 interrupt allocator		
Espressif ESP32-C2 interrupt allocator		
Espressif ESP32-C3 interrupt allocator		
Espressif ESP32-C5 interrupt allocator		
Espressif ESP32-C6 interrupt allocator		
Espressif ESP32-C61 interrupt allocator		
Espressif ESP32-H2 interrupt allocator		
Espressif ESP32-P4 interrupt allocator		
Espressif ESP32-S2 interrupt allocator		
Espressif ESP32-S3 interrupt allocator		
ITE IT51XXX interrupt controller		
ITE IT51XXX wake-up controller		
ITE IT8XXX2 interrupt controller		
ITE IT8XXX2 wake-up controller		
Infineon XMC4xxx interrupt controller		
Intel I/O APIC interrupt controller		
Microchip SmartFusion2 interrupt controller		
Microchip XEC ECIA interrupt controller		
OpenISA RV32M1 INTMUX		
Realtek RTS5817 interrupt sources		
TI VIM interrupt controller		
Devicetree MBOX API		
Devicetree MUX API		
Devicetree Mapped Partition API		
Devicetree PWMs API		
Devicetree Partition API		
Devicetree Port Endpoint API		
Devicetree Regulator API		
ADP5360 Devicetree helpers		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
AXP192 Devicetree helpers		
MAX20335 Devicetree helpers		
MAX20362 Devicetree helpers		
MSPM0 VREF Devicetree helpers		
NXP VREF Devicetree helpers		
Raspberry Pi Pico regulator Devicetree helpers		
Silicon Labs DCDC Devicetree helpers		
Texas Instruments TPS6287x Devicetree helpers		
nPM10xx Devicetree helpers		
nPM1100 Devicetree helpers		
nPM13xx Devicetree helpers		
nPM2100 Devicetree helpers		
nPM6001 Devicetree helpers		
nRF5X Devicetree helpers		
Devicetree Renesas RA ELC helpers		
Renesas RA2A1 ELC devicetree macros		
Renesas RA2L1 ELC devicetree macros		
Renesas RA4E1 ELC devicetree macros		
Renesas RA4E2 ELC devicetree macros		
Renesas RA4L1 ELC devicetree macros		
Renesas RA4M1 ELC devicetree macros		
Renesas RA4M2 ELC devicetree macros		
Renesas RA4M3 ELC devicetree macros		
Renesas RA4T1 ELC devicetree macros		
Renesas RA4W1 ELC devicetree macros		
Renesas RA6E1 ELC devicetree macros		
Renesas RA6E2 ELC devicetree macros		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Renesas RA6M1 ELC devicetree macros		
Renesas RA6M2 ELC devicetree macros		
Renesas RA6M3 ELC devicetree macros		
Renesas RA6M4 ELC devicetree macros		
Renesas RA6M5 ELC devicetree macros		
Renesas RA8D1 ELC devicetree macros		
Renesas RA8E1 ELC devicetree macros		
Renesas RA8M1 ELC devicetree macros		
Renesas RA8T1 ELC devicetree macros		
Devicetree Reset Controller API		
Devicetree SPI API		
Devicetree SRAM API		
Devicetree WUC Controller API		
Devicetree battery helpers		
Devicetree memory attributes		
Architecture-specific memory attributes		
ARM MPU memory attributes		
ARM64 memory attributes		
RISC-V memory attributes		
Xtensa memory attributes		
Generic memory attributes		
Software-specific memory attributes		
Software-defined memory attributes		
Existence checks		
Instance-based devicetree APIs		
Map helpers		
NVMEM Devicetree Helpers		
NXP i.MX952 Power Domains		
Node identifiers and helpers		
Pin control		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Ambiq pin control helpers		
Ambiq Apollo2 pin control helpers		
Ambiq Apollo3 pin control helpers		
Ambiq Apollo4 pin control helpers		
Ambiq Apollo5 pin control helpers		
Espressif pin control helpers		
Espressif ESP32 pin control helpers		
Espressif ESP32C2 pin control helpers		
Espressif ESP32C3 pin control helpers		
Espressif ESP32C5 pin control helpers		
Espressif ESP32C6 pin control helpers		
Espressif ESP32C61 pin control helpers		
Espressif ESP32H2 pin control helpers		
Espressif ESP32P4 pin control helpers		
Espressif ESP32S2 pin control helpers		
Espressif ESP32S3 pin control helpers		
FocalTech pin control helpers		
FocalTech FT9001 pin control helpers		
FT9001 IOCTRL register offsets		
Nuvoton pin control helpers		
Nuvoton NuMaker M46x pin control helpers		
Renesas pin control helpers		
Renesas R-Car pin control helpers		
Renesas RA pin control helpers		
Renesas RA0 pin control helpers		
Renesas RX pin control helpers		
Renesas RZ/A pin control helpers		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Renesas RZ/A2M pin control helpers		
Renesas RZ/G pin control helpers		
Renesas RZ/N pin control helpers		
Renesas RZ/T pin control helpers		
Renesas RZ/V pin control helpers		
SiFli pin control helpers		
SiFli SF32LB52x pin control helpers		
Silicon Labs pin control helpers		
Silicon Labs SiWx91x pin control helpers		
Silicon Labs Six301 pin control helpers		
Silicon Labs xG21 pin control helpers		
Silicon Labs xG22 pin control helpers		
Silicon Labs xG23 pin control helpers		
Silicon Labs xG24 pin control helpers		
Silicon Labs xG26 pin control helpers		
Silicon Labs xG27 pin control helpers		
Silicon Labs xG28 pin control helpers		
Silicon Labs xG29 pin control helpers		
WCH CH32H41x pin control helpers		
Property accessors		
Vendor and model name helpers		
interrupts property		
ranges property		
reg property		
Error numbers		
Internal and System API		
Architecture Interface		
Architecture thread APIs		
Architecture timing APIs		
Architecture-specific IRQ APIs		
Architecture-specific SMP APIs		
Architecture-specific Stack Walk APIs		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Architecture-specific Thread Local Storage APIs		
Architecture-specific cache controllers.		
Architecture-specific core dump APIs		
Architecture-specific gdbstub APIs		
Architecture-specific memory-mapping APIs		
Architecture-specific power management APIs		
Architecture-specific userspace APIs		
Instruction Memory APIs		
Miscellaneous architecture APIs		
Kernel Memory Management Internal APIs		
User Mode Internal APIs		
User mode and Syscall APIs		
Kernel APIs	1.0.0	v1.0.0
Async polling APIs		
Atomic Services APIs		
Barrier Services APIs	0.1.0	v3.4.0
CPU Idling APIs		
Condition Variables APIs		
Event APIs		
FIFO APIs		
FUTEX APIs		
Fatal error APIs		
Fatal error base types		
Floating Point APIs		
Heap APIs		
Interrupt Service Routine APIs		
Kernel Memory Management		
Demand Paging		
Backing Store APIs		
Demand Paging APIs		
Eviction Algorithm APIs		
LIFO APIs		
Mailbox APIs		
Memory Slab APIs		
Memory domain APIs		
Application memory domain APIs		
Message Queue APIs		
Mutex APIs		
Object Core APIs		
Object Core Statistics APIs		
Pipe APIs		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Queue APIs		
Semaphore APIs		
Spinlock APIs		
Stack APIs		
System Clock APIs		
System Timer Low-power Companion Interface		
Thread APIs		
Thread Stack APIs	1.0.0	v1.8.0
Timer APIs		
User Mode APIs		
User mode mutex APIs		
User mode semaphore APIs		
Version APIs		
Work Queue APIs		
Memory Management APIs		
Memory Attribute Interface		
Memory heaps based on memory attributes		
Operating System Services		
Asynchronous Notification APIs		
Authentication		
FIDO2	0.1.1	v4.5.0
Binary Descriptors		
Binary Descriptor Definition		
Binary Descriptor Reading		
CPU Frequency (CPUFreq)	0.8.0	v4.3.0
CPU Frequency Policy	0.8.0	v4.3.0
CPU Frequency pstate	0.8.0	v4.3.0
CPU load monitor		
Cache Interface	1.0.0	v3.3.0
Checksum		
CRC	1.0.0	v1.8.0
Cleanup Helper Interface		
Connectivity		
Bluetooth APIs		
Advanced Audio Distribution Profile (A2DP)		
Alert Notification Service (ANS)	0.1.0	v4.4.0
Assigned Numbers.	1.0.0	v1.0.0
Bluetooth Mesh Assigned Numbers		
Core Specification Assigned Numbers		
Generic Audio Assigned Numbers		
Attribute Protocol (ATT)	1.1.0	v1.1.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Audio Input Control Service (AICS)	0.8.0	v2.6.0
Audio Stream Control Service (ASCS)	0.1.0	v4.5.0
Audio Video Remote Control Profile (AVRCP)		
Audio Video Remote Control Cover Art Profile		
Basic Audio Profile (BAP) LC3 Presets	0.8.0	v3.0.0
Basic Imaging Profile (BIP)		
Battery Service (BAS)		
BlueNRG HCI driver extended API		
Bluetooth Audio	0.8.0	v3.1.0
Assigned numbers to string API		
Codec capability parsing APIs		
Codec config parsing APIs		
Bluetooth Basic Audio Profile	0.8.0	v3.0.0
BAP Broadcast APIs		
BAP Broadcast Sink APIs		
BAP Broadcast Source APIs		
BAP Broadcast Sink APIs		
BAP Broadcast Source APIs		
BAP Unicast Client APIs		
BAP Unicast Server APIs		
Bluetooth Content Control Identifier	0.8.0	v3.7.0
Bluetooth Controller		
Bluetooth Gaming Audio Profile	0.8.0	v3.5.0
Bluetooth HCI	0.3.0	v3.7.0
Bluetooth H:4 UART driver vendor extension interface	0.1.0	v4.5.0
Bluetooth HCI Driver Backend API		
Bluetooth HCI lockstep command helper	0.1.0	v4.5.0
Bluetooth HCI packet helpers	0.1.0	v4.5.0
Bluetooth HID		
Bluetooth HID Device		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Bluetooth HID Host		
Bluetooth LC3 codec	0.8.0	v3.0.0
Bluetooth Logical Link Control and Adaptation Protocol (L2CAP) for BR/EDR		
Bluetooth Mesh		
Access layer		
Bluetooth Mesh BLOB Transfer Client model API		
Bluetooth Mesh BLOB Transfer Server model API		
Bluetooth Mesh BLOB flash stream		
Bluetooth Mesh BLOB model API		
Bluetooth Mesh Device Firmware Update		
Bluetooth Mesh Device Firmware Update (DFU) metadata		
Firmware Update Server model		
Firmware Uppdate Client model		
Bluetooth Mesh On-Demand Private GATT Proxy Client		
Bluetooth Mesh On-Demand Private GATT Proxy Server		
Bluetooth Mesh Private Beacon Client		
Bluetooth Mesh Private Beacon Server		
Bluetooth Mesh SAR Configuration Client Model		
Bluetooth Mesh SAR Configuration Server Model		
Bluetooth Mesh Solicitation PDU RPL Client		
Bluetooth Mesh Solicitation PDU RPL Server		
Bridge Configuration Client Model		
Bridge Configuration Server Model		
Bridge Configuration common header		
Configuration Client Model		
Configuration Server Model		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Firmware Distribution models		
Firmware Distribution Server model		
Health Client Model		
Health Server Model		
Health faults		
Heartbeat		
Large Composition Data Client model		
Large Composition Data Server model		
Message		
Opcodes Aggregator Client model		
Opcodes Aggregator Server model		
Provisioning		
Proxy		
Remote Provisioning Client model		
Remote Provisioning models		
Remote provisioning server		
Runtime Configuration		
Application Configuration		
Subnet Configuration		
SAR Configuration common header		
Statistic		
Byteorder	1.0.0	v3.4.0
Call Control Profile (CCP)	0.1.0	v3.7.0
CCP Call Control Client APIs		
CCP Call Control Server APIs		
Channel Sounding (CS)		
Common Audio Profile (CAP)	0.8.0	v3.2.0
Connection management	1.0.1	v1.0.0
Coordinated Set Identification Profile (CSIP)	0.8.0	v3.0.0
Cryptography	1.0.0	v1.8.0
Current Time Service (CTS)		
Data buffers	1.0.0	v1.7.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Device Address	1.0.0	v1.0.0
Elapsed Time Service (ETS)		
Encrypted Advertising Data (EAD)		
Gaming Audio Profile (GMAP) LC3 Presets	0.8.0	v3.5.0
Generic Access Profile (GAP)	1.1.1	v1.0.0
Defines and Assigned Numbers		
Generic Attribute Profile (GATT)	1.0.1	v1.0.0
GATT Client APIs		
GATT Server APIs		
Generic Object Exchange Profile (GOEP)		
GOEP transport L2CAP		
GOEP transport RFCOMM		
HCI RAW channel	1.0.0	v2.3.0
Hands Free Profile (HFP)		
Hands Free Profile - Audio Gateway (HFP-AG)		
Hearing Access Service (HAS)	0.8.0	v3.1.0
Heart Rate Service (HRS)		
Immediate Alert Service (IAS)		
IrDA Object Exchange Protocol (OBEX)		
Isochronous channels (ISO)	0.8.0	v2.3.0
L2CAP	1.0.0	v1.0.0
L2CAP channel identifiers		
Media Control Client (MCC)	0.8.0	v3.0.0
Media Control Service (MCS)	0.8.0	v3.0.0
Media Proxy	0.8.0	v3.0.0
Message Access Profile (MAP)		
Messaging Client Equipment(MCE) OBEX Client		
Messaging Client Equipment(MCE) OBEX Server		
Messaging Server Equipment(MSE) OBEX Client		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Messaging Server Equipment(MSE) OBEX Server		
Microphone Control Profile (MICP)	0.8.0	v2.7.0
Object Transfer Service (OTS)	0.1.0	v2.4.0
Phone Book Access Profile (PBAP)		
Public Broadcast Profile (PBP)	0.8.0	v3.5.0
Published Audio Capabilities Service (PACS)	0.8.0	v3.0.0
RFCOMM	0.1.0	v1.6.0
Service Discovery Protocol (SDP)	0.1.0	v1.6.0
Telephone Bearer Service (TBS)	0.8.0	v3.0.0
Telephone and Media Audio Profile (TMAP)	0.8.0	v3.4.0
UUIDs	1.0.0	v1.0.0
Volume Control Profile (VCP)	0.8.0	v2.7.0
Volume Offset Control Service (VOCS)	0.8.0	v2.6.0
CAN ISO-TP Protocol	1.0.0	v2.2.0
IEEE 802.15.4 and Thread APIs	0.8.0	v1.0.0
IEEE 802.15.4 Drivers	0.8.0	v1.0.0
IEEE 802.15.4 L2	0.8.0	v1.0.0
IEEE 802.15.4 Net Management	0.8.0	v1.0.0
OpenThread stack	0.8.0	v1.11.0
LoRaWAN APIs	0.2.0	v2.5.0
MODBUS		
Modem APIs	1.0.0	v3.5.0
Modem CMUX	1.0.0	v3.5.0
Modem Chat	1.0.0	v3.5.0
Modem PPP	1.0.0	v3.5.0
Modem Pipe	1.0.0	v3.5.0
Modem Ubx	1.0.0	v3.7.0
Modem pipelink	1.0.0	v3.7.0
Networking	1.0.0	v1.0.0
Application network context	0.8.0	v1.0.0
BSD Sockets compatible API	1.0.0	v1.9.0
Socket options for IPv4		
Socket options for IPv6		
Socket options for TLS	0.8.0	v1.13.0
Socket options for packet socket		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
BSD socket service API	0.2.0	v3.6.0
COAP Library	0.8.0	v1.10.0
CoAP Manager Events	0.1.0	v3.6.0
CoAP OSCORE API	0.1.0	v4.5.0
CoAP TCP client API		
CoAP client API	0.1.0	v3.4.0
CoAP service API	0.1.0	v3.6.0
Connection Manager API	0.1.0	v2.0.0
Connection Manager Connectivity API	0.8.0	v3.4.0
Connection Manager Connectivity Bulk API	0.1.0	v3.4.0
Connection Manager Connectivity Implementation API	0.8.0	v3.4.0
DHCPv4	0.8.0	v1.7.0
DHCPv4 server	0.8.0	v3.6.0
DHCPv6	0.8.0	v3.5.0
DHCPv6 server	0.1.0	v4.5.0
DNS Resolve Library	0.9.0	v1.8.0
DNS Service Discovery	0.8.0	v2.5.0
Datagram PLPMTUD API	0.1.0	v4.5.0
Distributed Switch Architecture (DSA)	0.8.0	v4.2.0
Dummy L2/driver Support Functions	0.8.0	v1.14.0
Ethernet Bridging API	0.8.0	v2.7.0
Ethernet Library	0.8.0	v1.12.0
Ethernet PHY Interface	0.8.0	v2.7.0
Ethernet PHY Driver Backend API		
Ethernet Support Functions	0.8.0	v1.0.0
DSA tag protocol	0.8.0	v4.4.0
Ethernet MII Support Functions	0.8.0	v1.7.0
IEEE 802.3 management interface	0.8.0	v3.5.0
Ethernet bridge FDB	0.1.0	v4.4.0
FTP client library API	0.1.0	v4.4.0
HTTP HPACK	0.1.0	v3.7.0
HTTP client API	0.2.0	v2.1.0
HTTP request methods	0.8.0	v3.3.0
HTTP response status codes	0.8.0	v3.3.0
HTTP server API	0.1.0	v3.7.0
HTTP service API	0.1.0	v3.4.0
ICMP	0.8.0	v3.5.0
IGMP API	0.8.0	v2.6.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
IPv4 Network Address Translation	0.1.0	v4.5.0
IPv4/IPv6 primitives and helpers	1.0.0	v1.0.0
Latency Monitor		
Link Layer Discovery Protocol definitions and helpers	0.8.0	v1.13.0
LwM2M high-level API	0.8.0	v1.9.0
LwM2M path helper macros	0.8.0	v2.5.0
MCP Server API	0.1.0	v4.5.0
MQTT Client library	0.8.0	v1.14.0
MQTT-SN Client library	0.1.0	v3.3.0
Multicast Listener Discovery API	0.8.0	v1.8.0
Network Configuration Library	0.8.0	v1.8.0
Network Core Library	1.0.0	v1.0.0
Network Hostname Library	0.8.0	v1.10.0
Network Interface abstraction layer	1.1.0	v1.5.0
Network L2 Abstraction Layer	1.0.0	v1.5.0
Network Link Address Library	1.0.0	v1.0.0
Network MIDI 2.0	0.1.0	v4.3.0
User Datagram Protocol for Universal MIDI Packets		
Network Management	2.0.0	v1.7.0
Network Offloading Interface	0.8.0	v1.7.0
Network Packet Filter API	0.8.0	v3.0.0
Basic Filter Conditions	0.8.0	v3.0.0
Ethernet Filter Conditions	0.8.0	v3.0.0
Network Packet Library	0.8.0	v1.5.0
Network Statistics Library	0.8.0	v1.5.0
Network long timeout primitives and helpers	0.8.0	v1.14.0
Network namespace compatibility mode symbols	0.0.1	v4.4.0
Network packet capture	0.8.0	v2.6.0
Network time representation.	0.1.0	v3.5.0
OCPP library	0.1.0	v4.3.0

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Offloaded Net Devices	0.8.0	v3.4.0
PPP L2/driver Support Functions	0.8.0	v2.0.0
PTP support	0.1.0	v3.7.0
PTP time	0.8.0	v1.13.0
Prometheus API	0.1.0	v4.0.0
Promiscuous mode	0.8.0	v1.13.0
QUIC Library	0.1.0	v4.5.0
RTP (Real-time Transport Protocol)	0.1.0	
SNTP	0.8.0	v1.10.0
SNTP server	0.1.0	v4.5.0
SSH Client Library	0.1.0	v4.5.0
SSH Server Library	0.1.0	v4.5.0
SSH client/server common API	0.1.0	v4.5.0
SSH keygen API	0.1.0	v4.5.0
Socket NET_MGMT library	0.1.0	v2.0.0
SocketCAN library	0.8.0	v1.14.0
TFTP Client library	0.1.0	v2.3.0
TLS credentials management	0.8.0	v1.13.0
Trickle Algorithm Library	0.8.0	v1.7.0
Virtual Interface Library	0.8.0	v2.6.0
Virtual LAN definitions and helpers	0.8.0	v1.12.0
Virtual Network Interface Support Functions	0.8.0	v2.6.0
Websocket API	0.1.0	v1.12.0
Wi-Fi Management	0.8.0	v1.12.0
ESP32 Wi-Fi Mesh	0.1.0	v4.5.0
Wi-Fi Network Manager API	0.8.0	v3.5.0
Wi-Fi credentials library	0.1.0	v4.0.0
Wireguard VPN service	0.1.0	v4.4.0
Zperf API	0.8.0	v3.3.0
gPTP support	0.1.0	v1.13.0
USB		
USB Audio Class 2 device API	0.2.0	v3.6.0
USB BOS support	1.0.0	v1.13.0
USB DFU device update API	0.1.0	v4.1.0
USB HID class API	1.0.0	v1.11.0
HID class USB specific definitions		
USB HID common definitions	1.0.0	v1.11.0
Mouse and keyboard report descriptors		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
USB HID Item helpers		
USB Host Core API	0.1.0	v3.3.0
USB MIDI 2.0 Class device API	0.1.0	v4.1.0
USB Mass Storage Class device API	0.2.0	v3.4.0
USB Video Class (UVC) device API	0.1.1	v4.2.0
USB buffer macros and definitions	0.1.1	v4.0.0
USB device core API	0.2.0	v3.3.0
USB device core API	0.2.0	v3.7.0
USBD HID device API	0.4.0	v3.7.0
Console API		
Crypto	1.0.0	v1.7.0
Cipher		
Crypto Driver Backend API		
Hash		
Random Function APIs	1.0.0	v1.0.0
Debug		
Coredump APIs	1.0.0	v2.4.0
Coresight APIs		
Coresight Trace Deformatter		
STP Decoder API		
Symbol Table API		
Thread analyzer		
Device Management		
EC Host Command Interface	0.1.0	v2.4.0
Host Command Driver Backend API		
Simulator		
MCTP		
MCUmgr		
Callbacks		
Enumeration Management Callbacks		
File System Management Callbacks		
Image Management Callbacks		
OS Management Callbacks		
Settings Management Callbacks		
Handlers		
Management	1.0.0	v1.11.0
Enumeration Management		
Enumeration Management Callbacks		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
File System Management		
File System Management Callbacks		
Image Management		
Image Management Callbacks		
Image Management Client		
OS Management		
OS Management Callbacks		
OS Management Client		
Settings Management		
Settings Management Callbacks		
Shell Management		
Statistics Management		
Transport management	0.1.0	v4.5.0
Zephyr Basic Management		
SMP client		
Transport layers		
Bluetooth transport		
Dummy transport		
SMP transport		
Serial transport		
Shell transport		
UDP transport		
Error checking	1.0.0	v2.2.0
FOTA HTTP	0.1.0	v4.5.0
File System APIs	1.0.0	v1.5.0
File System Storage		
Flash Circular Buffer (FCB)	1.0.0	v1.11.0
Flash Circular Buffer Data Structures		
fcb API		
fcb non-API prototypes		
Non-volatile Storage (NVS)	1.0.0	v1.12.0
Non-volatile Storage APIs		
Non-volatile Storage Data Structures		
Settings	1.0.0	v1.12.0
Settings backend interface		
Settings name processing		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Settings subsystem runtime		
Zephyr Memory Storage (ZMS)	0.1.0	v4.0.0
ZMS API		
ZMS data structures		
Flash image API		
Heap Management		
Heap KASAN APIs		
Heap Listener APIs		
Low Level Heap Allocator	1.0.0	v2.3.0
Multi-Heap Wrapper		
Shared multi-heap interface		
IPC		
IPC service APIs	1.0.0	v2.7.0
IPC service RPMsg API		
IPC service backend		
IPC service static VRINGs API		
Icmsg IPC library API		
Icmsg multi-endpoint IPC library API		
Packed Buffer API		
RPMsg service APIs		
Instrumentation		
Iterable Sections APIs	1.0.0	v2.7.0
Linkable loadable extensions	0.1.0	v3.5.0
ELF constants and data types		
ELF inspection APIs		
ELF loader context		
Exported symbol definitions		
Logging	1.0.0	v1.13.0
Logger system		v1.13.0
Log links		
Log messages		
Log output		
Log output formatting flags.		
Logger backends		
Bluetooth log backend		
Intel ADSP HDA log backend		
Intel ADSP mtrace log backend		
Logger multidomain backend helpers		
MQTT log backend		
Network log backend		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Standard backend helpers		
Websocket log backend		
Logger control API		v1.13.0
Logger frontend		
STMESP frontend		
Logging API		
Trace and debug domain		
STMESP demultiplexer		
Memory Management		
Memory Banks Driver APIs		
Memory Blocks APIs		
Memory Management Driver APIs		
Word Granular Access Memory APIs		
Module based assert handling		
Multimedia		
Video Subsystem	0.2.0	v4.4.0
Video Buffers		
Video Capabilities		
Video Control IDs	0.0.1	v4.4.0
Video Controls		
Video Format		
Video Formats	0.0.1	v4.4.0
Video Frame Interval		
Video Stream		
Video Types	0.0.1	v4.4.0
Multimedia Pipeline	0.1.0	v4.5.0
Framework		
Bins		
Buffers		
Capability Structure		
Dispatches		
Elements		
Messages		
Objects		
Pad		
Parsers		
Pipelines		
Sinks		
Sources		
Threads		
Transforms		
Value Container		
Workqueues		
Player		
Plugins		
Base Elements		
Application sinks		
Application sources		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Caps Filters		
Queues		
Tee		
Topology dump		
Network Buffer Library	1.1.0	v1.0.0
On-Off Service APIs		
PSA Secure Storage API	1.0.4	
Power Management (PM)		v1.2.0
CPU Power Management	0.8.0	v2.6.0
Device	1.0.0	v2.7.0
Device Runtime	1.0.0	v3.0.0
S2RAM APIs		
States		
System	1.0.0	v1.2.0
Hooks		
Policy	1.0.0	v2.7.0
Precision Timing	0.1.0	v4.5.0
Precision Clock	0.1.0	v4.5.0
Precision Clock PTP Adapter	0.1.0	v4.5.0
Precision PI Controller	0.1.0	v4.5.0
Precision Time	0.1.0	v4.5.0
RTIO	0.2.0	v3.2.0
RTIO CQE Flags		
RTIO I2C IODEV Flags		
RTIO I3C IODEV Flags		
RTIO Operation Codes		
RTIO Priorities		
RTIO SQE Flags		
Retention API	0.1.0	v3.4.0
Boot mode interface		
Bootloader info interface	0.1.0	v3.5.0
Semihosting APIs		
Sensing		
Data Types		
Sensor Types (Sensing)		
Sensors (Sensing)		
Shell	1.0.0	v1.14.0
Shell backends		
ADSP memory window shell backend		
Dummy shell backend		
MQTT shell backend		
RPMsg shell backend		
RTT shell backend		
SSH shell backend		
Telnet shell backend		
UART shell backend		
Websocket shell backend		
Shell remote		
State Machine Framework API	0.2.0	
Statistics		
Storage APIs		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Disk Access Interface		
Stream to flash interface	0.1.0	v2.3.0
flash area Interface	1.1.1	v1.11.0
System Initialization	1.0.0	v1.0.0
System power off	1.0.0	v3.5.0
Task Watchdog APIs	0.8.0	v2.5.0
Timing Measurement APIs	1.0.0	v2.4.0
Arch specific Timing Measurement APIs		
Board specific Timing Measurement APIs		
SoC specific Timing Measurement APIs		
Tracing	0.8.0	v1.13.0
Tracing format		
Tracing hooks		
Backend-specific extensions		
Conditional variable		
Delayable work item		
Event		
FIFO		
GPIO		
Heap		
LIFO		
Mailbox		
Memory slab		
Message queue		
Mutex		
Network		
Network socket		
PM Device Runtime		
Pipe		
Polling		
Queue		
RTIO Tracing APIs		
Semaphore		
Stack		
Syscall Tracing		
System PM		
Thread		
Timer		
Triggered work item		
User-defined event		
Work item		
Work queue		
Tracing utility macros		
Xen support		
Xen Dom0 control		
Xen Dom0 domain control		
Xen Dom0 sysctl		
Xen HVM operations		
Xen XSM FLASK		
Xen device model		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Xen event channels		
Xen extended regions		
Xen grant tables		
Xen internal support		
Xen console driver		
Xen memory management		
Xen version queries		
Zbus APIs	1.0.0	v3.3.0
Zbus Proxy Agent API		
Zbus Proxy Agent IPC Backend		
Testing		
Emulator interfaces		
BBRAM emulator backend API		
BC1.2 backed emulator APIs		
Biometrics emulator test helpers		
Emulated ADC		
Emulated GPIO		
Emulated counter		
Fake CAN controller		
Fake EEPROM		
Fake LED controller		
Fake PWM		
Fake RTC		
Fake comparator		
Fake regulator		
Fuel gauge backend emulator APIs		
I2C Emulation Interface		
MSPI Emulation Interface		
SPI Emulation Interface		
Sensor emulator backend API		
UART Emulation Interface		
eSPI Emulation Interface		
FFF extensions		
Zephyr Benchmarking Framework		
Zephyr Testing Framework (ZTest)		
Ztest assertion macros		
Ztest assumption macros		
Ztest expectation macros		
Ztest mocking support		
Ztest testing macros		
Ztest ztress macros		
Third-party		
BBC micro:bit display APIs		
Grove display		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
MCUboot image control API		
UpdateHub Firmware Over-the-Air		
hawkBit Firmware Over-the-Air		
hawkBit autohandler API		
hawkBit configuration API		
hawkBit event API		
Toolchain abstraction		
USB Device Controller API	1.0.0	v1.5.0
USB Device Core API	1.0.0	v1.5.0
USB-C Device API	0.1.0	v3.3.0
Sink_callbacks		
Source_callbacks		
Utilities		
Base64		
Bit Reverse Operations		
COBS (Consistent Overhead Byte Stuffing)		
DSP (Digital Signal Processing)	0.1.0	v3.3.0
Basic Math Functions		
Vector Absolute Value		
Vector Addition		
Vector Clipping		
Vector Dot Product		
Vector Multiplication		
Vector Negate		
Vector Offset		
Vector Scale		
Vector Shift		
Vector Subtraction		
Vector bitwise AND		
Vector bitwise NOT		
Vector bitwise OR		
Vector bitwise XOR		
Float/Fixed point shift conversion functions		
Fixed to Float point conversions		
Float to Fixed point conversions		
Helper macros for printing Q values.		
Data Structure APIs		
Balanced Red/Black Tree		
Bit array	1.0.0	v2.6.0
Disjoint-set		
Doubly-linked list	1.0.0	v1.0.0
Flagged Single-linked list		
Hashmap		

continues on next page

Table 1 – continued from previous page

API	Version	Available in Zephyr Since
Hash Functions		
Hashmap Implementations		
MPSC (Multi producer, single consumer) packet buffer API		
MPSC (Multi producer, single consumer) packet header		
MPSC packet buffer flags		
MPSC Lockfree Queue API		
Min-heap		
Reference min-heap		
Ring Buffer APIs		
Ring queue		
SPSC (Single producer, single consumer) packet buffer API		
SPSC packet buffer flags		
SPSC API		
Single-linked list	1.0.0	v1.2.0
Formatted Output APIs		
Package convert flags		
Package flags		
cbvprintf processing flags.		
JSON	1.0.0	v1.10.0
JSON Web Token (JWT)		
Linear Range	1.0.0	v3.3.0
Math Ilog2 Functions		
Math Interpolation Functions		
Math extras	1.0.0	v2.0.0
Navigation		
Reference counting	0.1.0	
System I/O APIs	1.0.0	v1.0.0
Memory bit manipulation APIs		
Memory-mapped register I/O APIs		
Port I/O APIs		
Time Utility APIs	1.0.0	v2.0.0
Time Representation APIs		
Time Synchronization APIs		
Time Units Helpers		
Timespec Utility APIs		
UUID	0.8.0	v4.2.0
Utility Functions	0.1.0	v2.4.0

2.6.2 API Lifecycle

Developers using Zephyr's APIs need to know how long they can trust that a given API will not change in future releases. At the same time, developers maintaining and extending Zephyr's APIs need to be able to introduce new APIs that aren't yet fully proven, and to potentially retire old APIs when they're no longer optimal or supported by the underlying platforms.

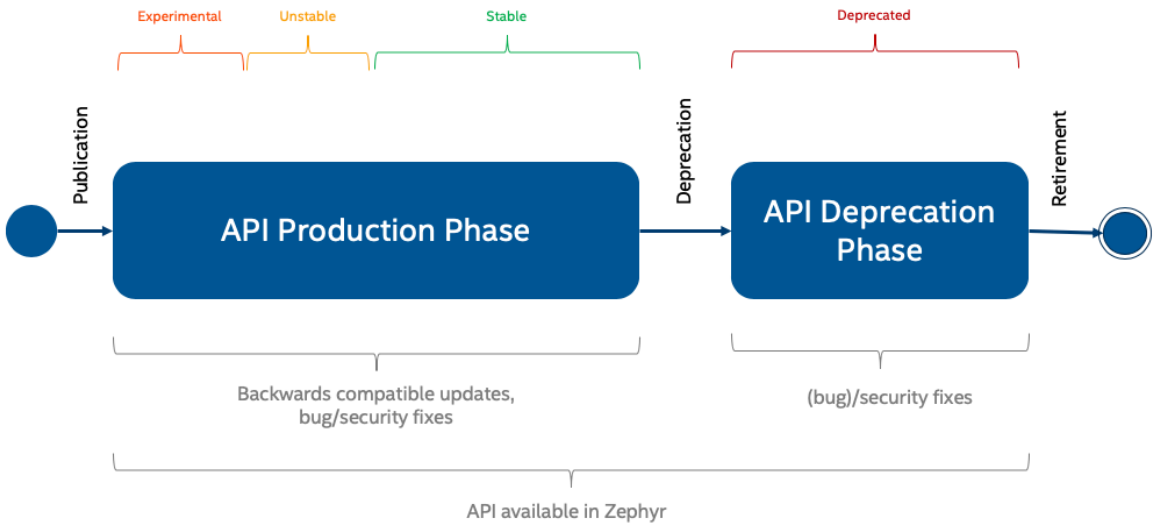


Fig. 2: API Life Cycle

An up-to-date table of all APIs and their maturity level can be found in the *API Overview* page.

Experimental

Experimental APIs denote that a feature was introduced recently, and may change or be removed in future versions. Try it out and provide feedback to the community via the Developer mailing list.

The following requirements apply to all new APIs:

- Documentation of the API (usage) explaining its design and assumptions, how it is to be used, current implementation limitations, and future potential, if appropriate.
- The API introduction should be accompanied by at least one implementation of said API (in the case of peripheral APIs, this corresponds to one driver)
- At least one sample using the new API (may only build on one single board)

When introducing a new and experimental API, mark the API version in the headers where the API is defined. An experimental API shall have a version where the minor version is up to one (0.1.z). (see *API Overview*)

Peripheral APIs (Hardware Related) When introducing an API (public header file with documentation) for a new peripheral or driver subsystem, review of the API is enforced and is driven by the Architecture working group consisting of representatives from different vendors.

The API shall be promoted to `unstable` when it has at least two implementations on different hardware platforms.

Unstable

The API is in the process of settling, but has not yet had sufficient real-world testing to be considered stable. The API is considered generic in nature and can be used on different hardware platforms.

When the API changes status to unstable API, mark the API version in the headers where the API is defined. Unstable APIs shall have a version where the minor version is larger than one (0.y.z | y > 1). (see *API Overview*)

> **Note**
>
> Changes will not be announced.

Peripheral APIs (Hardware Related) The API shall be promoted from `experimental` to `unstable` when it has at least two implementations on different hardware platforms.

Hardware Agnostic APIs For hardware agnostic APIs, multiple applications using it are required to promote an API from `experimental` to `unstable`.

Stable

The API has proven satisfactory, but cleanup in the underlying code may cause minor changes. Backwards-compatibility will be maintained if reasonable.

An API can be declared `stable` after fulfilling the following requirements:

- Test cases for the new API with 100% coverage
- Complete documentation in code. All public interfaces shall be documented and available in online documentation.
- The API has been in-use and was available in at least 2 development releases
- Stable APIs can get backward compatible updates, bug fixes and security fixes at any time.

In order to declare an API `stable`, the following steps need to be followed:

1. A Pull Request must be opened that changes the corresponding entry in the *API Overview* table
2. An email must be sent to the `devel` mailing list announcing the API upgrade request
3. The Pull Request must be submitted for discussion in the next Zephyr Architecture meeting where, barring any objections, the Pull Request will be merged

When the API changes status to stable API, mark the API version in the headers where the API is defined. Stable APIs shall have a version where the major version is one or larger (x.y.z | x >= 1). (see *API Overview*)

Introducing breaking API changes A stable API, as described above, strives to remain backwards-compatible through its life-cycle. There are however cases where fulfilling this objective prevents technical progress, or is simply unfeasible without unreasonable burden on the maintenance of the API and its implementation(s).

A breaking API change is defined as one that forces users to modify their existing code in order to maintain the current behavior of their application. The need for recompilation of applications (without changing the application itself) is not considered a breaking API change.

In order to restrict and control the introduction of a change that breaks the promise of backwards compatibility, the following steps must be followed whenever such a change is considered necessary in order to accept it in the project:

1. An *RFC issue* must be opened on GitHub with the following content:

```
Title:     RFC: Breaking API Change: <subsystem>
Contents:  - Problem Description:
             - Background information on why the change is required
           - Proposed Change (detailed):
             - Brief description of the API change
           - Detailed RFC:
             - Function call changes
             - Device Tree changes (source and bindings)
             - Kconfig option changes
           - Dependencies:
             - Impact to users of the API, including the steps required
               to adapt out-of-tree users of the API to the change
```

   Instead of a written description of the changes, the RFC issue may link to a Pull Request containing those changes in code form.

2. The RFC issue must be labeled with the GitHub `Breaking API Change` label
3. The RFC issue must be submitted for discussion in the next Zephyr Architecture meeting
4. An email must be sent to the `devel` mailing list with a subject identical to the RFC issue title and that links to the RFC issue

The RFC will then receive feedback through issue comments and will also be discussed in the Zephyr Architecture meeting, where the stakeholders and the community at large will have a chance to discuss it in detail.

Finally, and if not done as part of the first step, a Pull Request must be opened on GitHub. It is left to the person proposing the change to decide whether to introduce both the RFC and the Pull Request at the same time or to wait until the RFC has gathered consensus enough so that the implementation can proceed with confidence that it will be accepted. The Pull Request must include the following:

- A title that matches the RFC issue
- A link to the RFC issue
- The actual changes to the API
  - Changes to the API header file
  - Changes to the API implementation(s)
  - Changes to the relevant API documentation
  - Changes to Device Tree source and bindings
- The changes required to adapt in-tree users of the API to the change. Depending on the scope of this task this might require additional help from the corresponding maintainers
- An entry in the "API Changes" section of the release notes for the next upcoming release
- The labels `API`, `Breaking API Change` and `Release Notes`, as well as any others that are applicable
- The label `Architecture Review` if the RFC was not yet discussed and agreed upon in Zephyr Architecture meeting

Once the steps above have been completed, the outcome of the proposal will depend on the approval of the actual Pull Request by the maintainer of the corresponding subsystem. As with any other Pull Request, the author can request for it to be discussed and ultimately even voted on in the Zephyr TSC meeting.

If the Pull Request is merged then an email must be sent to the `devel` and `user` mailing lists informing them of the change.

The API version shall be changed to signal backward incompatible changes. This is achieved by incrementing the major version (X.y.z | X > 1). It MAY also include minor and patch level changes. Patch and minor versions MUST be reset to 0 when major version is incremented. (see *API Overview*)

> **Note**
>
> Breaking API changes will be listed and described in the migration guide.

Deprecated

> **Note**
>
> Unstable APIs can be removed without deprecation at any time. Deprecation and removal of APIs will be announced in the "API Changes" section of the release notes.

The following are the requirements for deprecating an existing API:

- Deprecation Time (stable APIs): 2 Releases The API needs to be marked as deprecated in at least two full releases. For example, if an API was first deprecated in release 4.0, it will be ready to be removed in 4.2 at the earliest. There may be special circumstances, determined by the Architecture working group, where an API is deprecated sooner.
- What is required when deprecating:
  - Mark as deprecated. This can be done by using the compiler itself (`__deprecated` for function declarations and `__DEPRECATED_MACRO` for macro definitions), or by introducing a Kconfig option (typically one that contains the `DEPRECATED` word in it) that, when enabled, reverts the APIs back to their previous form
  - Document the deprecation
  - Include the deprecation in the "API Changes" of the release notes for the next upcoming release
  - Code using the deprecated API needs to be modified to remove usage of said API
  - The change needs to be atomic and bisectable
  - Add an entry in the corresponding release GitHub issue tracking removal of deprecated APIs. In this example in the one corresponding to the 4.2 release.

During the deprecation waiting period, the API will be in the `deprecated` state. The Zephyr maintainers will track usage of deprecated APIs on `docs.zephyrproject.org` and support developers migrating their code. Zephyr will continue to provide warnings:

- API documentation will inform users that the API is deprecated.
- Attempts to use a deprecated API at build time will log a warning to the console.

Retired

In this phase, the API is removed.

The target removal date is 2 releases after deprecation is announced. The Zephyr maintainers will decide when to actually remove the API: this will depend on how many developers have successfully migrated from the deprecated API, and on how urgently the API needs to be removed.

If it's OK to remove the API, it will be removed. The maintainers will remove the corresponding documentation, and communicate the removal in the usual ways: the release notes, mailing lists, Github issues and pull-requests.

If it's not OK to remove the API, the maintainers will continue to support migration and update the roadmap with the aim to remove the API in the next release.

2.6.3 Vendor-Specific API Placement Policy

Overview

This policy defines how vendor-specific hardware functionality must be contributed, structured, and exposed within the Zephyr Project. It ensures that Zephyr's public API surface remains stable, that vendor extensions do not bypass the standard API lifecycle, and that potential cross-vendor features have a clear path to evolve into generic driver classes.

Vendor-specific APIs must not be exposed as public Zephyr interfaces unless they meet the criteria required for inclusion in Zephyr's formal API lifecycle (experimental stable deprecated). Features that are inherently tied to a single SoC or vendor must remain encapsulated within the vendor's SoC directories.

This policy provides a consistent decision model for contributors and maintainers.

Guiding Principles

- **Public Zephyr APIs must not contain vendor-specific interfaces.**

  Interfaces placed under `include/zephyr/` are expected to evolve through the standard API lifecycle. Vendor-specific interfaces that cannot satisfy this evolution must not be exposed as public Zephyr APIs.

- **Prefer generic solutions over vendor-specific ones.**

  When a hardware feature applies to multiple vendors or might in the future it should be implemented as part of an existing driver class or as a new cross-vendor driver class.

- **Vendor-specific features must be encapsulated.**

  Vendor-only mechanisms should live within the vendor's platform namespace (SoCs, boards, etc.).

- **Evolution must remain possible.**

  Placement must not prevent later promotion of a vendor-specific implementation into a portable Zephyr driver class.

Decision Process

When adding new hardware functionality or vendor-specific behavior, contributors must follow the decision tree below, in the given order:

- **Does the feature logically extend an existing driver class?**

  If yes, extend the existing class; do not create a vendor-specific API.

- **Could the feature become a cross-vendor driver class?**

  If the hardware concept is not clearly unique to a single vendor, or resembles patterns likely to appear across multiple architectures, create a new driver class.

- **Is the functionality clearly vendor-specific and unlikely to generalize?**

  In this case, place both headers and implementations under:

  ```
  soc/<vendor>/..
  ```

  This ensures true encapsulation, long-term maintainability, and clean removal when an SoC is dropped.

Vendor-specific code should be implemented in headers located under `soc/<vendor>/include` and code in `soc/<vendor>/common` or `soc/<vendor>/<series>`.

Such interfaces would be private and/or internal to the vendor to support their specific hardware and will be scrutinized in order to avoid creating feature islands specific to vendors.

Applications using these interfaces, given that they are internal and/or private to the vendor, shall expect no guarantees as to the lifecycle or backwards compatibility.

Allowed Locations Summary

1. When extending existing driver classes

- Where:

```
include/zephyr/drivers/<class>.h
drivers/<class>/
```

- Use when:

  The functionality fits or extends an existing class.

2. Vendor-specific extensions to existing driver classes

- Where:

```
include/zephyr/drivers/<class>/
drivers/<class>/
```

- Use when:

  The feature is a vendor-specific extension to an existing class, but the core concept is not unique to a single vendor and could be standardized in the future.

3. Create New Driver Class (Cross-Vendor)

- Where:

```
include/zephyr/drivers/<class>.h
drivers/<class>/
```

- Use when:

  The feature could reasonably apply to multiple vendors or evolve into a generic abstraction.

4. Vendor-Specific, SoC-Local Code (Default for Vendor-Unique Features)

- Where:

  Vendor-specific code should be implemented in headers located under `soc/<vendor>/include` and code in `soc/<vendor>/common` or `soc/<vendor>/<series>`.

- Use when:

  The feature is unique to a vendor or SoC and is not intended as a public API.

2.6.4 API Design Guidelines

Zephyr development and evolution is a group effort, and to simplify maintenance and enhancements there are some general policies that should be followed when developing a new capability or interface.

All public API must be documented using Doxygen. See *Doxygen Style Guidelines* for details.

Using Callbacks

Many APIs involve passing a callback as a parameter or as a member of a configuration structure. The following policies should be followed when specifying the signature of a callback:

- The first parameter should be a pointer to the object most closely associated with the callback. In the case of device drivers this would be `const struct device *dev`. For library functions it may be a pointer to another object that was referenced when the callback was provided.
- The next parameter(s) should be additional information specific to the callback invocation, such as a channel identifier, new status value, and/or a message pointer followed by the message length.
- The final parameter should be a `void *user_data` pointer carrying context that allows a shared callback function to locate additional material necessary to process the callback.

An exception to providing `user_data` as the last parameter may be allowed when the callback itself was provided through a structure that will be embedded in another structure. An example of such a case is `gpio_callback`, normally defined within a data structure specific to the code that also defines the callback function. In those cases further context can accessed by the callback indirectly by `CONTAINER_OF`.

Examples

- The requirements of `k_timer_expiry_t` invoked when a system timer alarm fires are satisfied by:

  ```
  void handle_timeout(struct k_timer *timer)
  { ... }
  ```

  The assumption here, as with `gpio_callback`, is that the timer is embedded in a structure reachable from `CONTAINER_OF` that can provide additional context to the callback.

- The requirements of `counter_alarm_callback_t` invoked when a counter device alarm fires are satisfied by:

  ```
  void handle_alarm(const struct device *dev,
                    uint8_t chan_id,
                    uint32_t ticks,
                    void *user_data)
  { ... }
  ```

  This provides more complete useful information, including which counter channel timed-out and the counter value at which the timeout occurred, as well as user context which may or may not be the `counter_alarm_cfg` used to register the callback, depending on user needs.

Conditional Data and APIs

APIs and libraries may provide features that are expensive in RAM or code size but are optional in the sense that some applications can be implemented without them. Examples of such feature include `capturing a timestamp` or `providing an alternative interface`. The developer in coordination with the community must determine whether enabling the features is to be controllable through a Kconfig option.

In the case where a feature is determined to be optional the following practices should be followed.

- Any data that is accessed only when the feature is enabled should be conditionally included via `#ifdef CONFIG_MYFEATURE` in the structure or union declaration. This reduces memory use for applications that don't need the capability.

- Function declarations that are available only when the option is enabled should be provided unconditionally. Add a note in the description that the function is available only when the specified feature is enabled, referencing the required Kconfig symbol by name. In the cases where the function is used but not enabled the definition of the function shall be excluded from compilation, so references to the unsupported API will result in a link-time error.
- Where code specific to the feature is isolated in a source file that has no other content that file should be conditionally included in `CMakeLists.txt`:

```
zephyr_sources_ifdef(CONFIG_MYFEATURE foo_funcs.c)
```

- Where code specific to the feature is part of a source file that has other content the feature-specific code should be conditionally processed using `#ifdef CONFIG_MYFEATURE`.

See *Conditional code* for details on how to ensure conditional code is visible to Doxygen and included in the public API documentation.

Return Codes

Implementations of an API, for example an API for accessing a peripheral might implement only a subset of the functions that is required for minimal operation. A distinction is needed between APIs that are not supported and those that are not implemented or optional:

- APIs that are supported but not implemented shall return `-ENOSYS`.
- Optional APIs that are not supported by the hardware should be implemented and the return code in this case shall be `-ENOTSUP`.
- When an API is implemented, but the particular combination of options requested in the call cannot be satisfied by the implementation the call shall return `-ENOTSUP`. (For example, a request for a level-triggered GPIO interrupt on hardware that supports only edge-triggered interrupts)

2.6.5 API Terminology

The following terms may be used as shorthand API tags to indicate the allowed calling context (thread, ISR, pre-kernel), the effect of a call on the current thread state, and other behavioral characteristics.

reschedule
: if executing the function reaches a reschedule point

sleep
: if executing the function can cause the invoking thread to sleep

no-wait
: if a parameter to the function can prevent the invoking thread from trying to sleep

isr-ok
: if the function can be safely called and will have its specified effect whether invoked from interrupt or thread context

pre-kernel-ok
: if the function can be safely called before the kernel has been fully initialized and will have its specified effect when invoked from that context.

async
: if the function may return before the operation it initializes is complete (i.e. function return and operation completion are asynchronous)

supervisor
: if the calling thread must have supervisor privileges to execute the function

Details on the behavioral impact of each attribute are in the following sections.

reschedule

The reschedule attribute is used on a function that can reach a *reschedule point* within its execution.

Details The significance of this attribute is that when a rescheduling function is invoked by a thread it is possible for that thread to be suspended as a consequence of a higher-priority thread being made ready. Whether the suspension actually occurs depends on the operation associated with the reschedule point and the relative priorities of the invoking thread and the head of the ready queue.

Note that in the case of timeslicing, or reschedule points executed from interrupts, any thread may be suspended in any function.

Functions that are not **reschedule** may be invoked from either thread or interrupt context.

Functions that are **reschedule** may be invoked from thread context.

Functions that are **reschedule** but not **sleep** may be invoked from interrupt context.

sleep

The sleep attribute is used on a function that can cause the invoking thread to *sleep*.

Explanation This attribute is of relevance specifically when considering applications that use only non-preemptible threads, because the kernel will not replace a running cooperative-only thread at a reschedule point unless that thread has explicitly invoked an operation that caused it to sleep.

This attribute does not imply the function will sleep unconditionally, but that the operation may require an invoking thread that would have to suspend, wait, or invoke `k_yield()` before it can complete its operation. This behavior may be mediated by **no-wait**.

Functions that are **sleep** are implicitly **reschedule**.

Functions that are **sleep** may be invoked from thread context.

Functions that are **sleep** may be invoked from interrupt and pre-kernel contexts if and only if invoked in **no-wait** mode.

no-wait

The no-wait attribute is used on a function that is also **sleep** to indicate that a parameter to the function can force an execution path that will not cause the invoking thread to sleep.

Explanation The paradigmatic case of a no-wait function is a function that takes a timeout, to which `K_NO_WAIT` can be passed. The semantics of this special timeout value are to execute the function's operation as long as it can be completed immediately, and to return an error code rather than sleep if it cannot.

It is use of the no-wait feature that allows functions like `k_sem_take()` to be invoked from ISRs, since it is not permitted to sleep in interrupt context.

A function with a no-wait path does not imply that taking that path guarantees the function is synchronous.

Functions with this attribute may be invoked from interrupt and pre-kernel contexts only when the parameter selects the no-wait path.

isr-ok

The isr-ok attribute is used on a function to indicate that it works whether it is being invoked from interrupt or thread context.

Explanation Any function that is not **sleep** is inherently **isr-ok**. Functions that are **sleep** are **isr-ok** if the implementation ensures that the documented behavior is implemented even if called from an interrupt context. This may be achieved by having the implementation detect the calling context and transfer the operation that would sleep to a thread, or by documenting that when invoked from a non-thread context the function will return a specific error (generally `-EWOULDBLOCK`).

Note that a function that is **no-wait** is safe to call from interrupt context only when the no-wait path is selected. **isr-ok** functions need not provide a no-wait path.

pre-kernel-ok

The pre-kernel-ok attribute is used on a function to indicate that it works as documented even when invoked before the kernel main thread has been started.

Explanation This attribute is similar to **isr-ok** in function, but is intended for use by any API that is expected to be called in `DEVICE_DEFINE()` or `SYS_INIT()` calls that may be invoked with `PRE_KERNEL_1` or `PRE_KERNEL_2` initialization levels.

Generally a function that is **pre-kernel-ok** checks `k_is_pre_kernel()` when determining whether it can fulfill its required behavior. In many cases it would also check `k_is_in_isr()` so it can be **isr-ok** as well.

async

A function is **async** (i.e. asynchronous) if it may return before the operation it initiates has completed. An asynchronous function will generally provide a mechanism by which operation completion is reported, e.g. a callback or event.

A function that is not asynchronous is synchronous, i.e. the operation will always be complete when the function returns. As most functions are synchronous this behavior does not have a distinct attribute to identify it.

Explanation Be aware that **async** is orthogonal to context-switching. Some APIs may provide completion information through a callback, but may suspend while waiting for the resource necessary to initiate the operation; an example is `spi_transceive_signal()`.

If a function is both **no-wait** and **async** then selecting the no-wait path only guarantees that the function will not sleep. It does not affect whether the operation will be completed before the function returns.

supervisor

The supervisor attribute is relevant only in user-mode applications, and indicates that the function cannot be invoked from user mode.

2.7 Language Support

2.7.1 C Language Support

C is a general-purpose low-level programming language that is widely used for writing code for embedded systems.

Zephyr is primarily written in C and natively supports applications written in the C language. All Zephyr API functions and macros are implemented in C and available as part of the C header files under the `include` directory, so writing Zephyr applications in C gives the developers access to the most features.

The `main()` function must have the return type of `int` as Zephyr applications run in a "hosted" environment as defined by the C standard. Applications must return zero (0) from main. All non-zero return values are reserved.

Language Standards

Zephyr does not target a specific version of the C standards; however, the Zephyr codebase makes extensive use of the features newly introduced in the 1999 release of the ISO C standard (ISO/IEC 9899:1999, hereinafter referred to as C99) such as those listed below, effectively requiring the use of a compiler toolchain that supports the C99 standard and above:

- inline functions
- standard boolean types (`bool` in `<stdbool.h>`)
- fixed-width integer types (`[u]intN_t` in `<stdint.h>`)
- designated initializers
- variadic macros
- `restrict` qualification

Additionally, some components or parts of them make use of features introduced in the C11 and C17 versions of the standard (ISO/IEC 9899:2011 and 9899:2018 respectively):

- _Generic keyword
- _Static_assert keyword

In summary, it is recommended to use a compiler toolchain that supports at least the C17 standard for developing with Zephyr. It is, however, important to note that some optional Zephyr components and external modules may make use of the C language features that have been introduced in more recent versions of the standards, in which case it will be necessary to use a more up-to-date compiler toolchain that supports such standards.

Standard Library

The C Standard Library is an integral part of any C program, and Zephyr provides the support for a number of different C libraries for the applications to choose from, depending on the compiler toolchain being used to build the application.

Common C library code Zephyr provides some C library functions that are designed to be used in conjunction with multiple C libraries. These either provide functions not available in multiple C libraries or are designed to replace functionality in the C library with code better suited for use in the Zephyr environment

Time function This provides an implementation of the standard C function, `time()`, relying on the Zephyr function, `sys_clock_gettime()`. This function can be enabled by selecting `COMMON_LIBC_TIME`.

Dynamic Memory Management The common dynamic memory management implementation can be enabled by selecting the `CONFIG_COMMON_LIBC_MALLOC` in the application configuration file.

The common C library internally uses the *kernel memory heap API* to manage the memory heap used by the standard dynamic memory management interface functions such as `malloc()` and `free()`.

The internal memory heap is normally located in the `.bss` section. When userspace is enabled, however, it is placed in a dedicated memory partition called `z_malloc_partition`, which can be accessed from the user mode threads. The size of the internal memory heap is specified by the `CONFIG_COMMON_LIBC_MALLOC_ARENA_SIZE`.

The default heap size for applications using the common C library is zero (no heap). For other C library users, if there is an MMU present, then the default heap is 16kB. Otherwise, the heap uses all available memory.

There are also separate controls to select `calloc()` (`COMMON_LIBC_CALLOC`) and `reallocarray()` (`COMMON_LIBC_REALLOCARRAY`). Both of these are enabled by default as that doesn't impact memory usage in applications not using them.

The standard dynamic memory management interface functions implemented by the common C library are thread safe and may be simultaneously called by multiple threads. These functions are implemented in `lib/libc/common/source/stdlib/malloc.c`.

Minimal libc The most basic C library, named "minimal libc", is part of the Zephyr codebase and provides the minimal subset of the standard C library required to meet the needs of Zephyr and its subsystems, primarily in the areas of string manipulation and display.

It is very low footprint and is suitable for projects that do not rely on less frequently used portions of the ISO C standard library. It can also be used with a number of different toolchains.

The minimal libc implementation can be found in `lib/libc/minimal` in the main Zephyr tree.

Functions The minimal libc implements the minimal subset of the ISO/IEC 9899:2011 standard C library functions required to meet the needs of the Zephyr kernel, as defined by the *Coding Guidelines Rule A.4*.

Formatted Output The minimal libc does not implement its own formatted output processor; instead, it maps the C standard formatted output functions such as `printf` and `sprintf` to the `cbprintf()` function, which is Zephyr's own C99-compatible formatted output implementation.

For more details, refer to the *Formatted Output* OS service documentation.

Dynamic Memory Management The minimal libc uses the malloc api family implementation provided by the *common C library*, which itself is built upon the *kernel memory heap API*.

Error numbers Error numbers are used throughout Zephyr APIs to signal error conditions as return values from functions. They are typically returned as the negative value of the integer literals defined in this section, and are defined in the `errno.h` header file.

A subset of the error numbers defined in the POSIX errno.h specification and other de-facto standard sources have been added to the minimal libc.

A conscious effort is made in Zephyr to keep the values of the minimal libc error numbers consistent with the different implementations of the C standard libraries supported by Zephyr. The minimal libc `errno.h` is checked against that of the *Newlib* to ensure that the error numbers are kept aligned.

Below is a list of the error number definitions. For the actual numeric values please refer to errno.h.

Error numbers

Newlib Newlib is a complete C library implementation written for the embedded systems. It is a separate open source project and is not included in source code form with Zephyr. Instead, the *Zephyr SDK* includes a precompiled library for each supported architecture (`libc.a` and `libm.a`).

> **Note**
>
> Other 3rd-party toolchains, such as *GNU Arm Embedded*, also bundle the Newlib as a precompiled library.

Zephyr implements the "API hook" functions that are invoked by the C standard library functions in the Newlib. These hook functions are implemented in `lib/libc/newlib/libc-hooks.c` and translate the library internal system calls to the equivalent Zephyr API calls.

Types of Newlib The Newlib included in the *Zephyr SDK* comes in two versions: 'full' and 'nano' variants.

Full Newlib The Newlib full variant (`libc.a` and `libm.a`) is the most capable variant of the Newlib available in the Zephyr SDK, and supports almost all standard C library features. It is optimized for performance (prefers performance over code size) and its footprint is significantly larger than the nano variant.

This variant can be enabled by selecting the `CONFIG_NEWLIB_LIBC` and de-selecting the `CONFIG_NEWLIB_LIBC_NANO` in the application configuration file.

Nano Newlib The Newlib nano variant (`libc_nano.a` and `libm_nano.a`) is the size-optimized version of the Newlib, and supports all features that the full variant supports except the new format specifiers introduced in C99, such as the `char`, `long long` type format specifiers (i.e. `%hhX` and `%llX`).

This variant can be enabled by selecting the `CONFIG_NEWLIB_LIBC` and `CONFIG_NEWLIB_LIBC_NANO` in the application configuration file.

Note that the Newlib nano variant is not available for all architectures. The availability of the nano variant is specified by the `CONFIG_HAS_NEWLIB_LIBC_NANO`.

Formatted Output Newlib supports all standard C formatted input and output functions, including `printf`, `fprintf`, `sprintf` and `sscanf`.

The Newlib formatted input and output function implementation supports all format specifiers defined by the C standard with the following exceptions:

- Floating point format specifiers (e.g. `%f`) require `CONFIG_NEWLIB_LIBC_FLOAT_PRINTF` and `CONFIG_NEWLIB_LIBC_FLOAT_SCANF` to be enabled.
- C99 format specifiers are not supported in the Newlib nano variant (i.e. `%hhX` for `char`, `%llX` for `long long`, `%jX` for `intmax_t`, `%zX` for `size_t`, `%tX` for `ptrdiff_t`).

Dynamic Memory Management Newlib implements an internal heap allocator to manage the memory blocks used by the standard dynamic memory management interface functions (for example, `malloc()` and `free()`).

The internal heap allocator implemented by the Newlib may vary across the different types of the Newlib used. For example, the heap allocator implemented in the Full Newlib (`libc.a` and `libm.a`) of the Zephyr SDK requests larger memory chunks to the operating system and has a significantly higher minimum memory requirement compared to that of the Nano Newlib (`libc_nano.a` and `libm_nano.a`).

The only interface between the Newlib dynamic memory management functions and the Zephyr-side libc hooks is the `sbrk()` function, which is used by the Newlib to manage the size of the memory pool reserved for its internal heap allocator.

The `_sbrk()` hook function, implemented in `libc-hooks.c`, handles the memory pool size change requests from the Newlib and ensures that the Newlib internal heap allocator memory pool size

does not exceed the amount of available memory space by returning an error when the system is out of memory.

When userspace is enabled, the Newlib internal heap allocator memory pool is placed in a dedicated memory partition called `z_malloc_partition`, which can be accessed from the user mode threads.

The amount of memory space available for the Newlib heap depends on the system configurations:

- When MMU is enabled (`CONFIG_MMU` is selected), the amount of memory space reserved for the Newlib heap is set by the size of the free memory space returned by the `k_mem_free_get()` function or the `CONFIG_NEWLIB_LIBC_MAX_MAPPED_REGION_SIZE`, whichever is the smallest.
- When MPU is enabled and the MPU requires power-of-two partition size and address alignment (`CONFIG_NEWLIB_LIBC_ALIGNED_HEAP_SIZE` is set to a non-zero value), the amount of memory space reserved for the Newlib heap is set by the `CONFIG_NEWLIB_LIBC_ALIGNED_HEAP_SIZE`.
- Otherwise, the amount of memory space reserved for the Newlib heap is equal to the amount of free (unallocated) memory in the SRAM region.

The standard dynamic memory management interface functions implemented by the Newlib are thread safe and may be simultaneously called by multiple threads.

Picolibc Picolibc is a complete C library implementation written for the embedded systems, targeting C17 (ISO/IEC 9899:2018) and POSIX 2018 (IEEE Std 1003.1-2017) standards. Picolibc is an external open source project which is provided for Zephyr as a module, and included as part of the *Zephyr SDK* in precompiled form for each supported architecture (`libc.a`).

> **Note**
>
> Picolibc is also available for other 3rd-party toolchains, such as *GNU Arm Embedded*.

Zephyr implements the "API hook" functions that are invoked by the C standard library functions in the Picolibc. These hook functions are implemented in lib/libc/picolibc/ and translate the library internal system calls to the equivalent Zephyr API calls.

Picolibc Module When built as a Zephyr module, there are several configuration knobs available to adjust the feature set in the library, balancing what the library supports versus the code size of the resulting functions. Because the standard C++ library must be compiled for the target C library, the Picolibc module cannot be used with applications which use the standard C++ library. Building the Picolibc module will increase the time it takes to compile the application.

The Picolibc module can be enabled by selecting `CONFIG_PICOLIBC_USE_MODULE` in the application configuration file.

When updating the Picolibc module to a newer version, the *toolchain-bundled Picolibc in the Zephyr SDK* must also be updated to the same version.

Toolchain Picolibc Starting with version 0.16, the Zephyr SDK includes precompiled versions of Picolibc for every target architecture, along with precompiled versions of libstdc++.

The toolchain version of Picolibc can be enabled by de-selecting `CONFIG_PICOLIBC_USE_MODULE` in the application configuration file.

For every release of Zephyr, the toolchain-bundled Picolibc and the *Picolibc module* are guaranteed to be in sync when using the *recommended version of Zephyr SDK*.

Building Without Toolchain bundled Picolibc For toolchain where there is no bundled Picolibc, it is still possible to use Picolibc by building it from source. Note that any restrictions mentioned in *Picolibc Module* still apply.

To build without toolchain bundled Picolibc, the toolchain must enable `CONFIG_PICOLIBC_SUPPORTED`. For example, this needs to be added to the toolchain Kconfig file:

```
config TOOLCHAIN_<name>_PICOLIBC_SUPPORTED
   def_bool y
   select PICOLIBC_SUPPORTED
```

By enabling `CONFIG_PICOLIBC_SUPPORTED`, the build system would automatically build Picolibc from source with its module when there is no toolchain bundled Picolibc.

Formatted Output Picolibc supports all standard C formatted input and output functions, including `printf()`, `fprintf()`, `sprintf()` and `sscanf()`.

Picolibc formatted input and output function implementation supports all format specifiers defined by the C17 and POSIX 2018 standards with the following exceptions:

- Floating point format specifiers (e.g. `%f`) require `CONFIG_PICOLIBC_IO_FLOAT`.
- Long long format specifiers (e.g. `%lld`) require `CONFIG_PICOLIBC_IO_LONG_LONG`. This option is automatically enabled with `CONFIG_PICOLIBC_IO_FLOAT`.

Printk, cbprintf and friends When using Picolibc, Zephyr formatted output functions are implemented in terms of stdio calls. This includes:

- printk, snprintk and vsnprintk
- cbprintf and cbvprintf
- fprintfcb, vfprintfcb, printfcb, vprintfcb, snprintfcb and vsnprintfcb

When using tagged args (`CONFIG_CBPRINTF_PACKAGE_SUPPORT_TAGGED_ARGUMENTS` and `CBPRINTF_PACKAGE_ARGS_ARE_TAGGED`), calls to cbpprintf will not use Picolibc, so formatting of output using those code will differ from Picolibc results as the cbprintf functions are not completely C/POSIX compliant.

Math Functions Picolibc provides full C17/IEEE STD 754-2019 support for float, double and long double math operations, except for long double versions of the Bessel functions.

Thread Local Storage Picolibc uses Thread Local Storage (TLS) (where supported) for data which is supposed to remain local to each thread, like `errno`. This means that TLS support is enabled when using Picolibc. As all TLS variables are allocated out of the thread stack area, this can affect stack size requirements by a few bytes.

C Library Local Variables Picolibc uses a few internal variables for things like heap management. These are collected in a dedicated memory partition called `z_libc_partition`. Applications using `CONFIG_USERSPACE` and memory domains must ensure that this partition is included in any domain active during Picolibc calls.

Dynamic Memory Management Picolibc uses the malloc api family implementation provided by the *common C library*, which itself is built upon the *kernel memory heap API*.

Formatted Output

C defines standard formatted output functions such as `printf` and `sprintf` and these functions are implemented by the C standard libraries.

Each C standard library has its own set of requirements and configurations for selecting the formatted output modes and capabilities. Refer to each C standard library documentation for more details.

Dynamic Memory Management

C defines a standard dynamic memory management interface (for example, `malloc()` and `free()`) and these functions are implemented by the C standard libraries.

While the details of the dynamic memory management implementation varies across different C standard libraries, all supported libraries must conform to the following conventions. Every supported C standard library shall:

- manage its own memory heap either internally or by invoking the hook functions (for example, `sbrk()`) implemented in `libc-hooks.c`.
- maintain the architecture- and memory region-specific alignment requirements for the memory blocks allocated by the standard dynamic memory allocation interface (for example, `malloc()`).
- allocate memory blocks inside the `z_malloc_partition` memory partition when userspace is enabled. See *Pre-defined Memory Partitions*.

For more details regarding the C standard library-specific memory management implementation, refer to each C standard library documentation.

> **Note**
>
> Native Zephyr applications should use the *memory management API* supported by the Zephyr kernel such as `k_malloc()` in order to take advantage of the advanced features that they offer.
>
> C standard dynamic memory management interface functions such as `malloc()` should be used only by the portable applications and libraries that target multiple operating systems.

2.7.2 C++ Language Support

C++ is a general-purpose object-oriented programming language that is based on the C language.

Enabling C++ Support

Zephyr supports applications written in both C and C++. However, to use C++ in an application you must configure Zephyr to include C++ support by selecting the `CONFIG_CPP` in the application configuration file.

To enable C++ support, the compiler toolchain must also include a C++ compiler and the included compiler must be supported by the Zephyr build system. The *Zephyr SDK*, which includes the GNU C++ Compiler (part of GCC), is supported by Zephyr, and the features and their availability documented here assume the use of the Zephyr SDK.

The default C++ standard level (i.e. the language enforced by the compiler flags passed) for Zephyr apps is C++11. Other standards are available via kconfig choice, for example `CONFIG_STD_CPP98`. The oldest standard supported and tested in Zephyr is C++98.

When compiling a source file, the build system selects the C++ compiler based on the suffix (extension) of the files. Files identified with either a **cpp** or a **cxx** suffix are compiled using the C++ compiler. For example, `myCplusplusApp.cpp` is compiled using C++.

The C++ standard requires the `main()` function to have the return type of `int`. Your `main()` must be defined as `int main(void)` or `int main(int, char **)`. To use main with arguments the `CONFIG_BOOTARGS` option has to be selected. Zephyr ignores the return value from main. However, Applications must return zero (0). All non-zero return values are reserved.

> **Note**
>
> Do not use C++ for kernel, driver, or system initialization code.

Language Features

Zephyr currently provides only a subset of C++ functionality. The following features are *not* supported:

- Static global object destruction
- OS-specific C++ standard library classes (e.g. `std::thread`, `std::mutex`)

While not an exhaustive list, support for the following functionality is included:

- Inheritance
- Virtual functions
- Virtual tables
- Static global object constructors
- Dynamic object management with the **new** and **delete** operators
- Exceptions
- RTTI (runtime type information)
- Standard Template Library (STL)

Static global object constructors are initialized after the drivers are initialized but before the application `main()` function. Therefore, use of C++ is restricted to application code.

In order to make use of the C++ exceptions, the `CONFIG_CPP_EXCEPTIONS` must be selected in the application configuration file.

Zephyr Minimal C++ Library

Zephyr minimal C++ library (`lib/cpp/minimal`) provides a minimal subset of the C++ standard library and application binary interface (ABI) functions to enable basic C++ language support. This includes:

- `new` and `delete` operators
- virtual function stub and vtables
- static global initializers for global constructors

The scope of the minimal C++ library is strictly limited to providing the basic C++ language support, and it does not implement any Standard Template Library (STL) classes and functions. For this reason, it is only suitable for use in the applications that implement their own (non-standard) class library and do not rely on the Standard Template Library (STL) components.

Any application that makes use of the Standard Template Library (STL) components, such as `std::string` and `std::vector`, must enable the C++ standard library support.

C++ Standard Library

The C++ Standard Library is a collection of classes and functions that are part of the ISO C++ standard (`std` namespace).

Zephyr does not include any C++ standard library implementation in source code form. Instead, it allows configuring the build system to link against the pre-built C++ standard library included in the C++ compiler toolchain.

To enable C++ standard library, select an applicable toolchain-specific C++ standard library type from the `CONFIG_LIBCPP_IMPLEMENTATION` in the application configuration file.

For instance, when building with the *Zephyr SDK*, the build system can be configured to link against the GNU C++ Library (`libstdc++.a`), which is a fully featured C++ standard library that provides all features required by the ISO C++ standard including the Standard Template Library (STL), by selecting `CONFIG_GLIBCXX_LIBCPP` in the application configuration file.

The following C++ standard libraries are supported by Zephyr:

- GNU C++ Library (`CONFIG_GLIBCXX_LIBCPP`)
- ARC MetaWare C++ Library (`CONFIG_ARCMWDT_LIBCPP`)

A Zephyr subsystem that requires the features from the full C++ standard library can select, from its config, `CONFIG_REQUIRES_FULL_LIBCPP`, which automatically selects a compatible C++ standard library unless the Kconfig symbol for a specific C++ standard library is selected.

Header files and incompatibilities between C and C++

To interact with each other, C and C++ must share code through header files: data structures, macros, static functions,... While C and C++ have a large overlap, they're different languages with known incompatibilities. C is not just a C++ subset. Standard levels (e.g.: "C++11") add another level of complexity as new features are often inspired by and copied from the other language but many years later and with subtle differences. Making things more complex, compilers often offer early prototypes of features before they become standardized. Standards can have ambiguities interpreted differently by different compilers. Compilers can have bugs and these may need workarounds. To help with this, many projects restrict themselves to a limited number of toolchains. Zephyr does not.

These compatibility issues affect header files dis-proportionally. Not just because they have to be compatible between C and C++, but also because they end up being compiled in a surprisingly high number of other source files due to *indirect* inclusion and the lack of structure and headers organization that is typical in real-world projects. So, header files are exposed to a much larger variety of toolchains and project configurations. Adding more constraints, the Zephyr project has demanding policies with respect to code style, compiler warnings, static analyzers and standard compliance (e.g.: MISRA).

Put together, all these constraints can make writing header files very challenging. The purpose of this section is to document some best "header practices" and lessons learned in a Zephyr-specific context. While a lot of the information here is not Zephyr-specific, this section is not a substitute for knowledge of C/C++ standards, textbooks and other references.

Testing Fortunately, the Zephyr project has an extensive test and CI infrastructure that provides coverage baselines, catches issues early, enforces policies and maintains such combinatorial explosions under some control. The `tests/lib/cpp/cxx/` are very useful in this context because their `testcase.yaml` configuration lets `twister` iterate quickly over a range of `-std` parameters: `-std=c++98`, `-std=c++11`, etc.

Keep in mind unused macros are not compiled.

Designated initializers Initialization macros are common in header files as they help reduce boilerplate code. C99 added initialization of `struct` and `union` types by "designated" member

names instead of a list of *bare* expressions. Some GCC versions support designated initializers even in their C90 mode.

When used at a simple level, designated initializers are less error-prone, more readable and more flexible. On the other hand, C99 allowed a surprisingly large and lax set of possibilities: designated initializers can be out of order, duplicated, "nested" (`.a.x =`), various other braces can be omitted, designated initializers and not can be mixed, etc.

Twenty years later, C++20 added designated initializers to C++ but in much more restricted way; partly because a C++ `struct` is actually a `class`. As described in the C++ proposal number P0329 (which compares with C) or in any complete C++ reference, a mix is not allowed and initializers must be in order (gaps are allowed).

Interestingly, the new restrictions in C++20 can cause `gcc -std=c++20` to fail to compile code that successfully compiles with `gcc -std=c++17`. For example, `gcc -std=c++17` and older allow the C-style mix of initializers and bare expressions. This fails to compile with using `gcc -std=c++20` *with the same GCC version*.

Recommendation: to maximize compatibility across different C and C++ toolchains and standards, designated initializers in Zephyr header files should follow all C++20 rules and restrictions. Non-designated, pre-C99 initialization offers more compatibility and is also allowed but designated initialization is the more readable and preferred code style. In any case, both styles must never be mixed in the same initializer.

Warning: successful compilation is not the end of the incompatibility story. For instance, the *evaluation order* of initializer expressions is unspecified in C99! It is the (expected) left-to-right order in C++20. Other standard revisions may vary. In doubt, do not rely on evaluation order (here and elsewhere).

Anonymous unions Anonymous unions (a.k.a. "unnamed" unions) seem to have been part of C++ from its very beginning. They were not officially added to C until C11. As usual, there are differences between C and C++. For instance, C supports anonymous unions only as a member of an enclosing `struct` or `union`, empty lists `{ }` have always been allowed in C++ but they require C23, etc.

When initializing anonymous members, the expression can be enclosed in braces or not. It can be either designated or bare. For maximum portability, when initializing *anonymous unions*:

- Do *not* enclose *designated* initializers with braces. This is required by C++20 and above which perceive such braces as mixing bare expressions with (other) designated initializers and fails to compile.
- Do enclose *bare* expressions with braces. This is required by C. Maybe because C is laxer and allows many initialization possibilities and variations, so it may need such braces to disambiguate? Note C does allow omitting most braces in initializer expressions - but not in this particular case of initializing anonymous unions with bare expressions.

Some pre-C11 GCC versions support some form of anonymous unions. They unfortunately require enclosing their designated initializers with braces which conflicts with this recommendation. This can be solved with an `#ifdef __STDC_VERSION__` as demonstrated in Zephyr commit c15f029a7108 and the corresponding code review.

2.7.3 Rust Language Support

Rust is a modern systems programming language designed to provide memory safety, concurrency, and performance without sacrificing low-level control. It achieves this through a unique ownership model that eliminates common bugs like null pointer dereferencing and data races at compile time.

Rust's emphasis on safety and correctness makes it particularly well-suited for embedded systems and environments where reliability is critical. Additionally, Rust offers powerful abstrac-

tions without a runtime or garbage collector, allowing developers to write both high-level code and low-level hardware interactions with confidence and efficiency.

These attributes make Rust a strong choice for projects on Zephyr, where resource constraints and system stability are paramount.

Enabling Rust Support

In order to enable Rust support in a Zephyr application, a few things need to be done:

1. As Rust is currently an optional module, the module needs to be enabled. The easiest way to do this is with west:

   ```
   west config manifest.project-filter +zephyr-lang-rust
   west update
   ```

   This should cause the Rust language support to be placed in `modules/lang/rust` in your Zephyr workspace.

2. Enable Rust support, via `CONFIG_RUST` in the `prj.conf`. The easiest way to do this (as well as the CMake setup from the next step) is to start with one of the samples in modules/lang/rust/samples.

3. Configure the application's `CMakeLists.txt` file to support Rust. Again this is easiest to copy from a sample, but this will look something like:

   ```
   cmake_minimum_required(VERSION 3.28.0)

   find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})

   project(my_app)
   rust_cargo_application()
   ```

4. Create a `Cargo.toml` that describes how to build the Rust application. From the Hello World sample:

   ```
   [package]
   # This must be rustapp for now.
   name = "rustapp"
   version = "0.1.0"
   edition = "2021"
   description = "The description of my app"
   license = "Apache-2.0 or MIT"

   [lib]
   crate-type = ["staticlib"]

   [dependencies]
   zephyr = "0.1.0"
   log = "0.4.22"
   ```

   The only required dependency is `zephyr` which provides the zephyr crate that is used to interface with Zephyr.

5. Build as you would any other Zephyr application. Only a few targets currently support Rust (these can be seen in the modules/lang/rust/etc/platforms.txt file).

API Documentation

The API Documentation for the latest version in the module is kept on gh-pages.

This documentation is generated for a general target, with all features enabled. Once you have an application that is buildable, you can generate documentation specifically for your target:

```
west build -t rustdoc

...

Generated /
↪my/path/app/zephyr/build/doc/rust/target/riscv32i-unknown-none-elf/doc/rustapp/index.html
```

The path printed at the end can be opened in a browser. This top level docs will be for your application itself. Look for the 'zephyr' crate on the left side bar, and this will take you to the docs for Zephyr. This page will also generate local docs for any dependencies used by your application, directly or indirectly.

2.8 Optimizations

Guides on how to optimize Zephyr for performance, power and footprint.

2.8.1 Optimizing for Footprint

Stack Sizes

Stack sizes of various system threads are specified generously to allow for usage in different scenarios on as many supported platforms as possible. You should start the optimization process by reviewing all stack sizes and adjusting them for your application:

`CONFIG_ISR_STACK_SIZE`
: Set to 2048 by default

`CONFIG_MAIN_STACK_SIZE`
: Set to 1024 by default

`CONFIG_IDLE_STACK_SIZE`
: Set to 320 by default

`CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE`
: Set to 1024 by default

`CONFIG_PRIVILEGED_STACK_SIZE`
: Set to 1024 by default, depends on userspace feature.

Unused Peripherals

Some peripherals are enabled by default. You can disable unused peripherals in your project configuration, for example:

```
CONFIG_GPIO=n
CONFIG_SPI=n
```

Various Debug/Informational Options

The following options output more information about the running application and provide means for debugging and error handling:

`CONFIG_BOOT_BANNER`
: This option can be disabled to save a few bytes.

`CONFIG_DEBUG`
: This option can be enabled for debug builds.

Note that the boot banner is enabled by default.

MPU/MMU Support

Depending on your application and platform needs, you can disable MPU/MMU support to gain some memory and improve performance. Consider the consequences of this configuration choice though, because you'll lose advanced stack checking and support.

2.8.2 Optimization Tools

The available optimization tools let you analyse *Footprint and Memory Usage* and *Data Structures* using different build system targets. An *HTML dashboard* can also be generated for a more dynamic view of build artifacts and metrics.

Footprint and Memory Usage

The build system offers 3 targets to view and analyse RAM, ROM and stack usage in generated images. The tools run on the final image and give information about size of symbols and code being used in both RAM and ROM. Additionally, with features available through the compiler, we can also generate worst-case stack usage analysis.

Some of the tools mentioned in this section are organizing their output based on the physical organization of the symbols. As some symbols might be external to the project's tree structure, or might lack metadata needed to display them by name, the following top-level containers are used to group such symbols:

- Hidden - The RAM and ROM reports list all processing symbols with no matching mapped files in the Hidden category.

  This means that the file for the listed symbol was not added to the metadata file, was empty, or was undefined. The tool was unable to get the name of the function for the given symbol nor identify where it comes from.

- No paths - The RAM and ROM reports list all processing symbols with relative paths in the No paths category.

  This means that the listed symbols cannot be placed in the tree structure of the report at an absolute path under one specific file. The tool was able to get the name of the function, but it was unable to identify where it comes from.

> **Note**
>
> You can have multiple cases of the same function, and the No paths category will list the sum of these in one entry.

Build Target: ram_report List all compiled objects and their RAM usage in a tabular form with bytes per symbol and the percentage it uses. The data is grouped based on the file system location of the object in the tree and the file containing the symbol.

Use the `ram_report` target with your board, as in the following example. If you are using *Sysbuild (System build)*, see *Dedicated image build targets* instead.

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t ram_report
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild ram_report
```

These commands will generate something similar to the output below:

```
Path                                                                  Size    %      ␣
↪Address
==========================================================================================
Root                                                                 4637 100.00%  -
├── (hidden)                                                            4   0.09%  -
├── (no paths)                                                       2748  59.26%  -
|   ├── _cpus_active                                                    4   0.09% ␣
↪0x20000314
|   ├── _kernel                                                        32   0.69% ␣
↪0x20000318
|   ├── _sw_isr_table                                                 384   8.28% ␣
↪0x00006474
|   ├── cli.1                                                          16   0.35% ␣
↪0x20000254
|   ├── on.2                                                            4   0.09% ␣
↪0x20000264
|   ├── poll_out_lock.0                                                 4   0.09% ␣
↪0x200002d4
|   ├── z_idle_threads                                                128   2.76% ␣
↪0x20000120
|   ├── z_interrupt_stacks                                          2048  44.17% ␣
↪0x20000360
|   └── z_main_thread                                                 128   2.76% ␣
↪0x200001a0
├── WORKSPACE                                                         184   3.97%  -
|   └── modules                                                       184   3.97%  -
|       └── hal                                                       184   3.97%  -
|           └── nordic                                                184   3.97%  -
|               └── nrfx                                              184   3.97%  -
|                   └── drivers                                       184   3.97%  -
|                       └── src                                       184   3.97%  -
|                           ├── nrfx_clock.c                            8   0.17%  -
|                           |   └── m_clock_cb                          8   0.17% ␣
↪0x200002e4
|                           ├── nrfx_gpiote.c                         132   2.85%  -
|                           |   └── m_cb                              132   2.85% ␣
↪0x20000060
|                           ├── nrfx_ppi.c                              4   0.09%  -
|                           |   └── m_channels_allocated                4   0.09% ␣
↪0x200000e4
|                           └── nrfx_twim.c                            40   0.86%  -
|                               └── m_cb                               40   0.86% ␣
↪0x200002ec
└── ZEPHYR_BASE                                                      1701  36.68%  -
    ├── arch                                                            5   0.11%  -
    |   └── arm                                                         5   0.11%  -
    |       └── core                                                    5   0.11%  -
    |           ├── mpu                                                 1   0.02%  -
```

(continues on next page)

(continued from previous page)

```
    |           |   └── arm_mpu.c                                  1   0.02%  -
    |           |       └── static_regions_num                     1   0.02% ␣
↪0x20000348
    |           └── tls.c                                          4   0.09%  -
    |               └── z_arm_tls_ptr                              4   0.09% ␣
↪0x20000240
    ├── drivers                                                   258   5.56%  -
    |   ├── ...                                                   ...    ...%
==========================================================================================
                                                                 4637
```

Build Target: rom_report List all compiled objects and their ROM usage in a tabular form with bytes per symbol and the percentage it uses. The data is grouped based on the file system location of the object in the tree and the file containing the symbol.

Use the `rom_report` target with your board, as in the following example. If you are using *Sysbuild (System build)*, see *Dedicated image build targets* instead.

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t rom_report
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild rom_report
```

These commands will generate something similar to the output below:

```
Path                                                            Size    %     ␣
↪Address
==========================================================================================
Root                                                           27828 100.00%  -
├── ...                                                          ...    ...%
└── ZEPHYR_BASE                                                13558  48.72%  -
    ├── arch                                                    1766   6.35%  -
    |   └── arm                                                 1766   6.35%  -
    |       └── core                                            1766   6.35%  -
    |           ├── cortex_m                                    1020   3.67%  -
    |           |   ├── fault.c                                  620   2.23%  -
    |           |   |   ├── bus_fault.constprop.0                108   0.39% ␣
↪0x00000749
    |           |   |   ├── mem_manage_fault.constprop.0         120   0.43% ␣
↪0x000007b5
    |           |   |   ├── usage_fault.constprop.0               84   0.30% ␣
↪0x000006f5
    |           |   |   ├── z_arm_fault                          292   1.05% ␣
↪0x0000082d
    |           |   |   └── z_arm_fault_init                      16   0.06% ␣
↪0x00000951
    |           |   ├── ...                                      ...    ...%
```

(continues on next page)

(continued from previous page)

```
    ├── boards                                                    32   0.11%  -
    │   └── arm                                                   32   0.11%  -
    │       └── reel_board                                        32   0.11%  -
    │           └── board.c                                       32   0.11%  -
    │               ├── __init_board_reel_board_init               8   0.03% ␣
↪0x000063e4
    │               └── board_reel_board_init                     24   0.09% ␣
↪0x00000ed5
    ├── build                                                    194   0.70%  -
    │   └── zephyr                                               194   0.70%  -
    │       ├── isr_tables.c                                     192   0.69%  -
    │       │   └── _irq_vector_table                             192   0.69% ␣
↪0x00000040
    │       └── misc                                               2   0.01%  -
    │           └── generated                                      2   0.01%  -
    │               └── configs.c                                  2   0.01%  -
    │                   └── _ConfigAbsSyms                         2   0.01% ␣
↪0x00005945
    ├── drivers                                                 6282  22.57%  -
    │   ├── ...                                                  ...    ...%
==================================================================================
                                                               21652
```

Build Targets: ram_plot/rom_plot Similar to the `ram_report` and `rom_report` build targets, these targets generate memory usage reports in a sunburst chart as a visual representation. A user can click on segments to navigate through the directory structures, and hover over segments to get more details.

Running the targets will first generate the CLI report and then open a browser window.

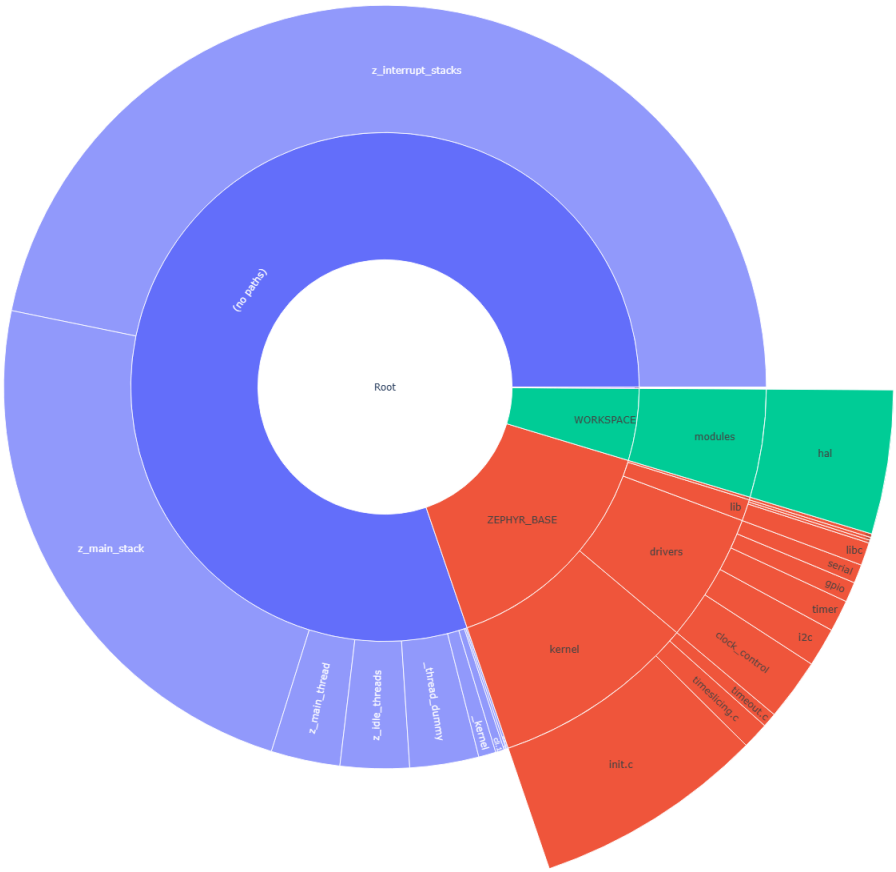
Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t ram_plot
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild ram_plot
```



And similarly for the ROM usage.

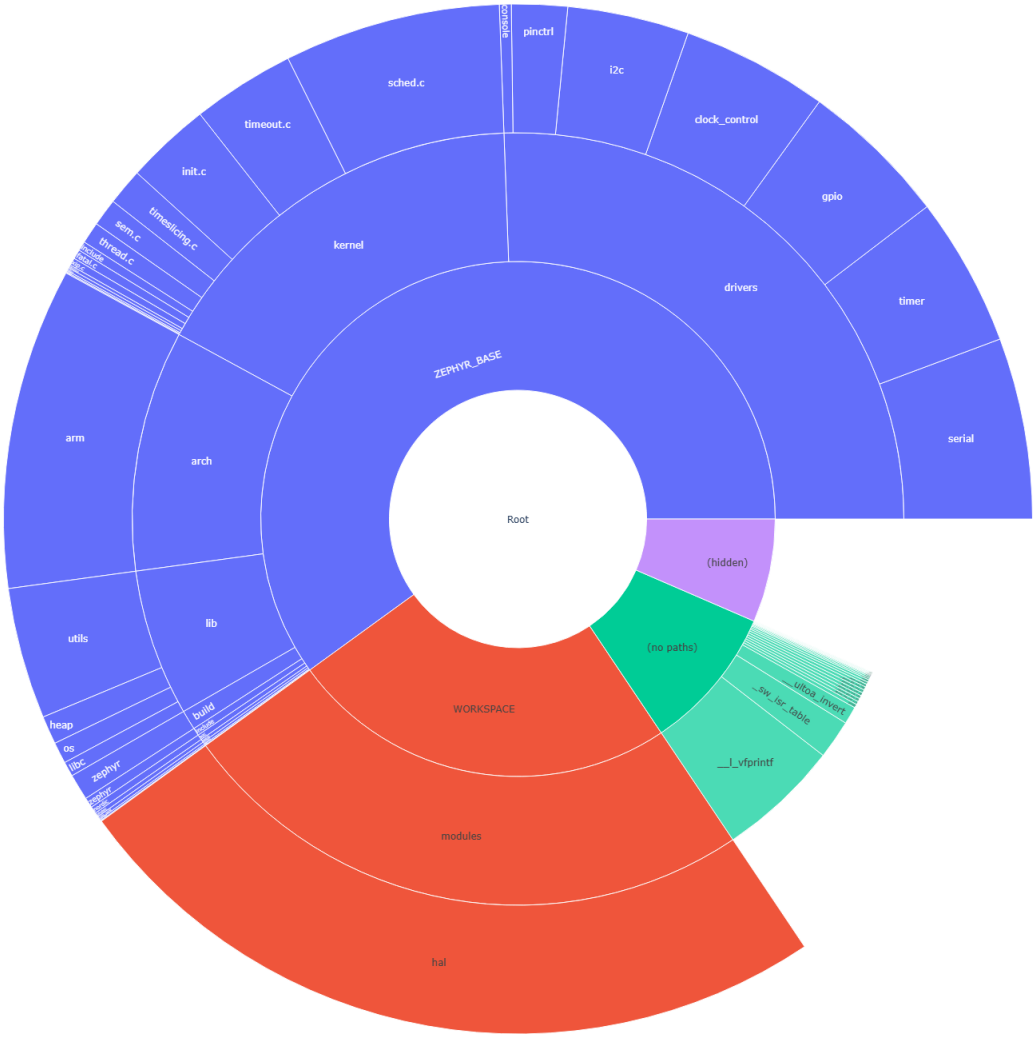
Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t rom_plot
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild rom_plot
```



Build Target: puncover This target uses a third-party tool called puncover which can be found at https://github.com/HBehrens/puncover. When this target is built, it will launch a local web server which will allow you to open a web client and browse the files and view their ROM, RAM, and stack usage.

Before you can use this target, install the puncover Python module:

```
pip3 install --user puncover
```

Warning

This is a third-party tool that might or might not be working at any given time. Please check the GitHub issues, and report new problems to the project maintainer.

After you installed the Python module, use `puncover` target with your board, as in the following example. If you are using *Sysbuild (System build)*, see *Dedicated image build targets* instead.

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t puncover
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

The `puncover` target will start a local web server on `localhost:5000` by default. The host IP and port the HTTP server runs on can be changed by setting the environment variables `PUNCOVER_HOST` and `PUNCOVER_PORT`. Further arguments to puncover can be provided by defining them in `EXTRA_PUNCOVER_ARGS`.

To view worst-case stack usage analysis in the interactive web app, build with the `CONFIG_STACK_USAGE` enabled.

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world -- -DCONFIG_STACK_USAGE=y -
↪DCONFIG_CALLGRAPH_INFO=y
west build -t puncover
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board -DCONFIG_STACK_USAGE=y -
↪DCONFIG_CALLGRAPH_INFO=y samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

If you want a report in a JSON file about the worst-case stack usages found, without launching the web app you can add the option `--generate-report --non-interactive --report-type json` for non-interactive use. And add all functions you are interested in the report by adding each as an argument in the form of `--report-max-static-stack-usage FUNCTION_NAME:::MAXIMAL_STACK_SIZE` for example to export `log_process_thread_func`, `shell_thread`, `bg_thread_main` and `work_queue_main` this would look like this:

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world -- -DCONFIG_STACK_USAGE=y -
↪DCONFIG_CALLGRAPH_INFO=y -DEXTRA_PUNCOVER_ARGS="--
↪generate-report;--non-interactive;--report-type;json;--report-max-static-stack-usage;log_proces
↪$PWD/report"
west build -t puncover
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board -DCONFIG_STACK_USAGE=y -
↪DCONFIG_CALLGRAPH_INFO=y -DEXTRA_PUNCOVER_ARGS="--
↪generate-report;--non-interactive;--report-type;json;--report-max-static-stack-usage;log_proces
↪$PWD/report" samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

All arguments can also for better maintainability and overview be put in a yaml config:

```yaml
elf_file: ~/zephyrproject/zephyr/samples/net/mqtt_publisher/build/zephyr/zephyr.elf
gcc-tools-base: ~/zephyr-sdk-1.0.1/gnu/arm-zephyr-eabi/bin/arm-zephyr-eabi-
src_root: ~/zephyrproject/zephyr
build_dir: ~/zephyrproject/zephyr/samples/net/mqtt_publisher/build
generate-report: true
non-interactive: true
report-type: json
report-max-static-stack-usage:
   - log_process_thread_func:::832
   - shell_thread:::4160
   - bg_thread_main:::2112
   - work_queue_main:::1088
   - mgmt_event_work_handler:::896
```

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world -- -DCONFIG_STACK_USAGE=y -
↪DCONFIG_CALLGRAPH_INFO=y -DEXTRA_PUNCOVER_ARGS="-c puncover_config.yaml"
west build -t puncover
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board -DCONFIG_STACK_USAGE=y -
↪DCONFIG_CALLGRAPH_INFO=y -DEXTRA_PUNCOVER_ARGS="-c puncover_config.yaml"␣
↪samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild puncover
```

Data Structures

Build Target: pahole Poke-a-hole (pahole) is an object-file analysis tool to find the size of the data structures, and the holes caused due to aligning the data elements to the word-size of the CPU by the compiler.

Poke-a-hole (pahole) must be installed prior to using this target. It can be obtained from https://git.kernel.org/pub/scm/devel/pahole/pahole.git and is available in the dwarves package in both fedora and ubuntu:

```
sudo apt-get install dwarves
```

Alternatively, you can get it from fedora:

```
sudo dnf install dwarves
```

After you installed the package, use `pahole` target with your board, as in the following example. If you are using *Sysbuild (System build)*, see *Dedicated image build targets* instead.

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t pahole
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild pahole
```

Pahole will generate something similar to the output below in the console:

```
/* Used at: [...]/build/zephyr/kobject_hash.c */
/* <375> [...]/zephyr/include/zephyr/sys/dlist.h:37 */
union {
        struct _dnode *            head;               /*     0     4 */
        struct _dnode *            next;               /*     0     4 */
};
/* Used at: [...]/build/zephyr/kobject_hash.c */
/* <397> [...]/zephyr/include/zephyr/sys/dlist.h:36 */
struct _dnode {
        union {
                struct _dnode *    head;                 /*     0     4 */
                struct _dnode *    next;                 /*     0     4 */
        };                                               /*     0     4 */
        union {
                struct _dnode *    tail;                 /*     4     4 */
                struct _dnode *    prev;                 /*     4     4 */
        };                                               /*     4     4 */

        /* size: 8, cachelines: 1, members: 2 */
        /* last cacheline: 8 bytes */
};
/* Used at: [...]/build/zephyr/kobject_hash.c */
/* <3b7> [...]/zephyr/include/zephyr/sys/dlist.h:41 */
union {
        struct _dnode *            tail;               /*     0     4 */
        struct _dnode *            prev;               /*     0     4 */
};
...
...
```

Dashboard

An HTML dashboard can be generated that consolidates the various tool outputs and artifacts into one simple view. In addition to a basic summary of the build results, the following details are included:

- Full memory reports (ram, rom) in drill-down table form as well as *plots* (as per the `footprint`, `ram_plot`, and `rom_plot` build targets).
- Kconfig symbol values and sources (as per the `traceconfig` build target).
- Init-levels with function names, and report on any priority issues with sys-init against devicetree (as per the `initlevels` build target).
- Navigable devicetree view with property values and details from any bindings.

Use the `dashboard` target with your board, as in the following example. If you are using *Sysbuild (System build)*, see *Dedicated image build targets* instead.

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t dashboard
```

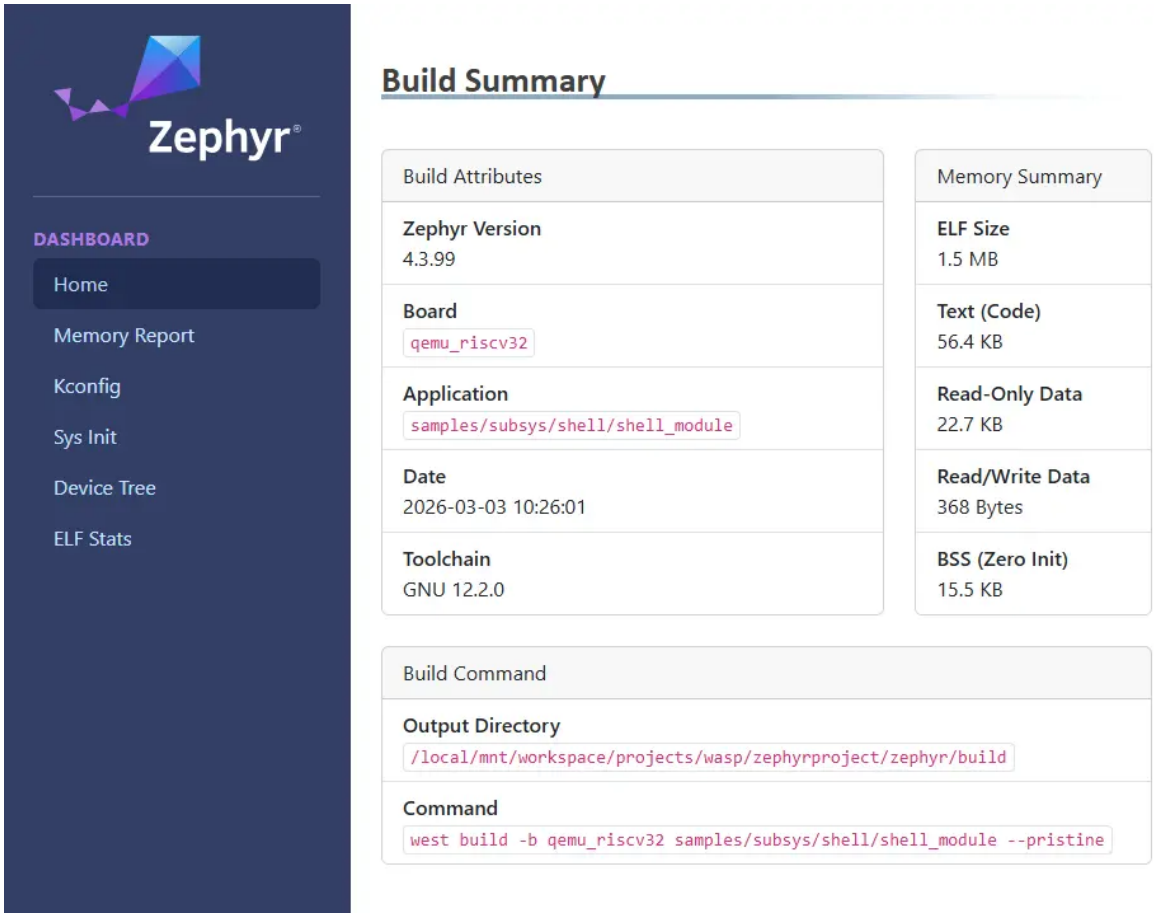
Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild dashboard
```

This will generate the following output file and open it in the default browser:

```
build/dashboard/index.html
```



2.9 Flashing and Hardware Debugging

2.9.1 Flash & Debug Host Tools

This guide describes the software tools you can run on your host workstation to flash and debug Zephyr applications.

Zephyr's west tool has built-in support for all of these in its `flash`, `debug`, `debugserver`, and `attach` commands, provided your board hardware supports them and your Zephyr board directory's `board.cmake` file declares that support properly. See *Building, Flashing and Debugging* for more information on these commands.

Black Magic Probe

Black Magic Probe (BMP) is an open-source debugging hardware incorporating GDB debug server functionality into the firmware. There is no need for a GDB server program, so there is no program equivalent to host-tool.

For more details, including usage instructions and supported targets, see *Black Magic Probe*.

SAM Boot Assistant (SAM-BA)

Atmel SAM Boot Assistant (Atmel SAM-BA) allows In-System Programming (ISP) from USB or UART host without any external programming interface. Zephyr allows users to develop and program boards with SAM-BA support using *west*. Zephyr supports devices with/without ROM bootloader and both extensions from Arduino and Adafruit. Full support was introduced in Zephyr SDK 0.12.0.

The typical command to flash the board is:

```
west flash [ -r bossac ] [ -p /dev/ttyX ] [ --erase ]
```

> **Note**
>
> By default, flashing with bossac will only erase the flash pages containing the flashed application, leaving other pages untouched. Should you wish to erase the entire flash of the target when flashing, pass the `--erase` parameter when flashing.

Flash configuration for devices:

With ROM bootloader

These devices don't need any special configuration. After building your application, just run `west flash` to flash the board.

Without ROM bootloader

For these devices, the user should:

1. Define flash partitions required to accommodate the bootloader and application image; see *Flash map* for details.
2. Have board `.defconfig` file with the `CONFIG_USE_DT_CODE_PARTITION` Kconfig option set to `y` to instruct the build system to use these partitions for code relocation. This option can also be set in `prj.conf` or any other Kconfig fragment.
3. Build and flash the SAM-BA bootloader on the device.

With compatible SAM-BA bootloader

For these devices, the user should:

1. Define flash partitions required to accommodate the bootloader and application image; see *Flash map* for details.
2. Have board `.defconfig` file with the `CONFIG_BOOTLOADER_BOSSA` Kconfig option set to `y`. This will automatically select the `CONFIG_USE_DT_CODE_PARTITION` Kconfig option which instruct the build system to use these partitions for code relocation. The board `.defconfig` file should have `CONFIG_BOOTLOADER_BOSSA_ARDUINO` , `CONFIG_BOOTLOADER_BOSSA_ADAFRUIT_UF2` or the `CONFIG_BOOTLOADER_BOSSA_LEGACY` Kconfig option set to `y` to select the right compatible SAM-BA bootloader mode. These options can also be set in `prj.conf` or any other Kconfig fragment.
3. Build and flash the SAM-BA bootloader on the device.

> **Note**
>
> The `CONFIG_BOOTLOADER_BOSSA_LEGACY` Kconfig option should be used as last resource. Try configure first with Devices without ROM bootloader.

Typical flash layout and configuration For bootloaders that reside on flash, the devicetree partition layout is mandatory. For devices that have a ROM bootloader, they are mandatory when the application uses a storage or other non-application partition. In this special case, the boot partition should be omitted and code_partition should start from offset 0. It is necessary to define the partitions with sizes that avoid overlaps, always.

A typical flash layout for devices without a ROM bootloader is:

```
/ {
        chosen {
                zephyr,code-partition = &code_partition;
        };
};

&flash0 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                boot_partition: partition@0 {
                        label = "sam-ba";
                        reg = <0x00000000 0x2000>;
                        read-only;
                };

                code_partition: partition@2000 {
                        label = "code";
                        reg = <0x2000 0x3a000>;
                        read-only;
                };

                /*
                * The final 16 KiB is reserved for the application.
                * Storage partition will be used by FCB/LittleFS/NVS
                * if enabled.
                */
                storage_partition: partition@3c000 {
                        label = "storage";
                        reg = <0x0003c000 0x00004000>;
                };
        };
};
```

A typical flash layout for devices with a ROM bootloader and storage partition is:

```
/ {
        chosen {
                zephyr,code-partition = &code_partition;
        };
};
```

(continues on next page)

(continued from previous page)

```
&flash0 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                code_partition: partition@0 {
                        label = "code";
                        reg = <0x0 0xF0000>;
                        read-only;
                };

                /*
                * The final 64 KiB is reserved for the application.
                * Storage partition will be used by FCB/LittleFS/NVS
                * if enabled.
                */
                storage_partition: partition@F0000 {
                        label = "storage";
                        reg = <0x000F0000 0x00100000>;
                };
        };
};
```

Enabling SAM-BA runner In order to instruct Zephyr west tool to use the SAM-BA bootloader the `board.cmake` file must have `include(${ZEPHYR_BASE}/boards/common/bossac.board.cmake)` entry. Note that Zephyr tool accept more entries to define multiple runners. By default, the first one will be selected when using `west flash` command. The remaining options are available passing the runner option, for instance `west flash -r bossac`.

More implementation details can be found in the boards documentation. As a quick reference, see these three board documentation pages:

- sam4e_xpro (ROM bootloader)
- adafruit_feather_m0_basic_proto (Adafruit UF2 bootloader)
- arduino_nano_33_iot (Arduino bootloader)
- arduino_nano_33_ble (Arduino legacy bootloader)

Enabling BOSSAC on Windows Native [Experimental] Zephyr SDK´s bossac is currently supported on Linux and macOS only. Windows support can be achieved by using the bossac version from BOSSA official releases. After installing using default options, the `bossac.exe` must be added to Windows PATH. A specific bossac executable can be used by passing the `--bossac` option, as follows:

```
west flash -r bossac --bossac="C:\Program Files (x86)\BOSSA\bossac.exe" --
↪bossac-port="COMx"
```

> **Note**
>
> WSL is not currently supported.

LinkServer Debug Host Tools

Linkserver is a utility for launching and managing GDB servers for NXP debug probes, which also provides a command-line target flash programming capabilities. Linkserver can be used with the NXP MCUXpresso for Visual Studio Code implementation, with custom debug configurations based on GNU tools or as part of a headless solution for continuous integration and test. LinkServer can be used with MCU-Link, LPC-Link2, LPC11U35-based and OpenSDA based standalone or on-board debug probes from NXP.

NXP recommends installing LinkServer by using NXP's MCUXpresso Installer. This method will also install the tools supporting the debug probes below, including NXP's MCU-Link and LPC-Scrypt tools.

LinkServer is compatible with the following debug probes:

- *LPC-LINK2 CMSIS DAP Onboard Debug Probe*
- *MCU-Link CMSIS-DAP Onboard Debug Probe*
- *OpenSDA DAPLink Onboard Debug Probe*

To use LinkServer with West commands, the install folder should be added to the *`PATH` environment variable*. The default installation path to add is:

Linux

```
/usr/local/LinkServer
```

macOS

```
/Applications/LinkServer_<version>
```

Windows

```
c:\nxp\LinkServer_<version>
```

Supported west commands:

1. flash
2. debug
3. debugserver
4. attach

Notes:

1. Probes can be listed with LinkServer:

```
LinkServer probes
```

2. With multiple debug probes attached to the host, use the LinkServer west runner `--probe` option to pass the probe index.

```
west flash --runner=linkserver --probe=3
```

3. Device-specific settings can be overridden with the west runner for LinkServer with the option '–override'. May be used multiple times. The format is dictated by LinkServer, e.g.:

```
west flash --runner=linkserver --override /
↪device/memory/5/flash-driver=MIMXRT500_SFDP_MXIC_OSPI_S.cfx
```

4. LinkServer does not install an implicit breakpoint at the reset handler. If you would like to single step from the start of their application, you will need to add a breakpoint at `main` or the reset handler manually.
5. That breakpoint, and any other GDB command, can be installed automatically with `--gdb-init`, which this runner appends to the GDB client for `west debug` and `west attach`, e.g.:

   ```
   west debug --runner=linkserver --gdb-init 'b main'
   ```

   See *Passing Extra GDB Commands*.

J-Link Debug Host Tools

Segger provides a suite of debug host tools for Linux, macOS, and Windows operating systems:

- J-Link GDB Server: GDB remote debugging
- J-Link Commander: Command-line control and flash programming
- RTT Viewer: RTT terminal input and output
- SystemView: Real-time event visualization and recording

These debug host tools are compatible with the following debug probes:

- *LPC-Link2 J-Link Onboard Debug Probe*
- *OpenSDA J-Link Onboard Debug Probe*
- *MCU-Link JLink Onboard Debug Probe*
- *J-Link External Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in J-Link Supported Devices.

Download and install the J-Link Software and Documentation Pack to get the J-Link GDB Server and Commander, and to install the associated USB device drivers. RTT Viewer and SystemView can be downloaded separately, but are not required.

Note that the J-Link GDB server does not yet support Zephyr RTOS-awareness.

Passing Extra GDB Commands The `jlink` runner accepts `--gdb-init`, which appends a command to the GDB client started by `west debug` and `west attach`. The option can be given multiple times and the commands run in the order given, last of everything the runner sends to GDB and before the target is resumed. Since they are only appended, they cannot change the runner's own connect, load and reset sequence. `west flash`, `west reset`, `west rtt` and `west debugserver` are unaffected.

For example, to enable J-Link semihosting for the duration of a debug session:

```
west debug -r jlink --gdb-init 'monitor semihosting enable' \
  --gdb-init 'monitor semihosting basedir .'
```

The same commands can be made the default for a board by adding them to its `board.cmake`:

```
board_runner_args(jlink "--gdb-init=monitor semihosting enable")
```

Note that `--gdb-init` only reaches the GDB client. Options for the J-Link GDB server itself are passed with `--tool-opt`, and J-Link Commander commands run while flashing are passed with `--pre-script-cmd`.

The `linkserver`, `openocd` and `intel_cyclonev` runners accept the same option. Which commands it applies to, and where the commands are inserted, is up to each runner.

OpenOCD Debug Host Tools

OpenOCD is a community open source project that provides GDB remote debugging and flash programming support for a wide range of SoCs. A fork that adds Zephyr RTOS-awareness is included in the Zephyr SDK; otherwise see Getting OpenOCD for options to download OpenOCD from official repositories.

These debug host tools are compatible with the following debug probes:

- *OpenSDA DAPLink Onboard Debug Probe*
- *J-Link External Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in OpenOCD Supported Devices.

Note

On Linux, openocd is available though the Zephyr SDK. Windows users should use the following steps to install openocd:

- Download openocd for Windows from here: OpenOCD Windows
- Copy bin and share dirs to `C:\Program Files\OpenOCD\`
- Add `C:\Program Files\OpenOCD\bin` to 'PATH' environment variable

pyOCD Debug Host Tools

pyOCD is an open source project from Arm that provides GDB remote debugging and flash programming support for Arm Cortex-M SoCs. It is distributed on PyPi and installed when you complete the *Get Zephyr and install Python dependencies* step in the Getting Started Guide. pyOCD includes support for Zephyr RTOS-awareness.

These debug host tools are compatible with the following debug probes:

- *LPC-LINK2 CMSIS DAP Onboard Debug Probe*
- *MCU-Link CMSIS-DAP Onboard Debug Probe*
- *OpenSDA DAPLink Onboard Debug Probe*
- *ST-LINK/V2-1 Onboard Debug Probe*

Check if your SoC is listed in pyOCD Supported Devices.

Lauterbach TRACE32 Debug Host Tools

Lauterbach TRACE32 is a product line of microprocessor development tools, debuggers and real-time tracer with support for JTAG, SWD, NEXUS or ETM over multiple core architectures, including Arm Cortex-A/-R/-M, RISC-V, Xtensa, etc. Zephyr allows users to develop and program boards with Lauterbach TRACE32 support using *west*.

The runner consists of a wrapper around TRACE32 software, and allows a Zephyr board to execute a custom start-up script (Practice Script) for the different commands supported, including the ability to pass extra arguments from CMake. Is up to the board using this runner to define the actions performed on each command.

Install Lauterbach TRACE32 Software Download Lauterbach TRACE32 software from the Lauterbach TRACE32 download website (registration required) and follow the installation steps described in Lauterbach TRACE32 Installation Guide.

Flashing and Debugging Set the *environment variable* `T32_DIR` to the TRACE32 system directory. Then execute `west flash` or `west debug` commands to flash or debug the Zephyr application as detailed in *Building, Flashing and Debugging*. The `debug` command launches TRACE32 GUI to allow debug the Zephyr application, while the `flash` command hides the GUI and perform all operations in the background.

By default, the `t32` runner will launch TRACE32 using the default configuration file named `config.t32` located in the TRACE32 system directory. To use a different configuration file, supply the argument `--config CONFIG` to the runner, for example:

```
west flash --config myconfig.t32
```

For more options, run `west flash --context -r t32` to print the usage.

Zephyr RTOS Awareness To enable Zephyr RTOS awareness follow the steps described in Lauterbach TRACE32 Zephyr OS Awareness Manual.

NXP S32 Debug Probe Host Tools

NXP S32 Debug Probe is designed to work in conjunction with NXP S32 Design Studio for S32 Platform.

Download (registration required) NXP S32 Design Studio for S32 Platform and follow the S32 Design Studio for S32 Platform Installation User Guide to get the necessary debug host tools and associated USB device drivers.

Note that Zephyr RTOS-awareness support for the NXP S32 GDB server depends on the target device. Consult the product release notes for more information.

Supported west commands:

1. debug
2. debugserver
3. attach

Basic usage Before starting, add NXP S32 Design Studio installation directory to the system *PATH environment variable*. Alternatively, it can be passed to the runner on each invocation via `--s32ds-path` as shown below:

Linux

```
west debug --s32ds-path=/opt/NXP/S32DS.3.6
```

Windows

```
west debug --s32ds-path=C:\NXP\S32DS.3.6
```

If multiple S32 debug probes are connected to the host via USB, the runner will ask the user to select one via command line prompt before continuing. The connection string for the probe can be also specified when invoking the runner via `--dev-id=<connection-string>`. Consult NXP S32 debug probe user manual for details on how to construct the connection string. For example, if using a probe with serial ID `00:04:9f:00:ca:fe`:

```
west debug --dev-id='s32dbg:00:04:9f:00:ca:fe'
```

It is possible to pass extra options to the debug host tools via `--tool-opt`. When executing `debug` or `attach` commands, the tool options will be passed to the GDB client only. When executing `debugserver`, the tool options will be passed to the GDB server. For example, to load a Zephyr application to SRAM and afterwards detach the debug session:

```
west debug --tool-opt='--batch'
```

Requirements

- **S32 Design Studio version**: 3.6.0 or newer.
- **S32DebugProbe OS (firmware)**: 1.1.0 or newer.

S32 Debug Probe OS Upgrade Procedure Refer to the "Reprogramming S32 Debug Probe Firmware Images" chapter in the S32 Debug Probe User Guide to upgrade the OS of the S32DebugProbe.

probe-rs Debug Host Tools

probe-rs is an open-source embedded toolkit written in Rust. It provides out-of-the-box support for a variety of debug probes, including CMSIS-DAP, ST-Link, SEGGER J-Link, FTDI and built-in USB-JTAG interface on ESP32 devices.

Check probe-rs Installation for more setup details.

Check if your SoC is listed in probe-rs Supported Devices.

Renesas Flash Programmer (RFP) Host Tools

Renesas provides Renesas Flash Programmer as an official programming tool for Renesas boards using the Renesas standard boot firmware. It is available as a GUI and CLI.

For boards configured with the `rfp` west runner, the RFP CLI can be easily used to flash Zephyr.

Supported west commands:

1. flash

Once downloaded, if `rfp-cli` is not placed somewhere in your system PATH, you can pass the location to `rfp-cli` when flashing:

```
west flash --rfp-cli ~/Downloads/RFP_CLI_Linux_V31800_x64/linux-x64/rfp-cli
```

STM32CubeCLT Flash & Debug Host Tools

STMicroelectronics provides STM32CubeCLT as an official all-in-one toolset compatible with Linux®, macOS® and Windows®, allowing the use of STMicroelectronics proprietary tools within third-party development environments.

It notably provides a GDB debugging server (the *ST-LINK GDB Server*) that can be used to debug applications on STM32 boards thanks to on-board or external ST-LINK debug probes.

It is compatible with the following debug probes:

- *ST-LINK/V2-1 Onboard Debug Probe*
- Standalone ST-LINK-V2, ST-LINK-V3, and STLINK-V3PWR probes

Install STM32CubeCLT The easiest way to get the ST-LINK GDB Server is to install STM32CubeCLT from STMicroelectronics' website. A valid email address is needed to receive the downloading link.

Basic usage The ST-Link GDB Server can be used through the `west attach`, `west debug` or `west debugserver` commands to debug Zephyr applications.

```
west debug --runner stlink_gdbserver
```

> **Note**
>
> The STM32CubeProgrammer version contained in the STM32CubeCLT installation can also be used to flash applications. To do so, the dedicated *STM32CubeProgrammer runner* should be used instead of `stlink_gdbserver`, as done in the following example:
>
> ```
> west flash --runner stm32cubeprogrammer
> ```

STM32CubeProgrammer Flash Host Tools

STMicroelectronics provides STM32CubeProgrammer (STM32CubeProg) as an official programming tool for STM32 boards on Linux®, macOS®, and Windows® operating systems.

It provides an easy-to-use and efficient environment for reading, writing, and verifying device memory through both the debug interface (JTAG and SWD) and the bootloader interface (UART and USB DFU, I2C, SPI, and CAN).

It offers a wide range of features to program STM32 internal memories (such as flash, RAM, and OTP) as well as external memories.

It also allows option programming and upload, programming content verification, and programming automation through scripting.

It is delivered in GUI (graphical user interface) and CLI (command-line interface) versions.

It is compatible with the following debug probes:

- *ST-LINK/V2-1 Onboard Debug Probe*
- *J-Link External Debug Probe*
- Standalone ST-LINK-V2, ST-LINK-V3, and STLINK-V3PWR probes

Install STM32CubeProgrammer The easiest way to get STM32CubeProgrammer is to download it from STMicroelectronics website. A valid email address is needed to receive the downloading link.

Alternatively, it can be installed as part of STM32CubeCLT all-in-one multi-OS command-line toolset which also includes GDB debugger client and server.

If you have STM32CubeIDE installed on your system, then STM32CubeProg is already present.

Basic usage STM32CubeProgrammer is setup as the default west runner for all active STM32 boards supported by Zephyr. It can be used through the `west flash` command to flash Zephyr applications.

```
west flash --runner stm32cubeprogrammer
```

For advanced usage via the GUI or CLI, check out the STM32CubeProgrammer User Manual.

XSDB Flash & Debug Host Tools

The AMD XSDB utility (Xilinx Software Command-line Tool for Debug) is the command-line tool used to program and debug many AMD adaptive SoC and FPGA platforms. It is **not** included in the Zephyr SDK: install AMD Vitis (or the equivalent AMD toolchain distribution for your platform) and ensure the `xsdb` executable is on your system *PATH*.

Boards that select the `xsdb` west runner normally ship a board-specific `support/xsdb.cfg` next to the board definition. See your board's documentation for required boot artifacts (PDI, bitstream, FSBL, etc.).

Supported west commands include `flash`, `debug`, and `debugserver`.

For this runner, `west debug` and `west debugserver` both start the same native XSDB interactive session. Unlike GDB-based runners (where `debugserver` starts a remote stub for an IDE), the xsdb runner always launches XSDB directly, loads the application through the board `xsdb.cfg`, and leaves you at the XSDB prompt.

```
west flash --runner xsdb

west debug --runner xsdb

west debugserver --runner xsdb
```

Note

This is the same class of dependency as other proprietary host tools in this chapter (for example *J-Link* or *STM32CubeCLT*): Zephyr integrates with the tool via the west runner; obtaining and licensing the toolchain is the user's responsibility.

UF2 Uploader

The uf2 runner supports flashing some boards using the UF2 (USB Flashing Format). UF2 is a user-friendly file format designed for drag-and-drop programming via a USB mass storage device.

It relies on the target device entering a special bootloader mode where it appears to the host as a USB mass storage device. Once in this mode, the application image can be uploaded by copying a `.uf2` file to the mounted volume.

```
west flash --runner uf2
```

If the UF2 volume is not automatically detected, you may need to manually specify the mount point using the `--device` option:

For more about the UF2 format and its tooling, see USB Flashing Format (UF2).

Realtek Alternate Flash Programmer (rtkprog)

`rtkprog` is an open source implementation of the protocol needed to flash Realtek Bee family SoCs via UART.

```
west flash --runner rtkprog
```

Realtek Bee Flash Programmer (MPCli) Host Tools

Realtek provides Realtek Flash Programmer (MPCli) as an official programming tool for the Bee series, supporting Linux, macOS, and Windows operating systems. MPCli enables In-System Programming (ISP) via a UART interface, eliminating the need for external programming hardware. On most official Bee-series evaluation boards, a built-in UART-to-USB converter is included for ready-to-use programming and logging.

After downloading the archive of MPCli, extract it and choose the version compatible with your operating system. Then, add the directory containing the `mpcli` executable to your system *PATH environment variable*.

Before starting, ensure your board is in download mode. Please refer to the board documentation at: Realtek Supported Boards

```
west flash [--runner mpcli] --port /dev/ttyX
```

IAR EW & C-Spy Host Tools

IAR provides Embedded Workbench and CSpyBat for debugging and flashing. The iar runner works with EWARM 10.10 or newer.

2.9.2 Debug Probes

A *debug probe* is special hardware which allows you to control execution of a Zephyr application running on a separate board. Debug probes usually allow reading and writing registers and memory, and support breakpoint debugging of the Zephyr application on your host workstation using tools like GDB. They may also support other debug software and more advanced features such as *tracing program execution*. For details on the related host software supported by Zephyr, see *Flash & Debug Host Tools*.

Debug probes are usually connected to your host workstation via USB; they are sometimes also accessible via an IP network or other means. They usually connect to the device running Zephyr using the JTAG or SWD protocols. Debug probes are either separate hardware devices or circuitry integrated into the same board which runs Zephyr.

Many supported boards in Zephyr include a second microcontroller that serves as an onboard debug probe, usb-to-serial adapter, and sometimes a drag-and-drop flash programmer. This eliminates the need to purchase an external debug probe and provides a variety of debug host tool options.

Several hardware vendors have their own branded onboard debug probe implementations: NXP boards may use *OpenSDA*, *LPC-Link2*, or *MCU-Link*, probes depending on the microcontroller the debug probe firmware runs on. ST boards have the *ST-LINK probe*. Each onboard debug probe microcontroller can support one or more types of firmware that communicate with their respective debug host tools. For example, an OpenSDA microcontroller can be programmed with DAPLink firmware to communicate with pyOCD or OpenOCD debug host tools, or with J-Link firmware to communicate with J-Link debug host tools.

Debug Probes & Host Tools Compatibility Chart		Host Tools					
		J-Link Debug	**OpenO**	**pyOCD**	**NXP S32DS**	**NXP LinkServe**	**ST-LINK GDB Server**
Debug Probes	**J-Link External**	✓	✓				
	LPC-Link2 CMSIS-DAP					✓	
	LPC-Link2 J-Link	✓					
	MCU-Link CMSIS-DAP					✓	
	MCU-Link J-Link	✓					
	NXP S32 Debug Probe				✓		
	OpenSDA DAPLink		✓	✓		✓	
	OpenSDA J-Link	✓					
	ST-LINK/V2-1	✓	✓	*some STM32 boards*			✓

Some supported boards in Zephyr do not include an onboard debug probe and therefore require an external debug probe. In addition, boards that do include an onboard debug probe often also have an SWD or JTAG header to enable the use of an external debug probe instead. One reason this may be useful is that the onboard debug probe may have limitations, such as lack of support for advanced debuggers or high-speed tracing. You may need to adjust jumpers to prevent the onboard debug probe from interfering with the external debug probe.

NXP Onboard Debug Probes

NXP boards may have one of several onboard debug probes. These probes include the *MCU-Link Onboard Debug Probe*, *LPC-LINK2 Onboard Debug Probe* and *OpenSDA Onboard Debug Probe*. Each of these probes is implemented as a secondary microcontroller present on the evaluation board. The specific debug probe type present on a given board can be determined based on the debug microcontroller SOC:

- LPC55S69: *MCU-Link Onboard Debug Probe*
- LPC4322: *LPC-LINK2 Onboard Debug Probe*
- MK20: *OpenSDA Onboard Debug Probe*

For example, the frdm_k64f board has an MK20 debug microcontroller, so this board uses the *OpenSDA Onboard Debug Probe*.

MCU-Link Onboard Debug Probe

The MCU-Link onboard debug probe uses an LPC55S69 SOC. This probe supports the following firmwares:

- *MCU-Link CMSIS-DAP Onboard Debug Probe* (default firmware)
- *MCU-Link JLink Onboard Debug Probe*

This probe is programmed using the MCU-Link host tools, which are installed with the *LinkServer Debug Host Tools*. NXP recommends using NXP's MCUXpresso Installer to install the Linkserver tools.

MCU-Link CMSIS-DAP Onboard Debug Probe This is the default firmware installed on MCU-Link debug probes. The CMSIS-DAP debug probes allow debugging from any compatible toolchain, including IAR EWARM, Keil MDK, NXP's MCUXpresso IDE and MCUXpresso extension for VS Code. In addition to debug probe functionality, the MCU-Link probes may also provide:

1. SWO trace end point: this virtual device is used by MCUXpresso to retrieve SWO trace data. See the MCUXpresso IDE documentation for more information.
2. Virtual COM (VCOM) port / UART bridge connected to the target processor
3. USB to UART, SPI and/or I2C interfaces (depending on MCU-Link type/implementation)
4. Energy measurements of the target MCU

This debug probe is compatible with the following debug host tools:

- *LinkServer Debug Host Tools*

Once the MCU-Link host tools are installed, the following steps are required to program the CMSIS-DAP firmware:

1. Make sure the MCU-Link utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the MCU-Link microcontroller into DFU boot mode by attaching the DFU jumper then connecting to the USB debug port on the board. This jumper may also be referred to as the ISP jumper, and will be connected to `PIO0_5` on the LPC55S69.
3. Run the `program_CMSIS` script, found in the installed MCU-Link `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

MCU-Link JLink Onboard Debug Probe This debug probe firmware provides a JLink compatible debug interface, as well as a USB-Serial adapter. It is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

These probes do not have JLink firmware installed by default, and must be updated. Once the MCU-Link host tools are installed, the following steps are required to program the JLink firmware:

1. Make sure the MCU-Link utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the MCU-Link microcontroller into DFU boot mode by attaching the DFU jumper then connecting to the USB debug port on the board. This jumper may also be referred to as the ISP jumper, and will be connected to `PIO0_5` on the LPC55S69.
3. Run the `program_JLINK` script, found in the installed MCU-Link `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

LPC-LINK2 Onboard Debug Probe

The LPC-LINK2 onboard debug probe uses an LPC4322 SOC. This probe supports the following firmwares:

- *LPC-LINK2 CMSIS DAP Onboard Debug Probe*
- *LPC-Link2 J-Link Onboard Debug Probe*
- *LPC-Link2 DAPLink Onboard Debug Probe* (default firmware)

This probe is programmed using the LPCScrypt host tools, which are installed with the *LinkServer Debug Host Tools*. NXP recommends using NXP's MCUXpresso Installer to install the Linkserver tools.

LPC-LINK2 CMSIS DAP Onboard Debug Probe The CMSIS-DAP debug probes allow debugging from any compatible toolchain, including IAR EWARM, Keil MDK, as well as NXP's MCUXpresso IDE and MCUXpresso extension for VS Code. As well as providing debug probe functionality, the LPC-Link2 probes also provide:

1. SWO trace end point: this virtual device is used by MCUXpresso to retrieve SWO trace data. See the MCUXpresso IDE documentation for more information.
2. Virtual COM (VCOM) port / UART bridge connected to the target processor
3. LPCSIO bridge that provides communication to I2C and SPI peripheral devices

This debug probe firmware is compatible with the following debug host tools:

- *LinkServer Debug Host Tools*

The probe may be updated to use CMSIS-DAP firmware with the following steps:

1. Make sure the LPCScrypt utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the LPC-Link2 microcontroller into DFU boot mode by attaching the DFU jumper, then connecting to the USB debug port on the board. This jumper is connected to `P2_6` on the LPC4322 SOC.
3. Run the `program_CMSIS` script, found in the installed LPCScrypt `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

> **Note**
>
> On some boards, the J-Link probe firmware will no longer power the board via the USB debug port. On these boards, an alternative method of powering the board must be used when this firmware is programmed.

LPC-Link2 J-Link Onboard Debug Probe This debug probe firmware provides a JLink compatible debug interface, as well as a USB-Serial adapter. It is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

The probe may be updated to use the J-Link firmware with the following steps:

> **Note**
>
> Verify the firmware supports your board by visiting Firmware for LPCXpresso

1. Make sure the LPCScrypt utility is present on your host machine. This can be done by installing *LinkServer Debug Host Tools*.
2. Put the LPC-Link2 microcontroller into DFU boot mode by attaching the DFU jumper, then connecting to the USB debug port on the board. This jumper is connected to `P2_6` on the LPC4322 SOC.
3. Run the `program_JLINK` script, found in the installed LPCScrypt `scripts` folder.
4. Remove the DFU jumper and power cycle the board.

LPC-Link2 DAPLink Onboard Debug Probe The LPC-Link2 DAPLink firmware is the default firmware shipped on LPC-Link2 based boards, but is not the recommended firmware. Users should update to the *LPC-LINK2 CMSIS DAP Onboard Debug Probe* firmware following the instructions provided above. For details on programming the DAPLink firmware, see NXP AN13206.

OpenSDA Onboard Debug Probe

The OpenSDA onboard debug probe is based on the NXP MK20 SOC. It features drag and drop programming supports, and supports the following debug firmwares:

- *OpenSDA DAPLink Onboard Debug Probe* (default firmware)
- *OpenSDA J-Link Onboard Debug Probe*

OpenSDA DAPLink Onboard Debug Probe This debug probe firmware is compatible with the following debug host tools:

- *pyOCD Debug Host Tools*
- *OpenOCD Debug Host Tools*
- *LinkServer Debug Host Tools*

This probe is realized by programming the OpenSDA microcontroller with DAPLink OpenSDA firmware. NXP provides OpenSDA DAPLink Board-Specific Firmwares.

Install the debug host tools before you program the firmware.

As with all OpenSDA debug probes, the steps for programming the firmware are:

1. Put the OpenSDA microcontroller into bootloader mode by holding the reset button while you power on the board. Note that "bootloader mode" in this context applies to the OpenSDA microcontroller itself, not the target microcontroller of your Zephyr application.
2. After you power on the board, release the reset button. A USB mass storage device called **BOOTLOADER** or **MAINTENANCE** will enumerate. If the enumerated device is named **BOOTLOADER**, please first update the bootloader to the latest revision by following the instructions for a DAPLink Bootloader Update.
3. Copy the OpenSDA firmware binary to the USB mass storage device.
4. Power cycle the board, this time without holding the reset button. You should see three USB devices enumerate: a CDC device (serial port), a HID device (debug port), and a mass storage device (drag-and-drop flash programming).

OpenSDA J-Link Onboard Debug Probe This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*

This probe is realized by programming the OpenSDA microcontroller with J-Link OpenSDA firmware. Segger provides OpenSDA J-Link Generic Firmwares and OpenSDA J-Link Board-Specific Firmwares, where the latter is generally recommended when available. Board-specific firmwares are required for i.MX RT boards to support their external flash memories, whereas generic firmwares are compatible with all Kinetis boards.

Install the debug host tools before you program the firmware.

As with all OpenSDA debug probes, the steps for programming the firmware are:

1. Put the OpenSDA microcontroller into bootloader mode by holding the reset button while you plug a USB into the board's USB debug port. Note that "bootloader mode" in this context applies to the OpenSDA microcontroller itself, not the target microcontroller of your Zephyr application.

2. After you power on the board, release the reset button. A USB mass storage device called **BOOTLOADER** or **MAINTENANCE** will enumerate. If the enumerated device is named **BOOTLOADER**, please first update the bootloader to the latest revision by following the instructions for a DAPLink Bootloader Update.
3. Copy the OpenSDA firmware binary to the USB mass storage device.
4. Power cycle the board, this time without holding the reset button. You should see two USB devices enumerate: a CDC device (serial port) and a vendor-specific device (debug port).

J-Link External Debug Probe

Segger J-Link is a family of external debug probes, including J-Link EDU, J-Link PLUS, J-Link ULTRA+, and J-Link PRO, that support a large number of devices from different hardware architectures and vendors.

This debug probe is compatible with the following debug host tools:

- *J-Link Debug Host Tools*
- *OpenOCD Debug Host Tools*

Install the debug host tools before you program the firmware.

ST-LINK/V2-1 Onboard Debug Probe

ST-LINK/V2-1 is a serial and debug adapter built into all Nucleo and Discovery boards. It provides a bridge between your computer (or other USB host) and the embedded target processor, which can be used for debugging, flash programming, and serial communication, all over a simple USB cable.

It is compatible with the following host debug tools:

- *OpenOCD Debug Host Tools*
- *J-Link Debug Host Tools*
- *STM32CubeCLT Flash & Debug Host Tools*

For some STM32 based boards, it is also compatible with:

- *pyOCD Debug Host Tools*

While it works out of the box with OpenOCD, it requires some flashing to work with J-Link. To do this, SEGGER offers a firmware upgrading the ST-LINK/V2-1 on board on the Nucleo and Discovery boards. This firmware makes the ST-LINK/V2-1 compatible with J-LinkOB, allowing users to take advantage of most J-Link features like the ultra fast flash download and debugging speed or the free-to-use GDBServer.

More information about upgrading ST-LINK/V2-1 to JLink or restore ST-Link/V2-1 firmware please visit: Segger over ST-Link

Flash and debug with ST-Link Using OpenOCD

OpenOCD is available by default on ST-Link and configured as the default flash and debug tool. Flash and debug can be done as follows:

```
# From the root of the zephyr repository
west build -b None samples/hello_world
west flash
```

```
# From the root of the zephyr repository
west build -b None samples/hello_world
west debug
```

Using Segger J-Link

Once STLink is flashed with SEGGER FW and J-Link GDB server is installed on your host computer, you can flash and debug as follows:

Use CMake with `-DBOARD_FLASH_RUNNER=jlink` to change the default OpenOCD runner to J-Link. Alternatively, you might add the following line to your application `CMakeList.txt` file.

```
set(BOARD_FLASH_RUNNER jlink)
```

If you use West (Zephyr's meta-tool) you can modify the default runner using the `--runner` (or `-r`) option.

```
west flash --runner jlink
```

To attach a debugger to your board and open up a debug console with `jlink`.

```
west debug --runner jlink
```

For more information about West and available options, see *West (Zephyr's meta-tool)*.

If you configured your Zephyr application to use Segger RTT console instead, open telnet:

```
$ telnet localhost 19021
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.30f - Real time terminal output
J-Link STLink V21 compiled Jun 26 2017 10:35:16 V1.0, SN=773895351
Process: JLinkGDBServerCLExe
Zephyr Shell, Zephyr version: 1.12.99
Type 'help' for a list of available commands
shell>
```

If you get no RTT output you might need to disable other consoles which conflict with the RTT one if they are enabled by default in the particular sample or application you are running, such as disable UART_CONSOLE in menuconfig

Updating or restoring ST-Link firmware ST-Link firmware can be updated using STM32CubeProgrammer Tool. It is usually useful when facing flashing issues, for instance when using twister's device-testing option.

Once installed, you can update attached board ST-Link firmware with the following command

```
s java -jar ~/
↪STMicroelectronics/STM32Cube/STM32CubeProgrammer/Drivers/FirmwareUpgrade/STLinkUpgrade.jar
↪-sn <board_uid>
```

Where board_uid can be obtained using twister's generate-hardware-map option. For more information about twister and available options, see *Test Runner (Twister)*.

OpenOCD Deprecates HLA ST-Link Interface OpenOCD has deprecated the legacy HLA interface used by ST-Link firmware in favor to the generic DAP interface in January 2024 (see OpenOCD deprecates ST-Link HLA Interface) in favor of the generic DAP interface. DAP interface is supported by ST-Link firmware since 2015 (from version v2j24).

Since OpenOCD deprecation, one using a recent OpenOCD tool (after release tag v0.12.0) may experience communication issues when the ST-Link adapter embeds a firmware prior version v2j24. In such case, it is recommended to update the ST-Link firmware (refer to *ST-LINK firmware*

update). If it is not possible to update the firmware, one may still use the old HLA interface by modifying its OpenOCD configuration script replacing (if applicable) the 2 below lines:

```
source [find interface/stlink-dap.cfg]
transport select dapdirect_swd
```

with the 2 following lines:

```
source [find interface/stlink-hla.cfg]
transport select hla_swd
```

NXP S32 Debug Probe

NXP S32 Debug Probe enables NXP S32 target system debugging via a standard debug port while connected to a developer's workstation via USB or remotely via Ethernet.

NXP S32 Debug Probe is designed to work in conjunction with NXP S32 Design Studio (S32DS) and NXP Automotive microcontrollers and processors. Install the debug host tools as in indicated in *NXP S32 Debug Probe Host Tools* before you program the firmware.

Black Magic Probe

The Black Magic Probe is an open-source hardware for debugging that is designed to be used with Black Magic Debug firmware. The firmware incorporates GDB Server so that you can connect directly from gdb to the target device.

Some of the STM32F103-based boards can run the Black Magic Debug firmware. See Black Magic Debug supported hardware.

2.10 Modules (External projects)

Zephyr relies on the source code of several externally maintained projects in order to avoid reinventing the wheel and to reuse as much well-established, mature code as possible when it makes sense. In the context of Zephyr's build system those are called *modules*. These modules must be integrated with the Zephyr build system, as described in more detail in other sections on this page.

To be classified as a candidate for being included in the default list of modules, an external project is required to have its own life-cycle outside the Zephyr Project, that is, reside in its own repository, and have its own contribution and maintenance workflow and release process. Zephyr modules should not contain code that is written exclusively for Zephyr. Instead, such code should be contributed to the main zephyr tree.

Modules to be included in the default manifest of the Zephyr project need to provide functionality or features endorsed and approved by the project Technical Steering Committee and should comply with the *module licensing requirements* and *contribution guidelines*. They should also have a Zephyr developer that is committed to maintain the module codebase.

Zephyr depends on several categories of modules, including but not limited to:

- Debugger integration
- Silicon vendor Hardware Abstraction Layers (HALs)
- Cryptography libraries
- File Systems
- Inter-Process Communication (IPC) libraries

Additionally, in some cases modules (particularly vendor HALs) can contain references to optional *binary blobs*.

This page summarizes a list of policies and best practices which aim at better organizing the workflow in Zephyr modules.

2.10.1 Modules vs west projects

Zephyr modules, described in this page, are not the same as *west projects*. In fact, modules *do not require west* at all. However, when using modules *with west*, then the build system uses west in order to find modules.

In summary:

Modules are repositories that contain a `zephyr/module.yml` file, so that the Zephyr build system can pull in the source code from the repository. *West projects* are entries in the `projects:` section in the `west.yml` manifest file. West projects are often also modules, but not always. There are west projects that are not included in the final firmware image (eg. tools) and thus do not need to be modules. Modules are found by the Zephyr build system either via *west itself*, or via the *ZEPHYR_MODULES CMake variable*.

The contents of this page only apply to modules, and not to west projects in general (unless they are a module themselves).

2.10.2 Module Repositories

- All modules included in the default manifest shall be hosted in repositories under the zephyrproject-rtos GitHub organization.
- The module repository codebase shall include a *module.yml* file in a `zephyr/` folder at the root of the repository.
- Module repository names should follow the convention of using lowercase letters and dashes instead of underscores. This rule will apply to all new module repositories, except for repositories that are directly tracking external projects (hosted in Git repositories); such modules may be named as their external project counterparts.

  > **Note**
  >
  > Existing module repositories that do not conform to the above convention do not need to be renamed to comply with the above convention.

- Module repositories names should be explicitly set in the `zephyr/module.yml` file.
- Modules should use "zephyr" as the default name for the repository main branch. Branches for specific purposes, for example, a module branch for an LTS Zephyr version, shall have names starting with the 'zephyr_' prefix.
- If the module has an external (upstream) project repository, the module repository should preserve the upstream repository folder structure.

  > **Note**
  >
  > It is not required in module repositories to maintain a 'master' branch mirroring the master branch of the external repository. It is not recommended as this may generate confusion around the module's main branch, which should be 'zephyr'.

- Modules should expose all provided header files with an include pathname beginning with the module-name. (E.g., mcuboot should expose its `bootutil/bootutil.h` as "mcuboot/bootutil/bootutil.h".)

Synchronizing with upstream

It is preferred to synchronize a module repository with the latest stable release of the corresponding external project. It is permitted, however, to update a Zephyr module repository with the latest development branch tip, if this is required to get important updates in the module codebase. When synchronizing a module with upstream it is mandatory to document the rationale for performing the particular update.

Requirements for allowed practices Changes to the main branch of a module repository, including synchronization with upstream code base, may only be applied via pull requests. These pull requests shall be *verifiable* by Zephyr CI and *mergeable* (e.g. with the *Rebase and merge*, or *Create a merge commit* option using Github UI). This ensures that the incoming changes are always **reviewable**, and the *downstream* module repository history is incremental (that is, existing commits, tags, etc. are always preserved). This policy also allows to run Zephyr CI, git lint, identity, and license checks directly on the set of changes that are to be brought into the module repository.

> **Note**
>
> Force-pushing to a module's main branch is not allowed.

Allowed practices The following practices conform to the above requirements and should be followed in all modules repositories. It is up to the module code owner to select the preferred synchronization practice, however, it is required that the selected practice is consistently followed in the respective module repository.

Updating modules with a diff from upstream: Upstream changes brought as a single *snapshot* commit (manual diff) in a pull request against the module's main branch, which may be merged using the *Rebase & merge* operation. This approach is simple and should be applicable to all modules with the downside of suppressing the upstream history in the module repository.

> **Note**
>
> The above practice is the only allowed practice in modules where the external project is not hosted in an upstream Git repository.

The commit message is expected to identify the upstream project URL, the version to which the module is updated (upstream version, tag, commit SHA, if applicable, etc.), and the reason for the doing the update.

Updating modules by merging the upstream branch: Upstream changes brought in by performing a Git merge of the intended upstream branch (e.g. main branch, latest release branch, etc.) submitting the result in pull request against the module main branch, and merging the pull request using the *Create a merge commit* operation. This approach is applicable to modules with an upstream project Git repository. The main advantages of this approach is that the upstream repository history (that is, the original commit SHAs) is preserved in the module repository. The downside of this approach is that two additional merge commits are generated in the downstream main branch.

2.10.3 Contributing to Zephyr modules

Individual Roles & Responsibilities

To facilitate management of Zephyr module repositories, the following individual roles are defined.

Administrator: Each Zephyr module shall have an administrator who is responsible for managing access to the module repository, for example, for adding individuals as Collaborators in the repository at the request of the module owner. Module administrators are members of the Administrators team, that is a group of project members with admin rights to module GitHub repositories.

Module owner: Each module shall have a module code owner. Module owners will have the overall responsibility of the contents of a Zephyr module repository. In particular, a module owner will:

- coordinate code reviewing in the module repository
- be the default assignee in pull-requests against the repository's main branch
- request additional collaborators to be added to the repository, as they see fit
- regularly synchronize the module repository with its upstream counterpart following the policies described in *Synchronizing with upstream*
- be aware of security vulnerability issues in the external project and update the module repository to include security fixes, as soon as the fixes are available in the upstream code base
- list any known security vulnerability issues, present in the module codebase, in Zephyr release notes.

> **Note**
>
> Module owners are not required to be Zephyr *Maintainers*.

Merger: The Zephyr Release Engineering team has the right and the responsibility to merge approved pull requests in the main branch of a module repository.

Maintaining the module codebase

Updates in the zephyr main tree, for example, in public Zephyr APIs, may require patching a module's codebase. The responsibility for keeping the module codebase up to date is shared between the **contributor** of such updates in Zephyr and the module **owner**. In particular:

- the contributor of the original changes in Zephyr is required to submit the corresponding changes that are required in module repositories, to ensure that Zephyr CI on the pull request with the original changes, as well as the module integration testing are successful.
- the module owner has the overall responsibility for synchronizing and testing the module codebase with the zephyr main tree. This includes occasional advanced testing of the module's codebase in addition to the testing performed by Zephyr's CI. The module owner is required to fix issues in the module's codebase that have not been caught by Zephyr pull request CI runs.

Contributing changes to modules

Submitting and merging changes directly to a module's codebase, that is, before they have been merged in the corresponding external project repository, should be limited to:

- changes required due to updates in the zephyr main tree
- urgent changes that should not wait to be merged in the external project first, such as fixes to security vulnerabilities.

Non-trivial changes to a module's codebase, including changes in the module design or functionality should be discouraged, if the module has an upstream project repository. In that case, such changes shall be submitted to the upstream project, directly.

Submitting changes to modules describes in detail the process of contributing changes to module repositories.

Contribution guidelines Contributing to Zephyr modules shall follow the generic project *Contribution guidelines*.

Pull Requests: may be merged with minimum of 2 approvals, including an approval by the PR assignee. In addition to this, pull requests in module repositories may only be merged if the introduced changes are verified with Zephyr CI tools, as described in more detail in other sections on this page.

The merging of pull requests in the main branch of a module repository must be coupled with the corresponding manifest file update in the zephyr main tree.

Issue Reporting: GitHub issues are intentionally disabled in module repositories, in favor of a centralized policy for issue reporting. Tickets concerning, for example, bugs or enhancements in modules shall be opened in the main zephyr repository. Issues should be appropriately labeled using GitHub labels corresponding to each module, where applicable.

> **Note**
>
> It is allowed to file bug reports for zephyr modules to track the corresponding upstream project bugs in Zephyr. These bug reports shall not affect the *Release Quality Criteria*.

2.10.4 Licensing requirements and policies

All source files in a module's codebase shall include a license header, unless the module repository has **main license file** that covers source files that do not include license headers.

Main license files shall be added in the module's codebase by Zephyr developers, only if they exist as part of the external project, and they contain a permissive OSI-compliant license. Main license files should preferably contain the full license text instead of including an SPDX license identifier. If multiple main license files are present it shall be made clear which license applies to each source file in a module's codebase.

Individual license headers in module source files supersede the main license.

Any new content to be added in a module repository will require to have license coverage.

> **Note**
>
> Zephyr recommends conveying module licensing via individual license headers and main license files. This not a hard requirement; should an external project have its own practice of conveying how licensing applies in the module's codebase (for example, by having a single or multiple main license files), this practice may be accepted by and be referred to in the Zephyr module, as long as licensing requirements, for example OSI compliance, are satisfied.

License policies

When creating a module repository a developer shall:

- import the main license files, if they exist in the external project, and
- document (for example in the module README or .yml file) the default license that covers the module's codebase.

License checks License checks (via CI tools) shall be enabled on every pull request that adds new content in module repositories.

2.10.5 Documentation requirements

All Zephyr module repositories shall include an .rst file documenting:

- the scope and the purpose of the module
- how the module integrates with Zephyr
- the owner of the module repository
- synchronization information with the external project (commit, SHA, version etc.)
- licensing information as described in *Licensing requirements and policies*.

The file shall be required for the inclusion of the module and the contained information should be kept up to date.

2.10.6 Testing requirements

All Zephyr modules should provide some level of **integration** testing, ensuring that the integration with Zephyr works correctly. Integration tests:

- may be in the form of a minimal set of samples and tests that reside in the zephyr main tree
- should verify basic usage of the module (configuration, functional APIs, etc.) that is integrated with Zephyr.
- shall be built and executed (for example in QEMU) as part of twister runs in pull requests that introduce changes in module repositories.

> **Note**
>
> New modules, that are candidates for being included in the Zephyr default manifest, shall provide some level of integration testing.

> **Note**
>
> Vendor HALs are implicitly tested via Zephyr tests built or executed on target platforms, so they do not need to provide integration tests.

The purpose of integration testing is not to provide functional verification of the module; this should be part of the testing framework of the external project.

Certain external projects provide test suites that reside in the upstream testing infrastructure but are written explicitly for Zephyr. These tests may (but are not required to) be part of the Zephyr test framework.

2.10.7 Deprecating and removing modules

Modules may be deprecated for reasons including, but not limited to:

- Lack of maintainership in the module
- Licensing changes in the external project
- Codebase becoming obsolete

The module information shall indicate whether a module is deprecated and the build system shall issue a warning when trying to build Zephyr using a deprecated module.

Deprecated modules may be removed from the Zephyr default manifest after 2 Zephyr releases.

> **Note**
>
> Repositories of removed modules shall remain accessible via their original URL, as they are required by older Zephyr versions.

2.10.8 Integrate modules in Zephyr build system

The build system variable **ZEPHYR_MODULES** is a CMake list of absolute paths to the directories containing Zephyr modules. These modules contain `CMakeLists.txt` and `Kconfig` files describing how to build and configure them, respectively. Module `CMakeLists.txt` files are added to the build using CMake's add_subdirectory() command, and the `Kconfig` files are included in the build's Kconfig menu tree.

If you have *west* installed, you don't need to worry about how this variable is defined unless you are adding a new module. The build system knows how to use west to set **ZEPHYR_MODULES**. You can add additional modules to this list by setting the **EXTRA_ZEPHYR_MODULES** CMake variable or by adding a **EXTRA_ZEPHYR_MODULES** line to `.zephyrrc` (See the section on *Environment Variables* for more details). This can be useful if you want to keep the list of modules found with west and also add your own. If **EXTRA_ZEPHYR_MODULES** is set in multiple places, for example both as an environment variable and as a CMake variable, the final list of additional modules will be the merged result of all sources.

> **Note**
>
> If the module `FOO` is provided by *west* but also given with `-DEXTRA_ZEPHYR_MODULES=/<path>/foo` then the module given by the command line variable **EXTRA_ZEPHYR_MODULES** will take precedence. This allows you to use a custom version of `FOO` when building and still use other Zephyr modules provided by *west*. This can for example be useful for special test purposes.

If you want to permanently add modules to the zephyr workspace and you are using zephyr as your manifest repository, you can also add a west manifest file into the submanifests directory. See submanifests/README.txt for more details.

See *Basics* for more on west workspaces.

Finally, you can also specify the list of modules yourself in various ways, or not use modules at all if your application doesn't need them.

2.10.9 Module yaml file description

A module can be described using a file named `zephyr/module.yml`. The format of `zephyr/module.yml` is described in the following:

Module name

Each Zephyr module is given a name by which it can be referred to in the build system.

The name should be specified in the `zephyr/module.yml` file. This will ensure the module name is not changeable through user-defined directory names or `west` manifest files:

```
name: <name>
```

In CMake the location of the Zephyr module can then be referred to using the CMake variable ZEPHYR_<MODULE_NAME>_MODULE_DIR and the variable ZEPHYR_<MODULE_NAME>_CMAKE_DIR holds the location of the directory containing the module's CMakeLists.txt file.

> **Note**
>
> When used for CMake and Kconfig variables, all letters in module names are converted to uppercase and all non-alphanumeric characters are converted to underscores (_). As example, the module foo-bar must be referred to as ZEPHYR_FOO_BAR_MODULE_DIR in CMake and Kconfig.

Here is an example for the Zephyr module foo:

```
name: foo
```

> **Note**
>
> If the name field is not specified then the Zephyr module name will be set to the name of the module folder. As example, the Zephyr module located in <workspace>/modules/bar will use bar as its module name if nothing is specified in zephyr/module.yml.

Module integration files (in-module)

Inclusion of build files, CMakeLists.txt and Kconfig, can be described as:

```
build:
  cmake: <cmake-directory>
  kconfig: <directory>/Kconfig
```

The cmake: <cmake-directory> part specifies that <cmake-directory> contains the CMakeLists.txt to use. The kconfig: <directory>/Kconfig part specifies the Kconfig file to use. Neither is required: cmake defaults to zephyr, and kconfig defaults to zephyr/Kconfig.

Here is an example module.yml file referring to CMakeLists.txt and Kconfig files in the root directory of the module:

```
build:
  cmake: .
  kconfig: Kconfig
```

Sysbuild integration

Sysbuild is the Zephyr build system that allows for building multiple images as part of a single application, the sysbuild build process can be extended externally with modules as needed, for example to add custom build steps or add additional targets to a build. Inclusion of sysbuild-specific build files, CMakeLists.txt and Kconfig, can be described as:

```
build:
  sysbuild-cmake: <cmake-directory>
  sysbuild-kconfig: <directory>/Kconfig
```

The sysbuild-cmake: <cmake-directory> part specifies that <cmake-directory> contains the CMakeLists.txt to use. The sysbuild-kconfig: <directory>/Kconfig part specifies the Kconfig file to use.

Here is an example module.yml file referring to CMakeLists.txt and Kconfig files in the sysbuild directory of the module:

```yaml
build:
  sysbuild-cmake: sysbuild
  sysbuild-kconfig: sysbuild/Kconfig
```

The module description file `zephyr/module.yml` can also be used to specify that the build files, `CMakeLists.txt` and `Kconfig`, are located in a *Module integration files (external)*.

Build files located in a `MODULE_EXT_ROOT` can be described as:

```yaml
build:
  sysbuild-cmake-ext: True
  sysbuild-kconfig-ext: True
```

This allows control of the build inclusion to be described externally to the Zephyr module.

Vulnerability monitoring

The module description file `zephyr/module.yml` can be used to improve vulnerability monitoring.

If your module needs to track vulnerabilities using an external reference (e.g your module is forked from another repository), you can use the `security` section. It contains the field `external-references` that contains a list of references that needs to be monitored for your module. The supported formats are:

- CPE (Common Platform Enumeration)
- PURL (Package URL)

```yaml
security:
  external-references:
    - <module-related-cpe>
    - <an-other-module-related-cpe>
    - <module-related-purl>
```

A real life example for Mbed TLS module could look like this:

```yaml
security:
  external-references:
    - cpe:2.3:a:arm:mbed_tls:3.5.2:*:*:*:*:*:*:*
    - pkg:github/Mbed-TLS/mbedtls@V3.5.2
```

> **Note**
>
> CPE field must follow the CPE 2.3 schema provided by NVD. PURL field must follow the PURL specification provided by Github.

Build system integration

When a module has a `module.yml` file, it will automatically be included into the Zephyr build system. The path to the module is then accessible through Kconfig and CMake variables.

Zephyr modules In both Kconfig and CMake, the variable `ZEPHYR_<MODULE_NAME>_MODULE_DIR` contains the absolute path to the module.

Additionally, `ZEPHYR_<MODULE_NAME>_MODULE` and `ZEPHYR_<MODULE_NAME>_MODULE_BLOBS` (in case the module declares blobs) symbols are automatically generated for available modules. These can be used e.g. to declare dependencies from other Kconfig symbols which depend on the module or blobs from the module. To satisfy compliance checking when building Zephyr without the

module present, it's recommended for the module to have default definitions for these symbols in its respective Kconfig file under `modules/` in the Zephyr main tree.

In CMake, `ZEPHYR_<MODULE_NAME>_CMAKE_DIR` contains the absolute path to the directory containing the `CMakeLists.txt` file that is included into CMake build system. This variable's value is empty if the module.yml file does not specify a CMakeLists.txt.

To read these variables for a Zephyr module named `foo`:

- In CMake: use `${ZEPHYR_FOO_MODULE_DIR}` for the module's top level directory, and `${ZEPHYR_FOO_CMAKE_DIR}` for the directory containing its `CMakeLists.txt`
- In Kconfig: use `$(ZEPHYR_FOO_MODULE_DIR)` for the module's top level directory

Notice how a lowercase module name `foo` is capitalized to `FOO` in both CMake and Kconfig.

These variables can also be used to test whether a given module exists. For example, to verify that `foo` is the name of a Zephyr module:

```
if(ZEPHYR_FOO_MODULE_DIR)
  # Do something if FOO exists.
endif()
```

In Kconfig, the variable may be used to find additional files to include. For example, to include the file `some/Kconfig` in module `foo`:

```
source "$(ZEPHYR_FOO_MODULE_DIR)/some/Kconfig"
```

During CMake processing of each Zephyr module, the following variables are also available:

- the current module's name: `${ZEPHYR_CURRENT_MODULE_NAME}`
- the current module's top level directory: `${ZEPHYR_CURRENT_MODULE_DIR}`
- the current module's `CMakeLists.txt` directory: `${ZEPHYR_CURRENT_CMAKE_DIR}`

This removes the need for a Zephyr module to know its own name during CMake processing. The module can source additional CMake files using these `CURRENT` variables. For example:

```
include(${ZEPHYR_CURRENT_MODULE_DIR}/cmake/code.cmake)
```

It is possible to append values to a Zephyr CMake list variable from the module's first CMakeLists.txt file. To do so, append the value to the list and then set the list in the PARENT_SCOPE of the CMakeLists.txt file. For example, to append `bar` to the `FOO_LIST` variable in the Zephyr CMakeLists.txt scope:

```
list(APPEND FOO_LIST bar)
set(FOO_LIST ${FOO_LIST} PARENT_SCOPE)
```

An example of a Zephyr list where this is useful is when adding additional directories to the `SYSCALL_INCLUDE_DIRS` list.

Sysbuild modules In both Kconfig and CMake, the variable `SYSBUILD_CURRENT_MODULE_DIR` contains the absolute path to the sysbuild module. In CMake, `SYSBUILD_CURRENT_CMAKE_DIR` contains the absolute path to the directory containing the `CMakeLists.txt` file that is included into CMake build system. This variable's value is empty if the module.yml file does not specify a CMakeLists.txt.

To read these variables for a sysbuild module:

- In CMake: use `${SYSBUILD_CURRENT_MODULE_DIR}` for the module's top level directory, and `${SYSBUILD_CURRENT_CMAKE_DIR}` for the directory containing its `CMakeLists.txt`
- In Kconfig: use `$(SYSBUILD_CURRENT_MODULE_DIR)` for the module's top level directory

In Kconfig, the variable may be used to find additional files to include. For example, to include the file `some/Kconfig`:

```
source "$(SYSBUILD_CURRENT_MODULE_DIR)/some/Kconfig"
```

The module can source additional CMake files using these variables. For example:

```
include(${SYSBUILD_CURRENT_MODULE_DIR}/cmake/code.cmake)
```

It is possible to append values to a Zephyr CMake list variable from the module's first CMakeLists.txt file. To do so, append the value to the list and then set the list in the PARENT_SCOPE of the CMakeLists.txt file. For example, to append `bar` to the `FOO_LIST` variable in the Zephyr CMakeLists.txt scope:

```
list(APPEND FOO_LIST bar)
set(FOO_LIST ${FOO_LIST} PARENT_SCOPE)
```

Sysbuild modules hooks Sysbuild provides an infrastructure which allows a sysbuild module to define a function which will be invoked by sysbuild at a pre-defined point in the CMake flow.

Functions invoked by sysbuild:

- `<module-name>_pre_cmake(IMAGES <images>)`: This function is called for each sysbuild module before CMake configure is invoked for all images.
- `<module-name>_post_cmake(IMAGES <images>)`: This function is called for each sysbuild module after CMake configure has completed for all images.
- `<module-name>_pre_domains(IMAGES <images>)`: This function is called for each sysbuild module before domains yaml is created by sysbuild.
- `<module-name>_post_domains(IMAGES <images>)`: This function is called for each sysbuild module after domains yaml has been created by sysbuild.

arguments passed from sysbuild to the function defined by a module:

- `<images>` is the list of Zephyr images that will be created by the build system.

If a module `foo` want to provide a post CMake configure function, then the module's sysbuild `CMakeLists.txt` file must define function `foo_post_cmake()`.

To facilitate naming of functions, the module name is provided by sysbuild CMake through the `SYSBUILD_CURRENT_MODULE_NAME` CMake variable when loading the module's sysbuild `CMakeLists.txt` file.

Example of how the `foo` sysbuild module can define `foo_post_cmake()`:

```
function(${SYSBUILD_CURRENT_MODULE_NAME}_post_cmake)
  cmake_parse_arguments(POST_CMAKE "" "" "IMAGES" ${ARGN})

  message("Invoking ${CMAKE_CURRENT_FUNCTION}. Images: ${POST_CMAKE_IMAGES}")
endfunction()
```

Zephyr module dependencies

A Zephyr module may be dependent on other Zephyr modules to be present in order to function correctly. Or it might be that a given Zephyr module must be processed after another Zephyr module, due to dependencies of certain CMake targets.

Such a dependency can be described using the `depends` field.

```yaml
build:
  depends:
    - <module>
```

Here is an example for the Zephyr module `foo` that is dependent on the Zephyr module `bar` to be present in the build system:

```yaml
name: foo
build:
  depends:
    - bar
```

This example will ensure that `bar` is present when `foo` is included into the build system, and it will also ensure that `bar` is processed before `foo`.

Module integration files (external)

Module integration files can be located externally to the Zephyr module itself. The `MODULE_EXT_ROOT` variable holds a list of roots containing integration files located externally to Zephyr modules.

Module integration files in Zephyr The Zephyr repository contain `CMakeLists.txt` and `Kconfig` build files for certain known Zephyr modules.

Those files are located under

```
<ZEPHYR_BASE>
└── modules
    └── <module_name>
        ├── CMakeLists.txt
        └── Kconfig
```

Module integration files in a custom location You can create a similar `MODULE_EXT_ROOT` for additional modules, and make those modules known to Zephyr build system.

Create a `MODULE_EXT_ROOT` with the following structure

```
<MODULE_EXT_ROOT>
└── modules
    ├── modules.cmake
    └── <module_name>
        ├── CMakeLists.txt
        └── Kconfig
```

and then build your application by specifying `-DMODULE_EXT_ROOT` parameter to the CMake build system. The `MODULE_EXT_ROOT` accepts a CMake list of roots as argument.

A Zephyr module can automatically be added to the `MODULE_EXT_ROOT` list using the module description file `zephyr/module.yml`, see *Build settings*.

> **Note**
>
> `ZEPHYR_BASE` is always added as a `MODULE_EXT_ROOT` with the lowest priority. This allows you to overrule any integration files under `<ZEPHYR_BASE>/modules/<module_name>` with your own implementation your own `MODULE_EXT_ROOT`.

The `modules.cmake` file must contain the logic that specifies the integration files for Zephyr modules via specifically named CMake variables.

To include a module's CMake file, set the variable ZEPHYR_<MODULE_NAME>_CMAKE_DIR to the path containing the CMake file.

To include a module's Kconfig file, set the variable ZEPHYR_<MODULE_NAME>_KCONFIG to the path to the Kconfig file.

The following is an example on how to add support the FOO module.

Create the following structure

```
<MODULE_EXT_ROOT>
└── modules
    ├── modules.cmake
    └── foo
        ├── CMakeLists.txt
        └── Kconfig
```

and inside the modules.cmake file, add the following content

```
set(ZEPHYR_FOO_CMAKE_DIR ${CMAKE_CURRENT_LIST_DIR}/foo)
set(ZEPHYR_FOO_KCONFIG   ${CMAKE_CURRENT_LIST_DIR}/foo/Kconfig)
```

Module integration files (zephyr/module.yml) The module description file zephyr/module.yml can be used to specify that the build files, CMakeLists.txt and Kconfig, are located in a *Module integration files (external)*.

Build files located in a MODULE_EXT_ROOT can be described as:

```
build:
  cmake-ext: True
  kconfig-ext: True
```

This allows control of the build inclusion to be described externally to the Zephyr module.

The Zephyr repository itself is always added as a Zephyr module ext root.

Build settings

It is possible to specify additional build settings that must be used when including the module into the build system.

All root settings are relative to the root of the module.

Build settings supported in the module.yml file are:

- board_root: Contains additional boards that are available to the build system. Additional boards must be located in a <board_root>/boards folder.
- dts_root: Contains additional dts files related to the architecture/soc families. Additional dts files must be located in a <dts_root>/dts folder.
- snippet_root: Contains additional snippets that are available for use. These snippets must be defined in snippet.yml files underneath the <snippet_root>/snippets folder. For example, if you have snippet_root: foo, then you should place your module's snippet.yml files in <your-module>/foo/snippets or any nested subdirectory.
- soc_root: Contains additional SoCs that are available to the build system. Additional SoCs must be located in a <soc_root>/soc folder.
- arch_root: Contains additional architectures that are available to the build system. Additional architectures must be located in a <arch_root>/arch folder.
- module_ext_root: Contains CMakeLists.txt and Kconfig files for Zephyr modules, see also *Module integration files (external)*.

- `sca_root`: Contains additional *SCA* tool implementations available to the build system. Each tool must be located in `<sca_root>/sca/<tool>` folder. The folder must contain a `sca.cmake`.

Example of a `module.yaml` file containing additional roots, and the corresponding file system layout.

```yaml
build:
  settings:
    board_root: .
    dts_root: .
    soc_root: .
    arch_root: .
    module_ext_root: .
```

requires the following folder structure:

```
<zephyr-module-root>
├── arch
├── boards
├── dts
├── modules
└── soc
```

Test Runner (Twister) integration

To execute both tests and samples available in modules, the Zephyr test runner (twister) should be pointed to the directories containing those samples and tests. This can be done by specifying the path to both samples and tests in the `zephyr/module.yml` file. Additionally, if a module defines out of tree boards, the module file can point twister to the path where those files are maintained in the module. For example:

```yaml
build:
  cmake: .
samples:
  - samples
tests:
  - tests
boards:
  - boards
```

Tests and Samples defined in the `zephyr/module.yml` file are not detected by twister automatically. To make twister aware of the tests and samples defined in modules, the path for those tests and samples must be added to the command line when executing twister, for example:

```bash
./scripts/zephyr_module.py --twister-out module_tests.args
if [ -s module_tests.args ]; then
    west twister +module_tests.args --outdir module_tests ...
fi
```

Binary Blobs

Zephyr supports fetching and using *binary blobs*, and their metadata is contained entirely in `zephyr/module.yml`. This is because a binary blob must always be associated with a Zephyr module, and thus the blob metadata belongs in the module's description itself.

Binary blobs are fetched using *west blobs*. If `west` is *not used*, they must be downloaded and verified manually.

The `blobs` section in `zephyr/module.yml` consists of a sequence of maps, each of which has the following entries:

- `path`: The path to the binary blob, relative to the `zephyr/blobs/` folder in the module repository
- `sha256`: SHA-256 checksum of the binary blob file
- `type`: The *type of binary blob*. Currently limited to `img` or `lib`
- `version`: A version string
- `license-path`: Path to the license file for this blob, relative to the root of the module repository
- `url`: URL(s) that identify the location the blob will be fetched from, as well as the fetching scheme to use. If it contains a list instead of a single string, each URL will be considered as a fallback for fetching the same blob.
- `description`: Human-readable description of the binary blob
- `doc-url`: A URL pointing to the location of the official documentation for this blob

The following entries may also be present:

- `click-through`: A boolean indicating if a click-through license must be accepted to download this blob
- `size`: Size of the blob in bytes. May be required by some fetchers
- `fetcher`: The method used to download the blob. If not set, the method is inferred from the URL

Package manager dependencies

Zephyr modules can describe dependencies available from package managers, currently only `pip` is supported.

A west extension command `west packages <manager>` is available to list dependencies for Zephyr and present modules that leverage this feature in their `module.yml` file. Run `west help packages` for more details.

Python pip Calling `west packages pip` lists requirement files for Zephyr and modules. Passing `--install` installs these if there's an active virtual environment.

The following example demonstrates a `zephyr/module.yml` file with some requirement files in the `scripts` directory of the module.

```
package-managers:
  pip:
    requirement-files:
      - scripts/requirements-build.txt
      - scripts/requirements-doc.txt
```

External Runners

If a module has out of tree boards that require custom *runners*, then it can add a list to its `zephyr/module.yml` file, for example:

```
runners:
  - file: scripts/my-runner.py
```

Each file entry is imported when executing `west flash` or `west debug` and subclasses of the `ZephyrBinaryRunner` are registered for use.

Module Inclusion

Using West If west is installed and `ZEPHYR_MODULES` is not already set, the build system finds all the modules in your *west installation* and uses those. It does this by running *west list* to get the paths of all the projects in the installation, then filters the results to just those projects which have the necessary module metadata files.

Each project in the `west list` output is tested like this:

- If the project contains a file named `zephyr/module.yml`, then the content of that file will be used to determine which files should be added to the build, as described in the previous section.
- Otherwise (i.e. if the project has no `zephyr/module.yml`), the build system looks for `zephyr/CMakeLists.txt` and `zephyr/Kconfig` files in the project. If both are present, the project is considered a module, and those files will be added to the build.
- If neither of those checks succeed, the project is not considered a module, and is not added to `ZEPHYR_MODULES`.

Without West If you don't have west installed or don't want the build system to use it to find Zephyr modules, you can set `ZEPHYR_MODULES` yourself using one of the following options. Each of the directories in the list must contain either a `zephyr/module.yml` file or the files `zephyr/CMakeLists.txt` and `Kconfig`, as described in the previous section.

1. At the CMake command line, like this:

```
cmake -DZEPHYR_MODULES=<path-to-module1>[;<path-to-module2>[...]] ...
```

2. At the top of your application's top level `CMakeLists.txt`, like this:

```
set(ZEPHYR_MODULES <path-to-module1> <path-to-module2> [...])
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

   If you choose this option, make sure to set the variable **before** calling `find_package(Zephyr ...)`, as shown above.

3. In a separate CMake script which is pre-loaded to populate the CMake cache, like this:

```
# Put this in a file with a name like "zephyr-modules.cmake"
set(ZEPHYR_MODULES <path-to-module1> <path-to-module2>
  CACHE STRING "pre-cached modules")
```

   You can tell the build system to use this file by adding `-C zephyr-modules.cmake` to your CMake command line.

Not using modules If you don't have west installed and don't specify `ZEPHYR_MODULES` yourself, then no additional modules are added to the build. You will still be able to build any applications that don't require code or Kconfig options defined in an external repository.

2.10.10 Submitting changes to modules

When submitting new or making changes to existing modules the main repository Zephyr needs a reference to the changes to be able to verify the changes. In the main tree this is done using revisions. For code that is already merged and part of the tree we use the commit hash, a tag, or a branch name. For pull requests however, we require specifying the pull request number in the revision field to allow building the zephyr main tree with the changes submitted to the module.

To avoid merging changes to master with pull request information, the pull request should be marked as `DNM` (Do Not Merge) or preferably a draft pull request to make sure it is not merged by mistake and to allow for the module to be merged first and be assigned a permanent commit hash. Drafts reduce noise by not automatically notifying anyone until marked as "Ready for

review". Once the module is merged, the revision will need to be changed either by the submitter or by the maintainer to the commit hash of the module which reflects the changes.

Note that multiple and dependent changes to different modules can be submitted using exactly the same process. In this case you will change multiple entries of all modules that have a pull request against them.

Process for submitting a new module

Please follow the process in *Submission and review process* and obtain the TSC approval to integrate the external source code as a module

If the request is approved, a new repository will created by the project team and initialized with basic information that would allow submitting code to the module project following the project contribution guidelines.

If a module is maintained as a fork of another project on Github, the Zephyr module related files and changes in relation to upstream need to be maintained in a special branch named `zephyr`.

Maintainers from the Zephyr project will create the repository and initialize it. You will be added as a collaborator in the new repository. Submit the module content (code) to the new repository following the guidelines described *here*, and then add a new entry to the west.yml with the following information:

```
- name: <name of repository>
  path: <path to where the repository should be cloned>
  revision: <ref pointer to module pull request>
```

For example, to add *my_module* to the manifest:

```
- name: my_module
  path: modules/lib/my_module
  revision: pull/23/head
```

Where 23 in the example above indicated the pull request number submitted to the *my_module* repository. Once the module changes are reviewed and merged, the revision needs to be changed to the commit hash from the module repository.

Process for submitting changes to existing modules

1. Submit the changes using a pull request to an existing repository following the *contribution guidelines* and *expectations*.
2. Submit a pull request changing the entry referencing the module into the west.yml of the main Zephyr tree with the following information:

```
- name: <name of repository>
  path: <path to where the repository should be cloned>
  revision: <ref pointer to module pull request>
```

For example, to add *my_module* to the manifest:

```
- name: my_module
  path: modules/lib/my_module
  revision: pull/23/head
```

Where 23 in the example above indicated the pull request number submitted to the *my_module* repository. Once the module changes are reviewed and merged, the revision needs to be changed to the commit hash from the module repository.

2.11 West (Zephyr's meta-tool)

The Zephyr project includes a swiss-army knife command line tool named `west`[1]. West is developed in its own repository.

West's built-in commands provide a multiple repository management system with features inspired by Google's Repo tool and Git submodules. West is also "pluggable": you can write your own west extension commands which add additional features to west. Zephyr uses this to provide conveniences for building applications, flashing and debugging them, and more.

Like `git` and `docker`, the top-level `west` command takes some common options, a sub-command to run, and then options and arguments for that sub-command:

```
west [common-opts] <command> [opts] <args>
```

Since west v0.8, you can also run west like this:

```
python3 -m west [common-opts] <command> [opts] <args>
```

You can run `west --help` (or `west -h` for short) to get top-level help for available west commands, and `west <command> -h` for detailed help on each command.

2.11.1 Installing west

West is written in Python 3 and distributed through PyPI. Use `pip3` to install or upgrade west:

On Linux:

```
pip3 install --user -U west
```

On Windows and macOS:

```
pip3 install -U west
```

> **Note**
>
> See *Python and pip* for additional clarification on using the `--user` switch.

Afterwards, you can run `pip3 show -f west` for information on where the west binary and related files were installed.

Once west is installed, you can use it to *clone the Zephyr repositories*.

Structure

West's code is distributed via PyPI in a Python package named `west`. This distribution includes a launcher executable, which is also named `west` (or `west.exe` on Windows).

When west is installed, the launcher is placed by `pip3` somewhere in the user's filesystem (exactly where depends on the operating system, but should be on the `PATH` *environment variable*). This launcher is the command-line entry point to running both built-in commands like `west init`, `west update`, along with any extensions discovered in the workspace.

In addition to its command-line interface, you can also use west's Python APIs directly. See west-apis for details.

[1] Zephyr is an English name for the Latin Zephyrus, the ancient Greek god of the west wind.

Enabling shell completion

West currently supports shell completion in the following shells:

- bash
- zsh
- fish
- powershell (board qualifiers only)

In order to enable shell completion, you will need to obtain the corresponding completion script and have it sourced. Using the completion scripts:

bash

One-time setup:

```
source <(west completion bash)
```

Permanent setup:

```
west completion bash > ~/west-completion.bash; printf '\n%s\n' "source ~/
↪west-completion.bash" >> ~/.bashrc
```

zsh

One-time setup:

```
source <(west completion zsh)
```

Permanent setup:

```
west completion zsh > "${fpath[1]}/_west"
```

fish

One-time setup:

```
west completion fish | source
```

Permanent setup:

```
west completion fish > $HOME/.config/fish/completions/west.fish
```

powershell

One-time setup:

```
west completion powershell | Out-String | Invoke-Expression
```

Permanent setup:

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
New-item -type file -force $PROFILE
west completion powershell > $HOME/west-completion.ps1
(Add-Content -Path $PROFILE -Value ". '{$HOME/west-completion.ps1}'")
```

2.11.2 West Release Notes

v1.5.0

Major changes:

- Add support for auto-caching. Pass the `--auto-cache <directory>` argument to `west update`.

Other changes:

- Allow combining `--name-cache` and `--path-cache` for `west update`.
- Document default revision value in the manifest schema.

Bug fixes:

- Allow empty or missing manifest projects list.
- Make `manifest.group-filter` list order deterministic when freezing or resolving manifest files.

v1.4.0

Changes:

- Allow appending data to configuration strings. To append to a value for `<name>`, type: `west config -a <name> <value>`.
- Add `--untracked` argument option to `west manifest`. Run `west manifest --untracked` in a workspace to print all files and directories that are not tracked or managed by west.
- Add `--inactive` argument option to `west list` to support printing inactive projects.
- Support `--active-only` argument option for the `west manifest --resolve` and `west manifest --freeze` commands. This allows freezing workspaces with active project or group filters.

API changes:

- `west.manifest.Manifest` methods `as_dict()`, `as_frozen_dict()`, `as_yaml()` and `as_frozen_yaml()` now have an optional `active_only` argument (defaults to `False`) to return an object containing all projects or only the active ones.

v1.3.0

Major changes:

- Added support for *West aliases* commands.
- Adopt the pyproject TOML specification for packaging.

Other changes:

- Add cache support for submodules.
- Decode manifest files as UTF-8 by default.
- Pass unknown arguments for `west diff` and `west status` to underlying `git` commands.
- Added `--manifest` argument to `west diff` to allow comparing the current workspace to the manifest revisions.
- Environment variables can be used with west forall The following are defined:
  - `WEST_PROJECT_NAME`
  - `WEST_PROJECT_PATH`
  - `WEST_PROJECT_ABSPATH`
  - `WEST_PROJECT_REVISION`
  - `WEST_PROJECT_URL`
  - `WEST_PROJECT_REMOTE`

- Added support for early argument `-q/--quiet` to reduce verbosity.
- Added `-o/--clone-opt` argument to `west init` to pass to `git clone`.
- Support Python 3.13 and drop support for Python 3.8.
- Prevent manifests from having projects in the `.west` directory.
- Add NTFS workarounds and `--rename-delay` for `west init`.
- Print a stack trace when calling die in debug `-vvv`.

Bug fixes:

- Use `'backslashreplace'` not to crash on malformed UTF from subprocess.
- Fix handling in `west diff` for repositories with merge conflicts. Additionally improve error printing and handle `git diff` return codes.
- Fix `--freeze` and `--resolve` for the `west manifest` command when git submodules are used.

v1.2.0

Major changes:

- New `west grep` command for running a "grep tool" in your west workspace's repositories. Currently, `git grep`, ripgrep, and standard `grep` are supported grep tools.

  To run this command to get `git grep foo` results from all cloned, active repositories, run:

  ```
  west grep foo
  ```

  Here are some other examples for running different grep commands with `west grep`:

`git grep --untracked`	`west grep --untracked foo`
`ripgrep`	`west grep --tool ripgrep foo`
`grep --recursive`	`west grep --tool grep foo`

  To switch the default grep tool in your workspace, run the appropriate command in this table:

`ripgrep`	`west config grep.tool ripgrep`
`grep`	`west config grep.tool grep`

  For more details, run `west help grep`.

Other changes:

- The manifest file format now supports a `description` field in each `projects:` element. See *Projects* for examples.
- `west list --format` now accepts `{description}` in the format string, which prints the project's `description:` value.
- `west compare` now always prints information about *The manifest-rev branch*.

Bug fixes:

- `west init` aborts if the destination directory already exists.

API changes:

- `west.commands.WestCommand` methods `check_call()` and `check_output()` now take any kwargs that can be passed on to the underlying subprocess function.

- `west.commands.WestCommand.run_subprocess()`: new wrapper around `subprocess.run()`. This could not be named `run()` because `WestCommand` already had a method by this name.
- `west.commands.WestCommand` methods `dbg()`, `inf()`, `wrn()`, and `err()` now all take an `end` kwarg, which is passed on to the call to `print()`.
- `west.manifest.Project` now has a `description` attribute, which contains the parsed value of the `description:` field in the manifest data.

v1.1.0

Major changes:

- `west compare`: new command that compares the state of the workspace against the manifest.
- Support for a new `manifest.project-filter` configuration option. See *Built-in Configuration Options* for details. The `west manifest --freeze` and `west manifest --resolve` commands currently cannot be used when this option is set. This restriction can be removed in a later release.
- Project names which contain comma (`,`) or whitespace now generate warnings. These warnings are errors if the new `manifest.project-filter` configuration option is set. The warnings may be promoted to errors in a future major version of west.

Other changes:

- `west forall` now takese a `--group` argument that can be used to restrict the command to only run in one or more groups. Run `west help forall` for details.
- All west commands will now output log messages from west API modules at warning level or higher. In addition, the `--verbose` argument to west can be used once to include informational messages, or twice to include debug messages, from all commands.

Bug fixes:

- Various improvements to error messages, debug logging, and error handling.

API changes:

- `west.manifest.Manifest.is_active()` now respects the `manifest.project-filter` configuration option's value.

v1.0.1

Major changes:

- Manifest schema version "1.0" is now available for use in this release. This is identical to the "0.13" schema version in terms of features, but can be used by applications that do not wish to use a "0.x" manifest "version:" field. See *Version* for details on this feature.

Bug fixes:

- West no longer exits with a successful error code when sent an interrupt signal. Instead, it exits with a platform-specific error code and signals to the calling environment that the process was interrupted.

v1.0.0

Major changes in this release:

- The west-apis are now declared stable. Any breaking changes will be communicated by a major version bump from v1.x.y to v2.x.y.

- West v1.0 no longer works with the Zephyr v1.14 LTS releases. This LTS has long been obsoleted by Zephyr v2.7 LTS. If you need to use Zephyr v1.14, you must use west v0.14 or earlier.
- Like the rest of Zephyr, west now requires Python v3.8 or later
- West commands no longer accept abbreviated command line arguments. For example, you must now specify `west update --keep-descendants` instead of using an abbreviation like `west update --keep-d`. This is part of a change applied to all of Zephyr's Python scripts' command-line interfaces. The abbreviations were causing problems in practice when commands were updated to add new options with similar names but different behavior to existing ones.

Other changes:

- All built-in west functions have stopped using `west.log`
- `west update`: new `--submodule-init-config` option. See commit 9ba92b05 for details.

Bug fixes:

- West extension commands that failed to load properly sometimes dumped stack. This has been fixed and west now prints a sensible error message in this case.
- `west config` now fails on malformed configuration option arguments which lack a `.` in the option name

API changes:

- The west package now contains the metadata files necessary for some static analyzers (such as mypy) to auto-detect its type annotations. See commit d9f00e24 for details.
- the deprecated `west.build` module used for Zephyr v1.14 LTS compatibility was removed
- the deprecated `west.cmake` module used for Zephyr v1.14 LTS compatibility was removed
- the `west.log` module is now deprecated. This module uses global state, which can make it awkward to use it as an API which multiple different python modules may rely on.
- The west-apis-commands module got some new APIs which lay groundwork for a future change to add a global verbosity control to a command's output, and work to remove global state from the `west` package's API:
  - New `west.commands.WestCommand.__init__()` keyword argument: `verbosity`
  - New `west.commands.WestCommand` property: `color_ui`
  - New `west.commands.WestCommand` methods, which should be used to print output from extension commands instead of writing directly to sys.stdout or sys.stderr: `inf()`, `wrn()`, `err()`, `die()`, `banner()`, `small_banner()`
  - New `west.commands.VERBOSITY` enum

v0.14.0

Bug fixes:

- West commands that were run with a bad local configuration file dumped stack in a confusing way. This has been fixed and west now prints a sensible error message in this case.
- A bug in the way west looks for the zephyr repository was fixed. The bug itself usually appeared when running an extension command like `west build` in a new workspace for the first time; this used to fail (just for the first time, not on subsequent command invocations) unless you ran the command in the workspace's top level directory.
- West now prints sensible error messages when the user lacks permission to open the manifest file instead of dumping stack traces.

API changes:

- The `west.manifest.MalformedConfig` exception type has been moved to the `west.configuration` module
- The `west.manifest.MalformedConfig` exception type has been moved to the west.configuration module
- The `west.configuration.Configuration` class now raises `MalformedConfig` instead of `RuntimeError` in some cases

v0.13.1

Bug fix:

- When calling west.manifest.Manifest.from_file() when outside of a workspace, west again falls back on the ZEPHYR_BASE environment variable to locate the workspace.

v0.13.0

New features:

- You can now associate arbitrary user data with the manifest repository itself in the `manifest: self: userdata:` value, like so:

```
manifest:
  self:
    userdata: <any YAML value can go here>
```

Bug fixes:

- The path to the manifest repository reported by west could be incorrect in certain circumstances detailed in [issue #572](https://github.com/zephyrproject-rtos/west/issues/572). This has been fixed as part of a larger overhaul of path handling support in the `west.manifest` API module.
- The `west.Manifest.ManifestProject.__repr__` return value was fixed

API changes:

- `west.configuration.Configuration`: new object-oriented interface to the current configuration. This reflects the system, global, and workspace-local configuration values, and allows you to read, write, and delete configuration options from any or all of these locations.
- `west.commands.WestCommand`:
  - `config`: new attribute, returns a `Configuration` object or aborts the program if none is set. This is always usable from within extension command `do_run()` implementations.
  - `has_config`: new boolean attribute, which is `True` if and only if reading `self.config` will abort the program.
- The path handling in the `west.manifest` package has been overhauled in a backwards-incompatible way. For more details, see commit [56cfe8d1d1](https://github.com/zephyrproject-rtos/west/commit/56cfe8d1d1f3c9b45de3e793c738acd62db52aca).
- `west.manifest.Manifest.validate()`: this now returns the validated data as a Python dict. This can be useful if the value passed to this function was a str, and the dict is desired.
- `west.manifest.Manifest`: new:
  - path attributes `abspath`, `posixpath`, `relative_path`, `yaml_path`, `repo_path`, `repo_posixpath`
  - `userdata` attribute, which contains the parsed value from `manifest: self: userdata:`, or is None
  - `from_topdir()` factory method

- `west.manifest.ManifestProject`: new `userdata` attribute, which also contains the parsed value from `manifest: self: userdata:`, or is None
- `west.manifest.ManifestImportFailed`: the constructor can now take any value; this can be used to reflect failed imports from a *map* or other compound value.
- Deprecated configuration APIs:

  The following APIs are now deprecated in favor of using a `Configuration` object. Usually this will be done via `self.config` from a `WestCommand` instance, but this can be done directly by instantiating a `Configuration` object for other usages.

  - `west.configuration.config`
  - `west.configuration.read_config`
  - `west.configuration.update_config`
  - `west.configuration.delete_config`

v0.12.0

New features:

- West now works on the MSYS2 platform.
- West manifest files can now contain arbitrary user data associated with each project. See *Repository user data* for details.

Bug fixes:

- The `west list` command's `{sha}` format key has been fixed for the manifest repository; it now prints `N/A` ("not applicable") as expected.

API changes:

- The `west.manifest.Project.userdata` attribute was added to support project user data.

v0.11.1

New features:

- `west status` now only prints output for projects which have a nonempty status.

Bug fixes:

- The manifest file parser was incorrectly allowing project names which contain the path separator characters `/` and `\`. These invalid characters are now rejected.

  Note: if you need to place a project within a subdirectory of the workspace topdir, use the `path:` key. If you need to customize a project's fetch URL relative to its remote `url-base:`, use `repo-path:`. See *Projects* for examples.
- The changes made in west v0.10.1 to the `west init --manifest-rev` option which selected the default branch name were leaving the manifest repository in a detached HEAD state. This has been fixed by using `git clone` internally instead of `git init` and `git fetch`. See issue #522 for details.
- The `WEST_CONFIG_LOCAL` environment variable now correctly overrides the default location, `<workspace topdir>/.west/config`.
- `west update --fetch=smart` (`smart` is the default) now correctly skips fetches for project revisions which are lightweight tags (it already worked correctly for annotated tags; only lightweight tags were unnecessarily fetched).

Other changes:

- The fix for issue #522 mentioned above introduces a new restriction. The `west init --manifest-rev` option value, if given, must now be either a branch or a tag. In particular, "pseudo-branches" like GitHub's `pull/1234/head` references which could previously be used to fetch a pull request can no longer be passed to `--manifest-rev`. Users must now fetch and check out such revisions manually after running `west init`.

API changes:

- `west.manifest.Manifest.get_projects()` avoids incorrect results in some edge cases described in issue #523.
- `west.manifest.Project.sha()` now works correctly for tag revisions. (This applies to both lightweight and annotated tags.)

v0.11.0

New features:

- `west update` now supports `--narrow`, `--name-cache`, and `--path-cache` options. These can be influenced by the `update.narrow`, `update.name-cache`, and `update.path-cache` *Configuration* options. These can be used to optimize the speed of the update.
- `west update` now supports a `--fetch-opt` option that will be passed to the `git fetch` command used to fetch remote revisions when updating each project.

Bug fixes:

- `west update` now synchronizes Git submodules in projects by default. This avoids issues if the URL changes in the manifest file from when the submodule was first initialized. This behavior can be disabled by setting the `update.sync-submodules` configuration option to `false`.

Other changes:

- the west-apis-manifest module has fixed docstrings for the Project class

v0.10.1

New features:

- The *west init* command's `--manifest-rev` (`--mr`) option no longer defaults to `master`. Instead, the command will query the repository for its default branch name and use that instead. This allows users to move from `master` to `main` without breaking scripts that do not provide this option.

v0.10.0

New features:

- The `name` key in a project's *submodules list* is now optional.

Bug fixes:

- West now checks that the manifest schema version is one of the explicitly allowed values documented in *Version*. The old behavior was just to check that the schema version was newer than the west version where the `manifest: version:` key was introduced. This incorrectly allowed invalid schema versions, like `0.8.2`.

Other changes:

- A manifest file's `group-filter` is now propagated through an `import`. This is a change from how west v0.9.x handled this. In west v0.9.x, only the top level manifest file's `group-filter` had any effect; the group filter lists from any imported manifests were ignored.

  Starting with west v0.10.0, the group filter lists from imported manifests are also imported. For details, see *Group Filters and Imports*.

The new behavior will take effect if `manifest: version:` is not given or is at least `0.10`. The old behavior is still available in the top level manifest file only with an explicit `manifest: version: 0.9`. See *Version* for more information on schema versions.

See west pull request #482 for the motivation for this change and additional context.

v0.9.1

Bug fixes:

- Commands like `west manifest --resolve` now correctly include group and group filter information.

Other changes:

- West now warns if you combine `import` with `group-filter`. Semantics for this combination have changed starting with v0.10.x. See the v0.10.0 release notes above for more information.

v0.9.0

Warning

The `west config` fix described below comes at a cost: any comments or other manual edits in configuration files will be removed when setting a configuration option via that command or the `west.configuration` API.

Warning

Combining the `group-filter` feature introduced in this release with manifest imports is discouraged. The resulting behavior has changed in west v0.10.

New features:

- West manifests now support *Git Submodules in Projects*. This allows you to clone Git submodules into a west project repository in addition to the project repository itself.
- West manifests now support *Project Groups*. Project groups can be enabled and disabled to determine what projects are "active", and therefore will be acted upon by the following commands: `west update`, `west list`, `west diff`, `west status`, `west forall`.
- `west update` no longer updates inactive projects by default. It now supports a `--group-filter` option which allows for one-time modifications to the set of enabled and disabled project groups.
- Running `west list`, `west diff`, `west status`, or `west forall` with no arguments does not print information for inactive projects by default. If the user specifies a list of projects explicitly at the command line, output for them is included regardless of whether they are active.

  These commands also now support `--all` arguments to include all projects, even inactive ones.
- `west list` now supports a `{groups}` format string key in its `--format` argument.

Bug fixes:

- The `west config` command and `west.configuration` API did not correctly store some configuration values, such as strings which contain commas. This has been fixed; see commit 36f3f91e for details.

- A manifest file with an empty `manifest: self: path:` value is invalid, but west used to let it pass silently. West now rejects such manifests.
- A bug affecting the behavior of the `west init -l .` command was fixed; see issue #435.

API changes:

- added `west.manifest.Manifest.is_active()`
- added `west.manifest.Manifest.group_filter`
- added `submodules` attribute to `west.manifest.Project`, which has newly added type `west.manifest.Submodule`

Other changes:

- The *Manifest Imports* feature now supports the terms `allowlist` and `blocklist` instead of `whitelist` and `blacklist`, respectively.

  The old terms are still supported for compatibility, but the documentation has been updated to use the new ones exclusively.

v0.8.0

This is a feature release which changes the manifest schema by adding support for a `path-prefix:` key in an `import:` mapping, along with some other features and fixes.

- Manifest import mappings now support a `path-prefix:` key, which places the project and its imported repositories in a subdirectory of the workspace. See *Example 3.4: Import into a subdirectory* for an example.
- The west command line application can now also be run using `python3 -m west`. This makes it easier to run west under a particular Python interpreter without modifying the *PATH* environment variable.
- *west manifest –path* prints the absolute path to west.yml
- `west init` now supports an `--mf foo.yml` option, which initializes the workspace using `foo.yml` instead of `west.yml`.
- `west list` now prints the manifest repository's path using the `manifest.path` *configuration option*, which may differ from the `self: path:` value in the manifest data. The old behavior is still available, but requires passing a new `--manifest-path-from-yaml` option.
- Various Python API changes; see west-apis for details.

v0.7.3

This is a bugfix release.

- Fix an error where a failed import could leave the workspace in an unusable state (see [PR #415](https://github.com/zephyrproject-rtos/west/pull/415) for details)

v0.7.2

This is a bugfix and minor feature release.

- Filter out duplicate extension commands brought in by manifest imports
- Fix `west.Manifest.get_projects()` when finding the manifest repository by path

v0.7.1

This is a bugfix and minor feature release.

- `west update --stats` now prints timing for operations which invoke a subprocess, time spent in west's Python process for each project, and total time updating each project.

- `west topdir` always prints a POSIX style path
- minor console output changes

v0.7.0

The main user-visible feature in west 0.7 is the *Manifest Imports* feature. This allows users to load west manifest data from multiple different files, resolving the results into a single logical manifest.

Additional user-visible changes:

- The idea of a "west installation" has been renamed to "west workspace" in this documentation and in the west API documentation. The new term seems to be easier for most people to work with than the old one.
- West manifests now support a *schema version*.
- The "west config" command can now be run outside of a workspace, e.g. to run `west config --global section.key value` to set a configuration option's value globally.
- There is a new *west topdir* command, which prints the root directory of the current west workspace.
- The `west -vv init` command now prints the git operations being performed, and their results.
- The restriction that no project can be named "manifest" is now enforced; the name "manifest" is reserved for the manifest repository, and is usable as such in commands like `west list manifest`, instead of `west list path-to-manifest-repository` being the only way to say that
- It's no longer an error if there is no project named "zephyr". This is part of an effort to make west generally usable for non-Zephyr use cases.
- Various bug fixes.

The developer-visible changes to the west-apis are:

- west.build and west.cmake: deprecated; this is Zephyr-specific functionality and should never have been part of west. Since Zephyr v1.14 LTS relies on it, it will continue to be included in the distribution, but will be removed when that version of Zephyr is obsoleted.
- west.commands:
  - WestCommand.requires_installation: deprecated; use requires_workspace instead
  - WestCommand.requires_workspace: new
  - WestCommand.has_manifest: new
  - WestCommand.manifest: this is now settable
- west.configuration: callers can now identify the workspace directory when reading and writing configuration files
- west.log:
  - msg(): new
- west.manifest:
  - The module now uses the standard logging module instead of west.log
  - QUAL_REFS_WEST: new
  - SCHEMA_VERSION: new
  - Defaults: removed
  - Manifest.as_dict(): new

  - Manifest.as_frozen_yaml(): new
  - Manifest.as_yaml(): new
  - Manifest.from_file() and from_data(): these factory methods are more flexible to use and less reliant on global state
  - Manifest.validate(): new
  - ManifestImportFailed: new
  - ManifestProject: semi-deprecated and will likely be removed later.
  - Project: the constructor now takes a topdir argument
  - Project.format() and its callers are removed. Use f-strings instead.
  - Project.name_and_path: new
  - Project.remote_name: new
  - Project.sha() now captures stderr
  - Remote: removed

West now requires Python 3.6 or later. Additionally, some features may rely on Python dictionaries being insertion-ordered; this is only an implementation detail in CPython 3.6, but it is part of the language specification as of Python 3.7.

v0.6.3

This point release fixes an error in the behavior of the deprecated `west.cmake` module.

v0.6.2

This point release fixes an error in the behavior of `west update --fetch=smart`, introduced in v0.6.1.

All v0.6.1 users must upgrade.

v0.6.1

Warning

Do not use this point release. Make sure to use v0.6.2 instead.

The user-visible features in this point release are:

- The *west update* command has a new `--fetch` command line flag and `update.fetch` *configuration option*. The default value, "smart", skips fetching SHAs and tags which are available locally.
- Better and more consistent error-handling in the `west diff`, `west status`, `west forall`, and `west update` commands. Each of these commands can operate on multiple projects; if a subprocess related to one project fails, these commands now continue to operate on the rest of the projects. All of them also now report a nonzero error code from the west process if any of these subprocesses fails (this was previously not true of `west forall` in particular).
- The *west manifest* command also handles errors better.
- The *west list* command now works even when the projects are not cloned, as long as its format string only requires information which can be read from the manifest file. It still fails if the format string requires data stored in the project repository, e.g. if it includes the `{sha}` format string key.

- Commands and options which operate on git revisions now accept abbreviated SHAs. For example, `west init --mr SHA_PREFIX` now works. Previously, the `--mr` argument needed to be the entire 40 character SHA if it wasn't a branch or a tag.

The developer-visible changes to the west-apis are:

- west.log.banner(): new
- west.log.small_banner(): new
- west.manifest.Manifest.get_projects(): new
- west.manifest.Project.is_cloned(): new
- west.commands.WestCommand instances can now access the parsed Manifest object via a new self.manifest property during the do_run() call. If read, it returns the Manifest object or aborts the command if it could not be parsed.
- west.manifest.Project.git() now has a capture_stderr kwarg

v0.6.0

- No separate bootstrapper

  In west v0.5.x, the program was split into two components, a bootstrapper and a per-installation clone. See Multiple Repository Management in the v1.14 documentation for more details.

  This is similar to how Google's Repo tool works, and lets west iterate quickly at first. It caused confusion, however, and west is now stable enough to be distributed entirely as one piece via PyPI.

  From v0.6.x onwards, all of the core west commands and helper classes are part of the west package distributed via PyPI. This eliminates complexity and makes it possible to import west modules from anywhere in the system, not just extension commands.

- The `selfupdate` command still exists for backwards compatibility, but now simply exits after printing an error message.
- Manifest syntax changes
  - A west manifest file's `projects` elements can now specify their fetch URLs directly, like so:

    ```
    manifest:
      projects:
        - name: example-project-name
          url: https://github.com/example/example-project
    ```

    Project elements with `url` attributes set in this way may not also have `remote` attributes.
  - Project names must be unique: this restriction is needed to support future work, but was not possible in west v0.5.x because distinct projects may have URLs with the same final pathname component, like so:

    ```
    manifest:
      remotes:
        - name: remote-1
          url-base: https://github.com/remote-1
        - name: remote-2
          url-base: https://github.com/remote-2
      projects:
        - name: project
          remote: remote-1
          path: remote-1-project
    ```

(continues on next page)

(continued from previous page)

```
    - name: project
      remote: remote-2
      path: remote-2-project
```

These manifests can now be written with projects that use `url` instead of `remote`, like so:

```
manifest:
  projects:
    - name: remote-1-project
      url: https://github.com/remote-1/project
    - name: remote-2-project
      url: https://github.com/remote-2/project
```

- The `west list` command now supports a `{sha}` format string key
- The default format string for `west list` was changed to `"{name:12} {path:28} {revision:40} {url}"`.
- The command `west manifest --validate` can now be run to load and validate the current manifest file, among other error-handling fixes related to manifest parsing.
- Incompatible API changes were made to west's APIs. Further changes are expected until API stability is declared in west v1.0.
  - The `west.manifest.Project` constructor's `remote` and `defaults` positional arguments are now kwargs. A new `url` kwarg was also added; if given, the `Project` URL is set to that value, and the `remote` kwarg is ignored.
  - `west.manifest.MANIFEST_SECTIONS` was removed. There is only one section now, namely `manifest`. The *sections* kwargs in the `west.manifest.Manifest` factory methods and constructor were also removed.
  - The `west.manifest.SpecialProject` class was removed. Use `west.manifest.ManifestProject` instead.

v0.5.x

West v0.5.x is the first version used widely by the Zephyr Project as part of its v1.14 Long-Term Support (LTS) release. The west v0.5.x documentation is available as part of the Zephyr's v1.14 documentation.

West's main features in v0.5.x are:

- Multiple repository management using Git repositories, including self-update of west itself
- Hierarchical configuration files
- Extension commands

Versions Before v0.5.x

Tags in the west repository before v0.5.x are prototypes which are of historical interest only.

2.11.3 Troubleshooting West

This page covers common issues with west and how to solve them.

west update fetching failures

One good way to troubleshoot fetching issues is to run west update in verbose mode, like this:

```
west -v update
```

The output includes Git commands run by west and their outputs. Look for something like this:

```
=== updating your_project (path/to/your/project):
west.manifest: your_project: checking if cloned
[...other west.manifest logs...]
--- your_project: fetching, need revision SOME_SHA
west.manifest: running 'git fetch ...␣
↪https://github.com/your-username/your_project ...' in /some/directory
```

The git fetch command example in the last line above is what needs to succeed.

One strategy is to go to /path/to/your/project, copy/paste and run the entire git fetch command, then debug from there using the documentation for your credential storage helper.

If you're behind a corporate firewall and may have proxy or other issues, curl -v FETCH_URL (for HTTPS URLs) or ssh -v FETCH_URL (for SSH URLs) may be helpful.

If you can get the git fetch command to run successfully without prompting for a password when you run it directly, you will be able to run west update without entering your password in that same shell.

"'west' is not recognized as an internal or external command, operable program or batch file.'

On Windows, this means that either west is not installed, or your *PATH* environment variable does not contain the directory where pip installed west.exe.

First, make sure you've installed west; see *Installing west*. Then try running west from a new cmd.exe window. If that still doesn't work, keep reading.

You need to find the directory containing west.exe, then add it to your *PATH*. (This *PATH* change should have been done for you when you installed Python and pip, so ordinarily you should not need to follow these steps.)

Run this command in cmd.exe:

```
pip3 show west
```

Then:

1. Look for a line in the output that looks like Location: C:\foo\python\python38\lib\site-packages. The exact location will be different on your computer.
2. Look for a file named west.exe in the scripts directory C:\foo\python\python38\scripts.

   Important

   Notice how lib\site-packages in the pip3 show output was changed to scripts!

3. If you see west.exe in the scripts directory, add the full path to scripts to your *PATH* using a command like this:

   ```
   setx PATH "%PATH%;C:\foo\python\python38\scripts"
   ```

   Do not just copy/paste this command. The scripts directory location will be different on your system.

4. Close your `cmd.exe` window and open a new one. You should be able to run `west`.

"invalid choice: 'build'" (or 'flash', etc.)

If you see an unexpected error like this when trying to run a Zephyr extension command (like *west flash*, *west build*, etc.):

```
$ west build [...]
west: error: argument <command>: invalid choice: 'build' (choose from 'init',␣
↪[...])

$ west flash [...]
west: error: argument <command>: invalid choice: 'flash' (choose from 'init',␣
↪[...])
```

The most likely cause is that you're running the command outside of a *west workspace*. West needs to know where your workspace is to find *Extensions*.

To fix this, you have two choices:

1. Run the command from inside a workspace (e.g. the `zephyrproject` directory you created when you *got started*).

   For example, create your build directory inside the workspace, or run `west flash --build-dir YOUR_BUILD_DIR` from inside the workspace.

2. Set the *ZEPHYR_BASE environment variable* and re-run the west extension command. If set, west will use *ZEPHYR_BASE* to find your workspace.

If you're unsure whether a command is built-in or an extension, run `west help` from inside your workspace. The output prints extension commands separately, and looks like this for mainline Zephyr:

```
$ west help

built-in commands for managing git repositories:
  init:                 create a west workspace
  [...]

other built-in commands:
  help:                 get help for west or a command
  [...]

extension commands from project manifest (path: zephyr):
  build:                compile a Zephyr application
  flash:                flash and run a binary on a board
  [...]
```

"invalid choice: 'post-init'"

If you see this error when running `west init`:

```
west: error: argument <command>: invalid choice: 'post-init'
(choose from 'init', 'update', 'list', 'manifest', 'diff',
'status', 'forall', 'config', 'selfupdate', 'help')
```

Then you have an old version of west installed, and are trying to use it in a workspace that requires a more recent version.

The easiest way to resolve this issue is to upgrade west and retry as follows:

1. Install the latest west with the `-U` option for `pip3 install` as shown in *Installing west*.

2. Back up any contents of `zephyrproject/.west/config` that you want to save. (If you don't have any configuration options set, it's safe to skip this step.)
3. Completely remove the `zephyrproject/.west` directory (if you don't, you will get the "already in a workspace" error message discussed next).
4. Run `west init` again.

2.11.4 Basics

This page introduces west's basic concepts and provides references to further reading.

West's built-in commands allow you to work with *projects* (Git repositories) under a common *workspace* directory.

West works in the following manner: the `west init` command creates the *west workspace*, and clones the *manifest repo*, while the `west update` command initially clones, and later updates, the *projects* listed in the manifest in the workspace.

Example workspace

If you've followed the *Getting Started Guide*, your local *west workspace*, which in this case is the folder named `zephyrproject` as well as all its subfolders, looks like this:

```
zephyrproject/                # west topdir
├── .west/                    # marks the location of the topdir
│   └── config                # per-workspace local configuration file
│
│   # The manifest repository, never modified by west after creation:
├── zephyr/                   # .git/ repo
│   ├── west.yml              # manifest file
│   └── [... other files ...]
│
│   # Projects managed by west:
├── modules/
│   └── lib/
│       └── zcbor/            # .git/ project
├── tools/
│   └── net-tools/            # .git/ project
└── [ ... other projects ...]
```

Workspace concepts

Here are the basic concepts you should understand about this structure. Additional details are in *Workspaces*.

topdir
: Above, `zephyrproject` is the name of the workspace's top level directory, or *topdir*. (The name `zephyrproject` is just an example – it could be anything, like `z`, `my-zephyr-workspace`, etc.)

  You'll typically create the topdir and a few other files and directories using *west init*.

.west directory
: The topdir contains the `.west` directory. When west needs to find the topdir, it searches for `.west`, and uses its parent directory. The search starts from the current working directory (and starts again from the location in the *ZEPHYR_BASE* environment variable as a fallback if that fails).

configuration file
: The file `.west/config` is the workspace's *local configuration file*.

manifest repository
Every west workspace contains exactly one *manifest repository*, which is a Git repository containing a *manifest file*. The location of the manifest repository is given by the *manifest.path configuration option* in the local configuration file.

For upstream Zephyr, `zephyr` is the manifest repository, but you can configure west to use any Git repository in the workspace as the manifest repository. The only requirement is that it contains a valid manifest file. See *Topologies supported* for information on other options, and *West Manifests* for details on the manifest file format.

manifest file
The manifest file is a YAML file that defines *projects*, which are the additional Git repositories in the workspace managed by west. The manifest file is named `west.yml` by default; this can be overridden using the `manifest.file` local configuration option.

You use the *west update* command to update the workspace's projects based on the contents of the manifest file.

projects
Projects are Git repositories managed by west. Projects are defined in the manifest file and can be located anywhere inside the workspace. In the above example workspace, `zcbor` and `net-tools` are projects.

By default, the Zephyr *build system* uses west to get the locations of all the projects in the workspace, so any code they contain can be used as *Modules (External projects)*. Note however that modules and projects *are conceptually different*.

extensions
Any repository known to west (either the manifest repository or any project repository) can define *Extensions*. Extensions are extra west commands you can run when using that workspace.

The zephyr repository uses this feature to provide Zephyr-specific commands like *west build*. Defining these as extensions keeps west's core agnostic to the specifics of any workspace's Zephyr version, etc.

ignored files
A workspace can contain additional Git repositories or other files and directories not managed by west. West basically ignores anything in the workspace except `.west`, the manifest repository, and the projects specified in the manifest file.

west init and west update

The two most important workspace-related commands are `west init` and `west update`.

`west init` basics This command creates a west workspace.

> **Important**
>
> West doesn't change your manifest repository contents after `west init` is run. Use ordinary Git commands to pull new versions, etc.

You will typically run it once, like this:

```
west init -m https://github.com/zephyrproject-rtos/zephyr --mr v2.5.0 zephyrproject
```

This will:

1. Create the topdir, `zephyrproject`, along with `.west` and `.west/config` inside it
2. Clone the manifest repository from https://github.com/zephyrproject-rtos/zephyr, placing it into `zephyrproject/zephyr`

3. Check out the `v2.5.0` git tag in your local zephyr clone
4. Set `manifest.path` to `zephyr` in `.west/config`
5. Set `manifest.file` to `west.yml`

Your workspace is now almost ready to use; you just need to run `west update` to clone the rest of the projects into the workspace to finish.

For more details, see *west init*.

`west update` basics This command makes sure your workspace contains Git repositories matching the projects in the manifest file.

> **Important**
>
> Whenever you check out a different revision in your manifest repository, you should run `west update` to make sure your workspace contains the project repositories the new revision expects.

The `west update` command reads the manifest file's contents by:

1. Finding the topdir. In the `west init` example above, that means finding `zephyrproject`.
2. Loading `.west/config` in the topdir to read the `manifest.path` (e.g. `zephyr`) and `manifest.file` (e.g. `west.yml`) options.
3. Loading the manifest file given by these options (e.g. `zephyrproject/zephyr/west.yml`).

It then uses the manifest file to decide where missing projects should be placed within the workspace, what URLs to clone them from, and what Git revisions should be checked out locally. Project repositories which already exist are updated in place by fetching and checking out their respective Git revisions in the manifest file.

For more details, see *west update*.

Other built-in commands

See *Built-in commands*.

Zephyr Extensions

See the following pages for information on Zephyr's extension commands:

- *Building, Flashing and Debugging*
- *Signing Binaries*
- *Additional Zephyr extension commands*
- *Enabling shell completion*

Troubleshooting

See *Troubleshooting West*.

2.11.5 Built-in commands

This page describes west's built-in commands, some of which were introduced in *Basics*, in more detail.

Some commands are related to Git commands with the same name, but operate on the entire workspace. For example, `west diff` shows local changes in multiple Git repositories in the workspace.

Some commands take projects as arguments. These arguments can be project names as specified in the manifest file, or (as a fallback) paths to them on the local file system. Omitting project arguments to commands which accept them (such as `west list`, `west forall`, etc.) usually defaults to using all projects in the manifest file plus the manifest repository itself.

For additional help, run `west <command> -h` (e.g. `west init -h`).

west init

This command creates a west workspace. It can be used in two ways:

1. Cloning a new manifest repository from a remote URL
2. Creating a workspace around an existing local manifest repository

Option 1: to clone a new manifest repository from a remote URL, use:

```
west init [-m URL] [--mr REVISION] [--mf FILE] [directory]
```

The new workspace is created in the given `directory`, creating a new `.west` inside this directory. You can give the manifest URL using the `-m` switch, the initial revision to check out using `--mr`, and the location of the manifest file within the repository using `--mf`.

For example, running:

```
west init -m https://github.com/zephyrproject-rtos/zephyr --mr v1.14.0 zp
```

would clone the upstream official zephyr repository into `zp/zephyr`, and check out the `v1.14.0` release. This command creates `zp/.west`, and set the `manifest.path` *configuration option* to `zephyr` to record the location of the manifest repository in the workspace. The default manifest file location is used.

The `-m` option defaults to `https://github.com/zephyrproject-rtos/zephyr`. The `--mf` option defaults to `west.yml`. Since west v0.10.1, west will use the default branch in the manifest repository unless the `--mr` option is used to override it. (In prior versions, `--mr` defaulted to `master`.)

If no `directory` is given, the current working directory is used.

Option 2: to create a workspace around an existing local manifest repository, use:

```
west init -l [--mf FILE] directory
```

This creates `.west` **next to** `directory` in the file system, and sets `manifest.path` to `directory`.

As above, `--mf` defaults to `west.yml`.

Reconfiguring the workspace:

If you change your mind later, you are free to change `manifest.path` and `manifest.file` using *west config* after running `west init`. Just be sure to run `west update` afterwards to update your workspace to match the new manifest file.

west update

```
west update [-f {always,smart}] [-k] [-r]
            [--group-filter FILTER] [--stats] [PROJECT ...]
```

Which projects are updated:

By default, this command parses the manifest file, usually `west.yml`, and updates each project specified there. If your manifest uses *project groups*, then only the active projects are updated.

To operate on a subset of projects only, give PROJECT argument(s). Each PROJECT is either a project name as given in the manifest file, or a path that points to the project within the workspace. If you specify projects explicitly, they are updated regardless of whether they are active.

Project update procedure:

For each project that is updated, this command:

1. Initializes a local Git repository for the project in the workspace, if it does not already exist
2. Inspects the project's `revision` field in the manifest, and fetches it from the remote if it is not already available locally
3. Sets the project's *manifest-rev* branch to the commit specified by the revision in the previous step
4. Checks out `manifest-rev` in the local working copy as a detached HEAD (see *west update detached HEADs* for details about this choice)
5. If the manifest file specifies a *submodules* key for the project, recursively updates the project's submodules as described below.

To avoid unnecessary fetches, `west update` will not fetch project `revision` values which are Git SHAs or tags that are already available locally. This is the behavior when the `-f` (`--fetch`) option has its default value, `smart`. To force this command to fetch from project remotes even if the revisions appear to be available locally, either use `-f always` or set the `update.fetch` *configuration option* to `always`. SHAs may be given as unique prefixes as long as they are acceptable to Git[1].

If the project `revision` is a Git ref that is neither a tag nor a SHA (i.e. if the project is tracking a branch), `west update` always fetches, regardless of `-f` and `update.fetch`.

Some branch names might look like short SHAs, like `deadbeef`. West treats these like SHAs. You can disambiguate by prefixing the `revision` value with `refs/heads/`, e.g. `revision: refs/heads/deadbeef`.

For safety, `west update` uses `git checkout --detach` to check out a detached `HEAD` at the manifest revision for each updated project, leaving behind any branches which were already checked out. This is typically a safe operation that will not modify any of your local branches.

However, if you had added some local commits onto a previously detached `HEAD` checked out by west, then git will warn you that you've left behind some commits which are no longer referred to by any branch. These may be garbage-collected and lost at some point in the future. To avoid this if you have local commits in the project, make sure you have a local branch checked out before running `west update`.

If you would rather rebase any locally checked out branches instead, use the `-r` (`--rebase`) option.

If you would like `west update` to keep local branches checked out as long as they point to commits that are descendants of the new `manifest-rev`, use the `-k` (`--keep-descendants`) option.

Note

`west update --rebase` will fail in projects that have git conflicts between your branch and new commits brought in by the manifest. You should immediately resolve these conflicts as you usually do with `git`, or you can use `git -C <project_path> rebase --abort` to ignore incoming changes for the moment.

With a clean working tree, a plain `west update` never fails because it does not try to hold on to your commits and simply leaves them aside.

`west update --keep-descendants` offers an intermediate option that never fails either but does not treat all projects the same:

[1] West may fetch all refs from the Git server when given a SHA as a revision. This is because some Git servers have historically not allowed fetching SHAs directly.

- in projects where your branch diverged from the incoming commits, it does not even try to rebase and leaves your branches behind just like a plain `west update` does;
- in all other projects where no rebase or merge is needed it keeps your branches in place.

One-time project group manipulation:

The `--group-filter` option can be used to change which project groups are enabled or disabled for the duration of a single `west update` command. See *Project Groups* for details on the project group feature.

The `west update` command behaves as if the `--group-filter` option's value were appended to the `manifest.group-filter` *configuration option*.

For example, running `west update --group-filter=+foo,-bar` would behave the same way as if you had temporarily appended the string `"+foo,-bar"` to the value of `manifest.group-filter`, run `west update`, then restored `manifest.group-filter` to its original value.

Note that using the syntax `--group-filter=VALUE` instead of `--group-filter VALUE` avoids issues parsing command line options if you just want to disable a single group, e.g. `--group-filter=-bar`.

Submodule update procedure:

If a project in the manifest has a `submodules` key, the submodules are updated as follows, depending on the value of the `submodules` key.

If the project has `submodules: true`, west first synchronizes the project's submodules with:

```
git submodule sync --recursive
```

West then runs one of the following in the project repository, depending on whether you run `west update` with the `--rebase` option or without it:

```
# without --rebase, e.g. "west update":
git submodule update --init --checkout --recursive

# with --rebase, e.g. "west update --rebase":
git submodule update --init --rebase --recursive
```

Otherwise, the project has `submodules: <list-of-submodules>`. In this case, west synchronizes the project's submodules with:

```
git submodule sync --recursive -- <submodule-path>
```

Then it updates each submodule in the list as follows, depending on whether you run `west update` with the `--rebase` option or without it:

```
# without --rebase, e.g. "west update":
git submodule update --init --checkout --recursive <submodule-path>

# with --rebase, e.g. "west update --rebase":
git submodule update --init --rebase --recursive <submodule-path>
```

The `git submodule sync` commands are skipped if the `update.sync-submodules` *Configuration* option is false.

Other project commands

West has a few more commands for managing the projects in the workspace, which are summarized here. Run `west <command> -h` for detailed help.

- `west compare`: compare the state of the workspace against the manifest
- `west diff`: run `git diff` in local project repositories
- `west forall`: run an arbitrary command in local project repositories
- `west grep`: search for patterns in local project repositories
- `west list`: print a line of information about each project in the manifest, according to a format string
- `west manifest`: manage the manifest file. See *Manifest Command*.
- `west status`: run `git status` in local project repositories

Other built-in commands

Finally, here is a summary of other built-in commands.

- `west config`: get or set *configuration options*
- `west topdir`: print the top level directory of the west workspace
- `west help`: get help about a command, or print information about all commands in the workspace, including *Extensions*

2.11.6 Workspaces

This page describes the *west workspace* concept introduced in *Basics* in more detail.

The `manifest-rev` branch

West creates and controls a Git branch named `manifest-rev` in each project. This branch points to the revision that the manifest file specified for the project at the time *west update* was last run. Other workspace management commands may use `manifest-rev` as a reference point for the upstream revision as of this latest update. Among other purposes, the `manifest-rev` branch allows the manifest file to use SHAs as project revisions.

Although `manifest-rev` is a normal Git branch, west will recreate and/or reset it on the next update. For this reason, it is **dangerous** to check it out or otherwise modify it yourself. For instance, any commits you manually add to this branch may be lost the next time you run `west update`. Instead, check out a local branch with another name, and either rebase it on top of a new `manifest-rev`, or merge `manifest-rev` into it.

> **Note**
>
> West does not create a `manifest-rev` branch in the manifest repository, since west does not manage the manifest repository's branches or revisions.

The `refs/west/*` Git refs

West also reserves all Git refs that begin with `refs/west/` (such as `refs/west/foo`) for itself in local project repositories. Unlike `manifest-rev`, these refs are not regular branches. West's behavior here is an implementation detail; users should not rely on these refs' existence or behavior.

Developing in a git repository

`west` "projects" in a workspace are ordinary git repositories. Left to itself, west only creates and updates the `manifest-rev` branch and the refs under `refs/west/`; the branches you create are yours, and west only rebases them when you ask it to with `west update --rebase`.

That is also why a plain *west update* leaves you on a detached `HEAD`: it checks out the new `manifest-rev` and leaves your branches exactly where they are. See *west update detached HEADs* for the rationale.

> **Note**
>
> Zephyr *modules* and west projects are not the same thing, even though west projects are often also modules; see *Modules vs west projects*. The git workflow below applies to all west projects whether they are Zephyr modules or not.

A workflow which works well when you have local changes is:

1. Create a branch in the repository and commit to it as usual:

   ```
   git -C <repository-path> switch --create <your-branch>
   ```

2. Update your workspace with `west update --rebase`, then look at what changed:

   ```
   west update --rebase
   west compare
   ```

   Your branch stays checked out and is rebased onto the new `manifest-rev`. This is the only time west touches your own branch. If it conflicts, the command fails and you resolve it with git as usual.

3. If you would rather not have west rebase your branch, a plain `west update` will switch away from it instead (unless this causes git conflicts). `west update --keep-descendants` is a third option which never fails. See the `checked out branch behavior` option group in `west update --help`, and *west update* for all the details.

Committing on a detached `HEAD` Do not commit on a detached `HEAD` left behind by `west update`: git warns you that those commits are not reachable from any branch and they may be garbage-collected later. Create a branch for your work instead; or use an alternative git client purposely designed for "branchless" work, like for instance JJ.

Private repositories

You can use west to fetch from private repositories. There is nothing west-specific about this.

The `west update` command essentially runs `git fetch YOUR_PROJECT_URL` when a project's `manifest-rev` branch must be updated to a newly fetched commit. It's up to your environment to make sure the fetch succeeds.

You can either enter the password manually or use any of the credential helpers built in to Git. Since Git has credential storage built in, there is no need for a west-specific feature.

The following sections cover common cases for running `west update` without having to enter your password, as well as how to troubleshoot issues.

Fetching via HTTPS On Windows when fetching from GitHub, recent versions of Git prompt you for your GitHub password in a graphical window once, then store it for future use (in a default installation). Passwordless fetching from GitHub should therefore work "out of the box" on Windows after you have done it once.

In general, you can store your credentials on disk using the "store" git credential helper. See the git-credential-store manual page for details.

To use this helper for all the repositories in your workspace, run:

```
west forall -c "git config credential.helper store"
```

To use this helper on just the projects `foo` and `bar`, run:

```
west forall -c "git config credential.helper store" foo bar
```

To use this helper by default on your computer, run:

```
git config --global credential.helper store
```

On GitHub, you can set up a personal access token to use in place of your account password. (This may be required if your account has two-factor authentication enabled, and may be preferable to storing your account password in plain text even if two-factor authentication is disabled.)

You can use the Git credential store to authenticate with a GitHub PAT (Personal Access Token) like so:

```
echo "https://x-access-token:$GH_TOKEN@github.com" >> ~/.git-credentials
```

If you don't want to store any credentials on the file system, you can store them in memory temporarily using git-credential-cache instead.

If you setup fetching via SSH, you can use Git URL rewrite feature. The following command instructs Git to use SSH URLs for GitHub instead of HTTPS ones:

```
git config --global url."git@github.com:".insteadOf "https://github.com/"
```

Fetching via SSH If your SSH key has no password, fetching should just work. If it does have a password, you can avoid entering it manually every time using ssh-agent.

On GitHub, see Connecting to GitHub with SSH for details on configuration and key creation.

Project locations

Projects can be located anywhere inside the workspace, but they may not "escape" it.

In other words, project repositories need not be located in subdirectories of the manifest repository or as immediate subdirectories of the topdir. However, projects must have paths inside the workspace.

You may replace a project's repository directory within the workspace with a symbolic link to elsewhere on your computer, but west will not do this for you.

Topologies supported

The following are example source code topologies supported by west.

- T1: star topology, zephyr is the manifest repository
- T2: star topology, a Zephyr application is the manifest repository
- T3: forest topology, freestanding manifest repository

T1: Star topology, zephyr is the manifest repository

- The zephyr repository acts as the central repository and specifies its *Modules (External projects)* in its `west.yml`
- Analogy with existing mechanisms: Git submodules with zephyr as the super-project

This is the default. See *Workspace concepts* for how mainline Zephyr is an example of this topology.

T2: Star topology, application is the manifest repository

- Useful for those focused on a single application
- A repository containing a Zephyr application acts as the central repository and names other projects required to build it in its `west.yml`. This includes the zephyr repository and any modules.
- Analogy with existing mechanisms: Git submodules with the application as the superproject, zephyr and other projects as submodules

A workspace using this topology looks like this:

```
west-workspace/
|
├── application/         # .git/     |
|   ├── CMakeLists.txt               |
|   ├── prj.conf                     |  never modified by west
|   ├── src/                         |
|   |   └── main.c                   |
|   └── west.yml         # main manifest with optional import(s) and override(s)
|                                    |
├── modules/
|   └── lib/
|       └── zcbor/       # .git/ project from either the main manifest or some␣
↪import.
|
└── zephyr/              # .git/ project
    └── west.yml         # This can be partially imported with lower precedence or␣
↪ignored.
                         # Only the 'manifest-rev' version can be imported.
```

Here is an example `application/west.yml` which uses *Manifest Imports*, available since west 0.7, to import Zephyr v2.5.0 and its modules into the application manifest file:

```yaml
# Example T2 west.yml, using manifest imports.
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.5.0
      import: true
  self:
    path: application
```

You can still selectively "override" individual Zephyr modules if you use `import:` in this way; see *Example 1.3: Downstream of a Zephyr release, with module fork* for an example.

Another way to do the same thing is to copy/paste `zephyr/west.yml` to `application/west.yml`, adding an entry for the zephyr project itself, like this:

```yaml
# Equivalent to the above, but with manually maintained Zephyr modules.
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  defaults:
    remote: zephyrproject-rtos
```

(continues on next page)

(continued from previous page)

```yaml
  projects:
    - name: zephyr
      revision: v2.5.0
      west-commands: scripts/west-commands.yml
    - name: net-tools
      revision: some-sha-goes-here
      path: tools/net-tools
    # ... other Zephyr modules go here ...
  self:
    path: application
```

(The `west-commands` is there for *Building, Flashing and Debugging* and other Zephyr-specific *Extensions*. It's not necessary when using `import`.)

The main advantage to using `import` is not having to track the revisions of imported projects separately. In the above example, using `import` means Zephyr's *module* versions are automatically determined from the `zephyr/west.yml` revision, instead of having to be copy/pasted (and maintained) on their own.

T3: Forest topology

- Useful for those supporting multiple independent applications or downstream distributions with no "central" repository
- A dedicated manifest repository which contains no Zephyr source code, and specifies a list of projects all at the same "level"
- Analogy with existing mechanisms: Google repo-based source distribution

A workspace using this topology looks like this:

```
west-workspace/
├── app1/               # .git/ project
│   ├── CMakeLists.txt
│   ├── prj.conf
│   └── src/
│       └── main.c
├── app2/               # .git/ project
│   ├── CMakeLists.txt
│   ├── prj.conf
│   └── src/
│       └── main.c
├── manifest-repo/      # .git/ never modified by west
│   └── west.yml        # main manifest with optional import(s) and override(s)
├── modules/
│   └── lib/
│       └── zcbor/      # .git/ project from either the main manifest or
│                       #       from some import
│
└── zephyr/             # .git/ project
    └── west.yml        # This can be partially imported with lower precedence or␣
↪ignored.
                        # Only the 'manifest-rev' version can be imported.
```

Here is an example T3 `manifest-repo/west.yml` which uses *Manifest Imports*, available since west 0.7, to import Zephyr v2.5.0 and its modules, then add the `app1` and `app2` projects:

```yaml
manifest:
  remotes:
```

(continues on next page)

(continued from previous page)

```
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
    - name: your-git-server
      url-base: https://git.example.com/your-company
  defaults:
    remote: your-git-server
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v2.5.0
      import: true
    - name: app1
      revision: SOME_SHA_OR_BRANCH_OR_TAG
    - name: app2
      revision: ANOTHER_SHA_OR_BRANCH_OR_TAG
  self:
    path: manifest-repo
```

You can also do this "by hand" by copy/pasting `zephyr/west.yml` as shown *above* for the T2 topology, with the same caveats.

Not supported: workspace topdir as .git repository

Some users have asked for support making the workspace *topdir* a git repository, like this example:

```
my-workspace/                  # workspace topdir
├── .git/                      # puts the entire workspace in a git repository
├── .west/                     # marks the location of the topdir
└── [ ... other projects ...]
```

This is **not** an officially supported topology. As a design decision, west assumes that the workspace topdir itself is not a git repository.

You may be able to make something like this "work" for yourself and your own goals. However, future versions of west might contain changes which can "break" your setup.

2.11.7 West Manifests

This page contains detailed information about west's multiple repository model, manifest files, and the `west manifest` command. For API documentation on the `west.manifest` module, see west-apis-manifest. For a more general introduction and command overview, see *Basics*.

Multiple Repository Model

West's view of the repositories in a *west workspace*, and their history, looks like the following figure (though some parts of this example are specific to upstream Zephyr's use of west):

The history of the manifest repository is the line of Git commits which is "floating" on top of the gray plane. Parent commits point to child commits using solid arrows. The plane below contains the Git commit history of the repositories in the workspace, with each project repository boxed in by a rectangle. Parent/child commit relationships in each repository are also shown with solid arrows.

The commits in the manifest repository (again, for upstream Zephyr this is the zephyr repository itself) each have a manifest file. The manifest file in each commit specifies the corresponding commits which it expects in each of the project repositories. This relationship is shown using

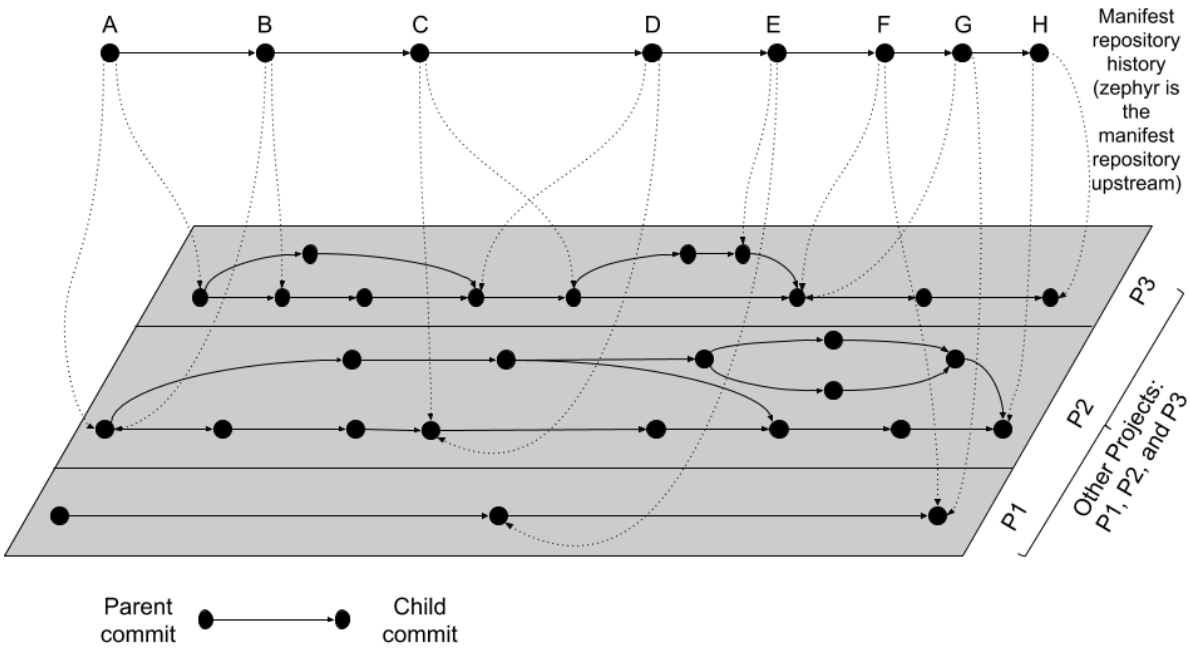


Fig. 3: West multi-repo history

dotted line arrows in the diagram. Each dotted line arrow points from a commit in the manifest repository to a corresponding commit in a project repository.

Notice the following important details:

- Projects can be added (like `P1` between manifest repository commits `D` and `E`) and removed (`P2` between the same manifest repository commits)
- Project and manifest repository histories don't have to move forwards or backwards together:
  - `P2` stays the same from `A → B`, as do `P1` and `P3` from `F → G`.
  - `P3` moves forward from `A → B`.
  - `P3` moves backward from `C → D`.

  One use for moving backward in project history is to "revert" a regression by going back to a revision before it was introduced.
- Project repository commits can be "skipped": `P3` moves forward multiple commits in its history from `B → C`.
- In the above diagram, no project repository has two revisions "at the same time": every manifest file refers to exactly one commit in the projects it cares about. This can be relaxed by using a branch name as a manifest revision, at the cost of being able to bisect manifest repository history.

Manifest Files

West manifests are YAML files. Manifests have a top-level `manifest` section with some subsections, like this:

```
manifest:
  remotes:
    # short names for project URLs
  projects:
```

(continues on next page)

(continued from previous page)

```
    # a list of projects managed by west
  defaults:
    # default project attributes
  self:
    # configuration related to the manifest repository itself,
    # i.e. the repository containing west.yml
  version: "<schema-version>"
  group-filter:
    # a list of project groups to enable or disable
```

In YAML terms, the manifest file contains a mapping, with a `manifest` key. Any other keys and their contents are ignored (west v0.5 also required a `west` key, but this is ignored starting with v0.6).

The manifest contains subsections, like `defaults`, `remotes`, `projects`, and `self`. In YAML terms, the value of the `manifest` key is also a mapping, with these "subsections" as keys. As of west v0.10, all of these "subsection" keys are optional.

The `projects` value is a list of repositories managed by west and associated metadata. We'll discuss it soon, but first we will describe the `remotes` section, which can be used to save typing in the `projects` list.

Remotes The `remotes` subsection contains a sequence which specifies the base URLs where projects can be fetched from.

Each `remotes` element has a name and a "URL base". These are used to form the complete Git fetch URL for each project. A project's fetch URL can be set by appending a project-specific path onto a remote URL base. (As we'll see below, projects can also specify their complete fetch URLs.)

For example:

```
manifest:
  # ...
  remotes:
    - name: remote1
      url-base: https://git.example.com/base1
    - name: remote2
      url-base: https://git.example.com/base2
```

The `remotes` keys and their usage are in the following table.

Table 2: remotes keys

Key	Description
`name`	Mandatory; a unique name for the remote.
`url-base`	A prefix that is prepended to the fetch URL for each project with this remote.

Above, two remotes are given, with names `remote1` and `remote2`. Their URL bases are respectively `https://git.example.com/base1` and `https://git.example.com/base2`. You can use SSH URL bases as well; for example, you might use `git@example.com:base1` if `remote1` supported Git over SSH as well. Anything acceptable to Git will work.

Projects The `projects` subsection contains a sequence describing the project repositories in the west workspace. Every project has a unique name. You can specify what Git remote URLs to use when cloning and fetching the projects, what revisions to track, and where the project should be stored on the local file system. Note that west projects *are different from modules*.

Here is an example. We'll assume the `remotes` given above.

```yaml
manifest:
  # [... same remotes as above...]
  projects:
    - name: proj1
      description: the first example project
      remote: remote1
      path: extra/project-1
    - name: proj2
      description: |
        A multi-line description of the second example
        project.
      repo-path: my-path
      remote: remote2
      revision: v1.3
    - name: proj3
      url: https://github.com/user/project-three
      revision: abcde413a111
    - name: proj4
      url: https://github.com/user/project-four
      revision: pull/69/head # GitHub Pull Request
```

In this manifest:

- `proj1` has remote `remote1`, so its Git fetch URL is `https://git.example.com/base1/proj1`. The remote `url-base` is appended with a `/` and the project `name` to form the URL.

  Locally, this project will be cloned at path `extra/project-1` relative to the west workspace's root directory, since it has an explicit `path` attribute with this value.

  Since the project has no `revision` specified, `master` is used by default. The current tip of this branch will be fetched and checked out as a detached `HEAD` when west next updates this project.

- `proj2` has a `remote` and a `repo-path`, so its fetch URL is `https://git.example.com/base2/my-path`. The `repo-path` attribute, if present, overrides the default `name` when forming the fetch URL.

  Since the project has no `path` attribute, its `name` is used by default. It will be cloned into a directory named `proj2`. The commit pointed to by the `v1.3` tag will be checked out when west updates the project.

- `proj3` has an explicit `url`, so it will be fetched from `https://github.com/user/project-three`.

  Its local path defaults to its name, `proj3`. Commit `abcde413a111` will be checked out when it is next updated.

The available project keys and their usage are in the following table. Sometimes we'll refer to the `defaults` subsection; it will be described next.

Table 3: projects elements keys

Key(s)	Description
`name`	Mandatory; a unique name for the project. The name cannot be one of the reserved values "west" or "manifest". The name must be unique in the manifest file.
`description`	Optional, an informational description of the project. Added in west v1.2.0.
`remote`, `url`	Mandatory (one of the two, but not both). If the project has a `remote`, that remote's `url-base` will be combined with the project's `name` (or `repo-path`, if it has one) to form the fetch URL instead. If the project has a `url`, that's the complete fetch URL for the remote Git repository. If the project has neither, the `defaults` section must specify a `remote`, which will be used as the project's remote. Otherwise, the manifest is invalid.
`repo-path`	Optional. If given, this is concatenated on to the remote's `url-base` instead of the project's `name` to form its fetch URL. Projects may not have both `url` and `repo-path` attributes.
`revision`	Optional. The Git revision that `west update` should check out. This will be checked out as a detached HEAD by default, to avoid conflicting with local branch names. If not given, the `revision` value from the `defaults` subsection will be used if present. A project revision can be any fetchable git reference: branch, tag, SHA, pull request,... The default `revision` is `master` if not otherwise specified. Using `HEAD~0`[1] as the `revision` will cause west to keep the current state of the project.
`path`	Optional. Relative path specifying where to clone the repository locally, relative to the top directory in the west workspace. If missing, the project's `name` is used as a directory name.
`clone-depth`	Optional. If given, a positive integer which creates a shallow history in the cloned repository limited to the given number of commits. This can only be used if the `revision` is a branch or tag.
`west-commands`	Optional. If given, a relative path to a YAML file within the project which describes additional west commands provided by that project. This file is named `west-commands.yml` by convention. See *Extensions* for details.
`import`	Optional. If `true`, imports projects from manifest files in the given repository into the current manifest. See *Manifest Imports* for details.
`groups`	Optional, a list of groups the project belongs to. See *Project Groups* for details.
`submodules`	Optional. You can use this to make `west update` also update Git submodules defined by the project. See *Git Submodules in Projects* for details.
`userdata`	Optional. The value is an arbitrary YAML value. See *Repository user data*.

Defaults The `defaults` subsection can provide default values for project attributes. In particular, the default remote name and revision can be specified here. Another way to write the same manifest we have been describing so far using `defaults` is:

```
manifest:
  defaults:
    remote: remote1
    revision: v1.3

  remotes:
```

(continues on next page)

[1] In git, HEAD is a reference, whereas HEAD~<n> is a valid revision but not a reference. West fetches references, such as refs/heads/main or HEAD, and commits not available locally, but will not fetch commits if they are already available. HEAD~0 is resolved to a specific commit that is locally available, and therefore west will simply checkout the locally available commit, identified by HEAD~0.

(continued from previous page)

```
    - name: remote1
      url-base: https://git.example.com/base1
    - name: remote2
      url-base: https://git.example.com/base2

  projects:
    - name: proj1
      description: the first example project
      path: extra/project-1
      revision: master
    - name: proj2
      description: |
        A multi-line description of the second example
        project.
      repo-path: my-path
      remote: remote2
    - name: proj3
      url: https://github.com/user/project-three
      revision: abcde413a111
```

The available `defaults` keys and their usage are in the following table.

Table 4: defaults keys

Key	Description
`remote`	Optional. This will be used for a project's `remote` if it does not have a `url` or `remote` key set.
`revision`	Optional. This will be used for a project's `revision` if it does not have one set. If not given, the default is `master`.

Self The `self` subsection can be used to control the manifest repository itself.

As an example, let's consider this snippet from the zephyr repository's `west.yml`:

```
manifest:
  # ...
  self:
    path: zephyr
    west-commands: scripts/west-commands.yml
```

This ensures that the zephyr repository is cloned into path `zephyr`, though as explained above that would have happened anyway if cloning from the default manifest URL, `https://github.com/zephyrproject-rtos/zephyr`. Since the zephyr repository does contain extension commands, its `self` entry declares the location of the corresponding `west-commands.yml` relative to the repository root.

The available `self` keys and their usage are in the following table.

Table 5: self keys

Key	Description
`path`	Optional. The path `west init` should clone the manifest repository into, relative to the west workspace topdir. If not given, the basename of the path component in the manifest repository URL will be used by default. For example, if the URL is `https://git.example.com/project-repo`, the manifest repository would be cloned to the directory `project-repo`.
`west-commands`	Optional. This is analogous to the same key in a project sequence element.
`import`	Optional. This is also analogous to the `projects` key, but allows importing projects from other files in the manifest repository. See *Manifest Imports*.

Version The `version` subsection declares that the manifest file uses features which were introduced in some version of west. Attempts to load the manifest with older versions of west will fail with an error message that explains the minimum required version of west which is needed.

Here is an example:

```
manifest:
  # Marks that this file uses version 0.10 of the west manifest
  # file format.
  #
  # An attempt to load this manifest file with west v0.8.0 will
  # fail with an error message saying that west v0.10.0 or
  # later is required.
  version: "0.10"
```

The pykwalify schema `manifest-schema.yml` in the west source code repository is used to validate the manifest section.

Here is a table with the valid `version` values, along with information about the manifest file features that were introduced in that version.

`version`	New features
`"0.7"`	Initial support for the `version` feature. All manifest file features that are not otherwise mentioned in this table were introduced in west v0.7.0 or earlier.
`"0.8"`	Support for `import: path-prefix:` (*Option 3: Mapping*)
`"0.9"`	**Use of west v0.9.x is discouraged.** This schema version is provided to allow users to explicitly request compatibility with west *v0.9.0*. However, west *v0.10.0* and later have incompatible behavior for features that were introduced in west v0.9.0. You should ignore version "0.9" if possible.
`"0.10"`	Support for: • `submodules:` in `projects:` (*Git Submodules in Projects*) • `manifest: group-filter:`, and `groups:` in `projects:` (*Project Groups*) • The `import:` feature now supports `allowlist:` and `blocklist:`; these are respectively recommended as replacements for older names as part of a general Zephyr-wide inclusive language change. The older key names are still supported for backwards compatibility. (*Manifest Imports*, *Option 3: Mapping*)
`"0.12"`	Support for `userdata:` in `projects:` (*Repository user data*)
`"0.13"`	Support for `self: userdata:` (*Repository user data*)
`"1.0"`	Identical to `"0.13"`, but available for use by users that do not wish to use a `"0.x"` version field.
`"1.2"`	Support for `description:` in `projects:` (*Projects*)

Note

Versions of west without any new features in the manifest file format do not change the list of valid `version` values. For example, `version: "0.11"` is **not** valid, because west v0.11.x did not introduce new manifest file format features.

Quoting the `version` value as shown above forces the YAML parser to treat it as a string. Without quotes, `0.10` in YAML is just the floating point value `0.1`. You can omit the quotes if the value is the same when cast to string, but it's best to include them. Always use quotes if you're not sure.

If you do not include a `version` in your manifest, each new release of west assumes that it should try to load it using the features that were available in that release. This may result in error messages that are harder to understand if that version of west is too old to load the manifest.

Group-filter See *Project Groups*.

Active and Inactive Projects

Projects defined in the west manifest can be *inactive* or *active*. The difference is that an inactive project is generally ignored by west. For example, `west update` will not update inactive projects, and `west list` will not print information about them by default. As another example, any *Manifest Imports* in an inactive project will be ignored by west.

There are two ways to make a project inactive:

1. Using the `manifest.project-filter` configuration option. If a project is made active or inactive using this option, then the rules related to making a project inactive using its `groups:` are ignored. That is, if a regular expression in `manifest.project-filter` applies to a project, the project's groups have no effect on whether it is active or inactive.

   See the entry for this option in *Built-in Configuration Options* for details.

2. Otherwise, if a project has groups, and they are all disabled, then the project is inactive.

   See the following section for details.

Project Groups

You can use the `groups` and `group-filter` keys briefly described *above* to place projects into groups, and to enable or disable groups.

For example, this lets you run a `west forall` command only on the projects in the group by using `west forall --group`. This can also let you make projects inactive; see the previous section for more information on inactive projects.

The next section introduces project groups. The following section describes *Enabled and Disabled Project Groups*. There are some basic examples in *Project Group Examples*. Finally, *Group Filters and Imports* provides a simplified overview of how `group-filter` interacts with the *Manifest Imports* feature.

Groups Basics The `groups:` and `group-filter:` keys appear in the manifest like this:

```
manifest:
  projects:
    - name: some-project
      groups: ...
  group-filter: ...
```

The `groups` key's value is a list of group names. Group names are strings.

You can enable or disable project groups using `group-filter`. Projects whose groups are all disabled, and which are not otherwise made active by a `manifest.project-filter` configuration option, are inactive.

For example, in this manifest fragment:

```
manifest:
  projects:
    - name: project-1
      groups:
        - groupA
    - name: project-2
      groups:
        - groupB
        - groupC
    - name: project-3
```

The projects are in these groups:

- `project-1`: one group, named `groupA`
- `project-2`: two groups, named `groupB` and `groupC`
- `project-3`: no groups

Project group names must not contain commas (,), colons (:), or whitespace.

Group names must not begin with a dash (-) or the plus sign (+), but they may contain these characters elsewhere in their names. For example, `foo-bar` and `foo+bar` are valid groups, but `-foobar` and `+foobar` are not.

Group names are otherwise arbitrary strings. Group names are case sensitive.

As a restriction, no project may use both `import:` and `groups:`. (This is necessary to avoid some pathological edge cases.)

Enabled and Disabled Project Groups All project groups are enabled by default. You can enable or disable groups in both your manifest file and *Configuration*.

Within a manifest file, `manifest: group-filter:` is a YAML list of groups to enable and disable.

To enable a group, prefix its name with a plus sign (+). For example, `groupA` is enabled in this manifest fragment:

```
manifest:
  group-filter: [+groupA]
```

Although this is redundant for groups that are already enabled by default, it can be used to override settings in an imported manifest file. See *Group Filters and Imports* for more information.

To disable a group, prefix its name with a dash (-). For example, `groupA` and `groupB` are disabled in this manifest fragment:

```
manifest:
  group-filter: [-groupA,-groupB]
```

Note

Since `group-filter` is a YAML list, you could have written this fragment as follows:

```
manifest:
  group-filter:
    - -groupA
    - -groupB
```

However, this syntax is harder to read and therefore discouraged.

In addition to the manifest file, you can control which groups are enabled and disabled using the `manifest.group-filter` configuration option. This option is a comma-separated list of groups to enable and/or disable.

To enable a group, add its name to the list prefixed with `+`. To disable a group, add its name prefixed with `-`. For example, setting `manifest.group-filter` to `+groupA,-groupB` enables `groupA`, and disables `groupB`.

The value of the configuration option overrides any data in the manifest file. You can think of this as if the `manifest.group-filter` configuration option is appended to the `manifest: group-filter:` list from YAML, with "last entry wins" semantics.

Practical Example: Reducing Workspace Downloads By default, `west update` fetches all active projects defined in the manifest. Large workspaces may include optional modules, vendor HALs, experimental components, or platform-specific dependencies and vulnerabilities that are not required for every workflow.

Project groups can be used to control which grouped projects are considered active during `west` operations.

For example, consider the following manifest fragment:

```
manifest:
  projects:
    - name: hal_nordic
      groups:
        - nordic
    - name: hal_stm32
      groups:
```

(continues on next page)

(continued from previous page)

```
        - stm32
    - name: experimental_lib
      groups:
        - optional
```

A workspace targeting Nordic devices can disable the `stm32` and `optional` groups before running `west update`:

```
west config manifest.group-filter -- "-stm32,-optional"
```

After configuring the filter, running:

```
west update
```

This skips projects that belong only to disabled groups.

> **Note**
>
> Project groups only affect projects explicitly assigned to matching groups in the manifest. Projects without matching group definitions remain active regardless of the configured `manifest.group-filter` value.

> **Note**
>
> Changing `manifest.group-filter` does not automatically remove repositories already cloned into the workspace. It affects whether projects are considered active during subsequent `west` operations.

This workflow can help reduce unnecessary downloads and simplify workspace management in large multi-project environments.

Project Group Examples This section contains example situations involving project groups and active projects. The examples use both `manifest: group-filter:` YAML lists and `manifest.group-filter` configuration lists, to show how they work together.

Note that the `defaults` and `remotes` data in the following manifests isn't relevant except to make the examples complete and self-contained.

> **Note**
>
> In all of the examples that follow, the `manifest.project-filter` option is assumed to be unset.

Example 1: no disabled groups The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB
```

(continues on next page)

(continued from previous page)

```
    - name: baz

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

No groups are disabled, because all groups are enabled by default. Therefore, all three projects (`foo`, `bar`, and `baz`) are active. Note that there is no way to make project `baz` inactive, since it has no groups.

Example 2: Disabling one group via manifest The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  group-filter: [-groupA]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

Since `groupA` is disabled, project `foo` is inactive. Project `bar` is active, because `groupB` is enabled.

Example 3: Disabling multiple groups via manifest The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  group-filter: [-groupA,-groupB]

  defaults:
    remote: example-remote
  remotes:
```

(continues on next page)

(continued from previous page)

```
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is not set (you can ensure this by running `west config -D manifest.group-filter`).

Both `foo` and `bar` are inactive, because all of their groups are disabled.

Example 4: Disabling a group via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
      groups:
        - groupA
    - name: bar
      groups:
        - groupA
        - groupB

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `-groupA` (you can ensure this by running `west config manifest.group-filter -- -groupA`; the extra `--` is required so the argument parser does not treat `-groupA` as a command line option `-g` with value `roupA`).

Project `foo` is inactive because `groupA` has been disabled by the `manifest.group-filter` configuration option. Project `bar` is active because `groupB` is enabled.

Example 5: Overriding a disabled group via configuration The entire manifest file is:

```
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  group-filter: [-groupA]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The `manifest.group-filter` configuration option is set to `+groupA` (you can ensure this by running `west config manifest.group-filter +groupA`).

In this case, groupA is enabled: the manifest.group-filter configuration option has higher precedence than the manifest: group-filter: [-groupA] content in the manifest file.

Therefore, projects foo and bar are both active.

Example 6: Overriding multiple disabled groups via configuration The entire manifest file is:

```yaml
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  group-filter: [-groupA,-groupB]

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The manifest.group-filter configuration option is set to +groupA,+groupB (you can ensure this by running west config manifest.group-filter "+groupA,+groupB").

In this case, both groupA and groupB are enabled, because the configuration value overrides the manifest file for both groups.

Therefore, projects foo and bar are both active.

Example 7: Disabling multiple groups via configuration The entire manifest file is:

```yaml
manifest:
  projects:
    - name: foo
    - name: bar
      groups:
        - groupA
    - name: baz
      groups:
        - groupA
        - groupB

  defaults:
    remote: example-remote
  remotes:
    - name: example-remote
      url-base: https://git.example.com
```

The manifest.group-filter configuration option is set to -groupA,-groupB (you can ensure this by running west config manifest.group-filter -- "-groupA,-groupB").

In this case, both groupA and groupB are disabled.

Therefore, projects foo and bar are both inactive.

Group Filters and Imports This section provides a simplified description of how the `manifest: group-filter:` value behaves when combined with *Manifest Imports*. For complete details, see *Manifest Import Details*.

> **Warning**
>
> The below semantics apply to west v0.10.0 and later. West v0.9.x semantics are different, and combining `group-filter` with `import` in west v0.9.x is discouraged.

In short:

- if you only import one manifest, any groups it disables in its `group-filter` are also disabled in your manifest
- you can override this in your manifest file's `manifest: group-filter:` value, your workspace's `manifest.group-filter` configuration option, or both

Here are some examples.

Example 1: no overrides You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

Only `child` and `project-2` are active in the resolved manifest.

The `unstable` group is disabled in `child/west.yml`, and that is not overridden in `parent/west.yml`. Therefore, the final `group-filter` for the resolved manifest is `[-unstable]`.

Since `project-1` and `project-3` are in the `unstable` group and are not in any other group, they are inactive.

Example 2: overriding an imported `group-filter` via manifest You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  group-filter: [+unstable,-optional]
```

(continues on next page)

(continued from previous page)

```
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
      groups:
        - optional
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

Only the `child`, `project-1`, and `project-3` projects are active.

The `[-unstable]` group filter in `child/west.yml` is overridden in `parent/west.yml`, so the `unstable` group is enabled. Since `project-1` and `project-3` are in the `unstable` group, they are active.

The same `parent/west.yml` file disables the `optional` group, so `project-2` is inactive.

The final group filter specified by `parent/west.yml` is `[+unstable,-optional]`.

Example 3: overriding an imported `group-filter` via configuration You are using this `parent/west.yml` manifest:

```
# parent/west.yml:
manifest:
  projects:
    - name: child
      url: https://git.example.com/child
      import: true
    - name: project-1
      url: https://git.example.com/project-1
      groups:
        - unstable
```

And `child/west.yml` contains:

```
# child/west.yml:
manifest:
  group-filter: [-unstable]
  projects:
    - name: project-2
      url: https://git.example.com/project-2
      groups:
        - optional
```

(continues on next page)

(continued from previous page)

```
    - name: project-3
      url: https://git.example.com/project-3
      groups:
        - unstable
```

If you run:

```
west config manifest.group-filter +unstable,-optional
```

Then only the `child`, `project-1`, and `project-3` projects are active.

The `-unstable` group filter in `child/west.yml` is overridden in the `manifest.group-filter` configuration option, so the `unstable` group is enabled. Since `project-1` and `project-3` are in the `unstable` group, they are active.

The same configuration option disables the `optional` group, so `project-2` is inactive.

The final group filter specified by `parent/west.yml` and the `manifest.group-filter` configuration option is `[+unstable,-optional]`.

Git Submodules in Projects

You can use the `submodules` keys briefly described *above* to force `west update` to also handle any Git submodules configured in project's git repository. The `submodules` key can appear inside `projects`, like this:

```
manifest:
  projects:
    - name: some-project
      submodules: ...
```

The `submodules` key can be a boolean or a list of mappings. We'll describe these in order.

Option 1: Boolean This is the easiest way to use `submodules`.

If `submodules` is `true` as a `projects` attribute, `west update` will recursively update the project's Git submodules whenever it updates the project itself. If it's `false` or missing, it has no effect.

For example, let's say you have a source code repository `foo`, which has some submodules, and you want `west update` to keep all of them in sync, along with another project named `bar` in the same workspace.

You can do that with this manifest file:

```
manifest:
  projects:
    - name: foo
      submodules: true
    - name: bar
```

Here, `west update` will initialize and update all submodules in `foo`. If `bar` has any submodules, they are ignored, because `bar` does not have a `submodules` value.

Option 2: List of mappings The `submodules` key may be a list of mappings, one list element for each desired submodule. Each submodule listed is updated recursively. You can still track and update unlisted submodules with `git` commands manually; present or not they will be completely ignored by `west`.

The `path` key must match exactly the path of one submodule relative to its parent west project, as shown in the output of `git submodule status`. The `name` key is optional and not used by west

for now; it's not passed to `git submodule` commands either. The `name` key was briefly mandatory in west version 0.9.0, but was made optional in 0.9.1.

For example, let's say you have a source code repository `foo`, which has many submodules, and you want `west update` to keep some but not all of them in sync, along with another project named `bar` in the same workspace.

You can do that with this manifest file:

```yaml
manifest:
  projects:
    - name: foo
      submodules:
        - path: path/to/foo-first-sub
        - name: foo-second-sub
          path: path/to/foo-second-sub
    - name: bar
```

Here, `west update` will recursively initialize and update just the submodules in `foo` with paths `path/to/foo-first-sub` and `path/to/foo-second-sub`. Any submodules in `bar` are still ignored.

Repository user data

West versions v0.12 and later support an optional `userdata` key in projects.

West versions v0.13 and later supports this key in the `manifest: self:` section.

It is meant for consumption by programs that require user-specific project metadata. Beyond parsing it as YAML, west itself ignores the value completely.

The key's value is arbitrary YAML. West parses the value and makes it accessible to programs using west-apis as the `userdata` attribute of the corresponding `west.manifest.Project` object.

Example manifest fragment:

```yaml
manifest:
  projects:
    - name: foo
    - name: bar
      userdata: a-string
    - name: baz
      userdata:
        key: value
  self:
    userdata: blub
```

Example Python usage:

```python
manifest = west.manifest.Manifest.from_file()

foo, bar, baz = manifest.get_projects(['foo', 'bar', 'baz'])

foo.userdata # None
bar.userdata # 'a-string'
baz.userdata # {'key': 'value'}
manifest.userdata # 'blub'
```

Manifest Imports

You can use the `import` key briefly described above to include projects from other manifest files in your `west.yml`. This key can be either a `project` or `self` section attribute:

```yaml
manifest:
  projects:
    - name: some-project
      import: ...
  self:
    import: ...
```

You can use a "self: import:" to load additional files from the repository containing your `west.yml`. You can use a "project: ... import:" to load additional files defined in that project's Git history.

West resolves the final manifest from individual manifest files in this order:

1. imported files in `self`
2. your `west.yml` file
3. imported files in `projects`

During resolution, west ignores projects which have already been defined in other files. For example, a project named `foo` in your `west.yml` makes west ignore other projects named `foo` imported from your `projects` list.

The `import` key can be a boolean, path, mapping, or sequence. We'll describe these in order, using examples:

- *Boolean*
  - *Example 1.1: Downstream of a Zephyr release*
  - *Example 1.2: "Rolling release" Zephyr downstream*
  - *Example 1.3: Downstream of a Zephyr release, with module fork*
- *Relative path*
  - *Example 2.1: Downstream of a Zephyr release with explicit path*
  - *Example 2.2: Downstream with directory of manifest files*
  - *Example 2.3: Continuous Integration overrides*
- *Mapping with additional configuration*
  - *Example 3.1: Downstream with name allowlist*
  - *Example 3.2: Downstream with path allowlist*
  - *Example 3.3: Downstream with path blocklist*
  - *Example 3.4: Import into a subdirectory*
- *Sequence of paths and mappings*
  - *Example 4.1: Downstream with sequence of manifest files*
  - *Example 4.2: Import order illustration*

A more *formal description* of how this works is last, after the examples.

Troubleshooting Note If you're using this feature and find west's behavior confusing, try *resolving your manifest* to see the final results after imports are done.

Option 1: Boolean This is the easiest way to use `import`.

If `import` is `true` as a `projects` attribute, west imports projects from the `west.yml` file in that project's root directory. If it's `false` or missing, it has no effect. For example, this manifest would import `west.yml` from the `p1` git repository at revision `v1.0`:

```yaml
manifest:
  # ...
  projects:
    - name: p1
      revision: v1.0
      import: true    # Import west.yml from p1's v1.0 git tag
    - name: p2
      import: false   # Nothing is imported from p2.
    - name: p3        # Nothing is imported from p3 either.
```

It's an error to set `import` to either `true` or `false` inside `self`, like this:

```yaml
manifest:
  # ...
  self:
    import: true  # Error
```

Example 1.1: Downstream of a Zephyr release You have a source code repository you want to use with Zephyr v1.14.1 LTS. You want to maintain the whole thing using west. You don't want to modify any of the mainline repositories.

In other words, the west workspace you want looks like this:

```
my-downstream/
├── .west/                      # west directory
├── zephyr/                     # mainline zephyr repository
|   └── west.yml                # the v1.14.1 version of this file is imported
├── modules/                    # modules from mainline zephyr
|   ├── hal/
|   └── [...other directories..]
├── [ ... other projects ...]  # other mainline repositories
└── my-repo/                    # your downstream repository
    ├── west.yml                # main manifest importing zephyr/west.yml v1.14.1
    └── [...other files..]
```

You can do this with the following `my-repo/west.yml`:

```yaml
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v1.14.1
      import: true
```

You can then create the workspace on your computer like this, assuming `my-repo` is hosted at `https://git.example.com/my-repo`:

```
west init -m https://git.example.com/my-repo my-downstream
cd my-downstream
west update
```

After `west init`, `my-downstream/my-repo` will be cloned.

After `west update`, all of the projects defined in the `zephyr` repository's `west.yml` at revision `v1.14.1` will be cloned into `my-downstream` as well.

You can add and commit any code to `my-repo` you please at this point, including your own Zephyr applications, drivers, etc. See *Application Development*.

Example 1.2: "Rolling release" Zephyr downstream This is similar to *Example 1.1: Downstream of a Zephyr release*, except we'll use `revision: main` for the zephyr repository:

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: main
      import: true
```

You can create the workspace in the same way:

```
west init -m https://git.example.com/my-repo my-downstream
cd my-downstream
west update
```

This time, whenever you run `west update`, the special *manifest-rev* branch in the `zephyr` repository will be updated to point at a newly fetched `main` branch tip from the URL https://github.com/zephyrproject-rtos/zephyr.

The contents of `zephyr/west.yml` at the new `manifest-rev` will then be used to import projects from Zephyr. This lets you stay up to date with the latest changes in the Zephyr project. The cost is that running `west update` will not produce reproducible results, since the remote `main` branch can change every time you run it.

It's also important to understand that west **ignores your working tree's** `zephyr/west.yml` entirely when resolving imports. West always uses the contents of imported manifests as they were committed to the latest `manifest-rev` when importing from a project.

You can only import manifest from the file system if they are in your manifest repository's working tree. See *Example 2.2: Downstream with directory of manifest files* for an example.

Example 1.3: Downstream of a Zephyr release, with module fork This manifest is similar to the one in *Example 1.1: Downstream of a Zephyr release*, except it:

- is a downstream of Zephyr 2.0
- includes a downstream fork of the `modules/hal/nordic` *module* which was included in that release

```
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
    - name: my-remote
      url-base: https://git.example.com
  projects:
    - name: hal_nordic          # higher precedence
      remote: my-remote
      revision: my-sha
      path: modules/hal/nordic
    - name: zephyr
```

(continues on next page)

(continued from previous page)

```
      remote: zephyrproject-rtos
      revision: v2.0.0
      import: true             # imported projects have lower precedence

# subset of zephyr/west.yml contents at v2.0.0:
manifest:
  defaults:
    remote: zephyrproject-rtos
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
  # ...
  - name: hal_nordic           # lower precedence, values ignored
    path: modules/hal/nordic
    revision: another-sha
```

With this manifest file, the project named `hal_nordic`:

- is cloned from `https://git.example.com/hal_nordic` instead of `https://github.com/zephyrproject-rtos/hal_nordic`.
- is updated to commit `my-sha` by `west update`, instead of the mainline commit `another-sha`

In other words, when your top-level manifest defines a project, like `hal_nordic`, west will ignore any other definition it finds later on while resolving imports.

This does mean you have to copy the `path: modules/hal/nordic` value into `my-repo/west.yml` when defining `hal_nordic` there. The value from `zephyr/west.yml` is ignored entirely. See *Resolving Manifests* for troubleshooting advice if this gets confusing in practice.

When you run `west update`, west will:

- update zephyr's `manifest-rev` to point at the `v2.0.0` tag
- import `zephyr/west.yml` at that `manifest-rev`
- locally check out the `v2.0.0` revisions for all zephyr projects except `hal_nordic`
- update `hal_nordic` to `my-sha` instead of `another-sha`

Option 2: Relative path The `import` value can also be a relative path to a manifest file or a directory containing manifest files. The path is relative to the root directory of the `projects` or `self` repository the `import` key appears in.

Here is an example:

```
manifest:
  projects:
    - name: project-1
      revision: v1.0
      import: west.yml
    - name: project-2
      revision: main
      import: p2-manifests
  self:
    import: submanifests
```

This will import the following:

- the contents of `project-1/west.yml` at `manifest-rev`, which points at tag `v1.0` after running `west update`

- any YAML files in the directory tree `project-2/p2-manifests` at the latest commit in the `main` branch, as fetched by `west update`, sorted by file name
- YAML files in `submanifests` in your manifest repository, as they appear on your file system, sorted by file name

Notice how `projects` imports get data from Git using `manifest-rev`, while `self` imports get data from your file system. This is because as usual, west leaves version control for your manifest repository up to you.

Example 2.1: Downstream of a Zephyr release with explicit path This is an explicit way to write an equivalent manifest to the one in *Example 1.1: Downstream of a Zephyr release*.

```yaml
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: v1.14.1
      import: west.yml
```

The setting `import: west.yml` means to use the file `west.yml` inside the `zephyr` project. This example is contrived, but shows the idea.

This can be useful in practice when the name of the manifest file you want to import is not `west.yml`.

Example 2.2: Downstream with directory of manifest files Your Zephyr downstream has a lot of additional repositories. So many, in fact, that you want to split them up into multiple manifest files, but keep track of them all in a single manifest repository, like this:

```
my-repo/
├── submanifests
|   ├── 01-libraries.yml
|   ├── 02-vendor-hals.yml
|   └── 03-applications.yml
└── west.yml
```

You want to add all the files in `my-repo/submanifests` to the main manifest file, `my-repo/west.yml`, in addition to projects in `zephyr/west.yml`. You want to track the latest development code in the Zephyr repository's `main` branch instead of using a fixed revision.

Here's how:

```yaml
# my-repo/west.yml:
manifest:
  remotes:
    - name: zephyrproject-rtos
      url-base: https://github.com/zephyrproject-rtos
  projects:
    - name: zephyr
      remote: zephyrproject-rtos
      revision: main
      import: true
  self:
    import: submanifests
```

Manifest files are imported in this order during resolution:

1. `my-repo/submanifests/01-libraries.yml`
2. `my-repo/submanifests/02-vendor-hals.yml`
3. `my-repo/submanifests/03-applications.yml`
4. `my-repo/west.yml`
5. `zephyr/west.yml`

> **Note**
>
> The `.yml` file names are prefixed with numbers in this example to make sure they are imported in the specified order.
>
> You can pick arbitrary names. West sorts files in a directory by name before importing.

Notice how the manifests in `submanifests` are imported *before* `my-repo/west.yml` and `zephyr/west.yml`. In general, an `import` in the `self` section is processed before the manifest files in `projects` and the main manifest file.

This means projects defined in `my-repo/submanifests` take highest precedence. For example, if `01-libraries.yml` defines `hal_nordic`, the project by the same name in `zephyr/west.yml` is simply ignored. As usual, see *Resolving Manifests* for troubleshooting advice.

This may seem strange, but it allows you to redefine projects "after the fact", as we'll see in the next example.

Example 2.3: Continuous Integration overrides Your continuous integration system needs to fetch and test multiple repositories in your west workspace from a developer's forks instead of your mainline development trees, to see if the changes all work well together.

Starting with *Example 2.2: Downstream with directory of manifest files*, the CI scripts add a file `00-ci.yml` in `my-repo/submanifests`, with these contents:

```yaml
# my-repo/submanifests/00-ci.yml:
manifest:
  projects:
    - name: a-vendor-hal
      url: https://github.com/a-developer/hal
      revision: a-pull-request-branch
    - name: an-application
      url: https://github.com/a-developer/application
      revision: another-pull-request-branch
```

The CI scripts run `west update` after generating this file in `my-repo/submanifests`. The projects defined in `00-ci.yml` have higher precedence than other definitions in `my-repo/submanifests`, because the name `00-ci.yml` comes before the other file names.

Thus, `west update` always checks out the developer's branches in the projects named `a-vendor-hal` and `an-application`, even if those same projects are also defined elsewhere.

Option 3: Mapping The `import` key can also contain a mapping with the following keys:

- `file`: Optional. The name of the manifest file or directory to import. This defaults to `west.yml` if not present.
- `name-allowlist`: Optional. If present, a name or sequence of project names to include.
- `path-allowlist`: Optional. If present, a path or sequence of project paths to match against. This is a shell-style globbing pattern, currently implemented with pathlib. Note that this means case sensitivity is platform specific.

- `name-blocklist`: Optional. Like `name-allowlist`, but contains project names to exclude rather than include.
- `path-blocklist`: Optional. Like `path-allowlist`, but contains project paths to exclude rather than include.
- `path-prefix`: Optional (new in v0.8.0). If given, this will be prepended to the project's path in the workspace, as well as the paths of any imported projects. This can be used to place these projects in a subdirectory of the workspace.

Allowlists override blocklists if both are given. For example, if a project is blocked by path, then allowed by name, it will still be imported.

Example 3.1: Downstream with name allowlist Here is a pair of manifest files, representing a mainline and a downstream. The downstream doesn't want to use all the mainline projects, however. We'll assume the mainline `west.yml` is hosted at `https://git.example.com/mainline/manifest`.

```yaml
# mainline west.yml:
manifest:
  projects:
    - name: mainline-app                # included
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib
      path: libraries/lib
      url: https://git.example.com/mainline/lib
    - name: lib2                        # included
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2

# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        name-allowlist:
          - mainline-app
          - lib2
    - name: downstream-app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

An equivalent manifest in a single file would be:

```yaml
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
    - name: downstream-app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
    - name: mainline-app                  # imported
      path: examples/app
```

(continues on next page)

(continued from previous page)

```
      url: https://git.example.com/mainline/app
    - name: lib2                          # imported
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
```

If an allowlist had not been used, the `lib` project from the mainline manifest would have been imported.

Example 3.2: Downstream with path allowlist Here is an example showing how to allowlist mainline's libraries only, using `path-allowlist`.

```
# mainline west.yml:
manifest:
  projects:
    - name: app
      path: examples/app
      url: https://git.example.com/mainline/app
    - name: lib
      path: libraries/lib                 # included
      url: https://git.example.com/mainline/lib
    - name: lib2
      path: libraries/lib2                # included
      url: https://git.example.com/mainline/lib2

# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        path-allowlist: libraries/*
    - name: app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

An equivalent manifest in a single file would be:

```
manifest:
  projects:
    - name: lib                         # imported
      path: libraries/lib
      url: https://git.example.com/mainline/lib
    - name: lib2                        # imported
      path: libraries/lib2
      url: https://git.example.com/mainline/lib2
    - name: mainline
      url: https://git.example.com/mainline/manifest
    - name: app
      url: https://git.example.com/downstream/app
    - name: lib3
      path: libraries/lib3
      url: https://git.example.com/downstream/lib3
```

Example 3.3: Downstream with path blocklist Here's an example showing how to block all vendor HALs from mainline by common path prefix in the workspace, add your own version for the chip you're targeting, and keep everything else.

```yaml
# mainline west.yml:
manifest:
  defaults:
    remote: mainline
  remotes:
    - name: mainline
      url-base: https://git.example.com/mainline
  projects:
    - name: app
    - name: lib
      path: libraries/lib
    - name: lib2
      path: libraries/lib2
    - name: hal_foo
      path: modules/hals/foo     # excluded
    - name: hal_bar
      path: modules/hals/bar     # excluded
    - name: hal_baz
      path: modules/hals/baz     # excluded

# downstream west.yml:
manifest:
  projects:
    - name: mainline
      url: https://git.example.com/mainline/manifest
      import:
        path-blocklist: modules/hals/*
    - name: hal_foo
      path: modules/hals/foo
      url: https://git.example.com/downstream/hal_foo
```

An equivalent manifest in a single file would be:

```yaml
manifest:
  defaults:
    remote: mainline
  remotes:
    - name: mainline
      url-base: https://git.example.com/mainline
  projects:
    - name: app                  # imported
    - name: lib                  # imported
      path: libraries/lib
    - name: lib2                 # imported
      path: libraries/lib2
    - name: mainline
      repo-path: https://git.example.com/mainline/manifest
    - name: hal_foo
      path: modules/hals/foo
      url: https://git.example.com/downstream/hal_foo
```

Example 3.4: Import into a subdirectory You want to import a manifest and its projects, placing everything into a subdirectory of your *west workspace*.

For example, suppose you want to import this manifest from project `foo`, adding this project and its projects `bar` and `baz` to your workspace:

```yaml
# foo/west.yml:
manifest:
  defaults:
    remote: example
  remotes:
    - name: example
      url-base: https://git.example.com
  projects:
    - name: bar
    - name: baz
```

Instead of importing these into the top level workspace, you want to place all three project repositories in an `external-code` subdirectory, like this:

```
workspace/
└── external-code/
    ├── foo/
    ├── bar/
    └── baz/
```

You can do this using this manifest:

```yaml
manifest:
  projects:
    - name: foo
      url: https://git.example.com/foo
      import:
        path-prefix: external-code
```

An equivalent manifest in a single file would be:

```yaml
# foo/west.yml:
manifest:
  defaults:
    remote: example
  remotes:
    - name: example
      url-base: https://git.example.com
  projects:
    - name: foo
      path: external-code/foo
    - name: bar
      path: external-code/bar
    - name: baz
      path: external-code/baz
```

Option 4: Sequence The `import` key can also contain a sequence of files, directories, and mappings.

Example 4.1: Downstream with sequence of manifest files This example manifest is equivalent to the manifest in *Example 2.2: Downstream with directory of manifest files*, with a sequence of explicitly named files.

```yaml
# my-repo/west.yml:
manifest:
  projects:
    - name: zephyr
      url: https://github.com/zephyrproject-rtos/zephyr
      import: west.yml
  self:
    import:
      - submanifests/01-libraries.yml
      - submanifests/02-vendor-hals.yml
      - submanifests/03-applications.yml
```

Example 4.2: Import order illustration This more complicated example shows the order that west imports manifest files:

```yaml
# my-repo/west.yml
manifest:
  # ...
  projects:
    - name: my-library
    - name: my-app
    - name: zephyr
      import: true
    - name: another-manifest-repo
      import: submanifests
  self:
    import:
      - submanifests/libraries.yml
      - submanifests/vendor-hals.yml
      - submanifests/applications.yml
  defaults:
    remote: my-remote
```

For this example, west resolves imports in this order:

1. the listed files in `my-repo/submanifests` are first, in the order they occur (e.g. `libraries.yml` comes before `applications.yml`, since this is a sequence of files), since the `self: import:` is always imported first
2. `my-repo/west.yml` is next (with projects `my-library` etc. as long as they weren't already defined somewhere in `submanifests`)
3. `zephyr/west.yml` is after that, since that's the first `import` key in the `projects` list in `my-repo/west.yml`
4. files in `another-manifest-repo/submanifests` are last (sorted by file name), since that's the final project `import`

Manifest Import Details This section describes how west resolves a manifest file that uses `import` a bit more formally.

Overview The `import` key can appear in a west manifest's `projects` and `self` sections. The general case looks like this:

```yaml
# Top-level manifest file.
manifest:
  projects:
```

(continues on next page)

(continued from previous page)

```
    - name: foo
      import:
        ... # import-1
    - name: bar
      import:
        ... # import-2
    # ...
    - name: baz
      import:
        ... # import-N
  self:
    import:
      ... # self-import
```

Import keys are optional. If any of `import-1`, `...`, `import-N` are missing, west will not import additional manifest data from that project. If `self-import` is missing, no additional files in the manifest repository (beyond the top-level file) are imported.

The ultimate outcomes of resolving manifest imports are:

- a `projects` list, which is produced by combining the `projects` defined in the top-level file with those defined in imported files
- a set of extension commands, which are drawn from the `west-commands` keys in the top-level file and any imported files
- a `group-filter` list, which is produced by combining the top-level and any imported filters

Importing is done in this order:

1. Manifests from `self-import` are imported first.
2. The top-level manifest file's definitions are handled next.
3. Manifests from `import-1`, ..., `import-N`, are imported in that order.

When an individual `import` key refers to multiple manifest files, they are processed in this order:

- If the value is a relative path naming a directory (or a map whose `file` is a directory), the manifest files it contains are processed in lexicographic order – i.e., sorted by file name.
- If the value is a sequence, its elements are recursively imported in the order they appear.

This process recurses if necessary. E.g., if `import-1` produces a manifest file that contains an `import` key, it is resolved recursively using the same rules before its contents are processed further.

The following sections describe these outcomes.

Projects This section describes how the final `projects` list is created.

Projects are identified by name. If the same name occurs in multiple manifests, the first definition is used, and subsequent definitions are ignored. For example, if `import-1` contains a project named `bar`, that is ignored, because the top-level `west.yml` has already defined a project by that name.

The contents of files named by `import-1` through `import-N` are imported from Git at the latest `manifest-rev` revisions in their projects. These revisions can be updated to the values `rev-1` through `rev-N` by running `west update`. If any `manifest-rev` reference is missing or out of date, `west update` also fetches project data from the remote fetch URL and updates the reference.

Also note that all imported manifests, from the root manifest to the repository which defines a project `P`, must be up to date in order for west to update `P` itself. For example, this means `west update P` would update `manifest-rev` in the `baz` project if `baz/west.yml` defines `P`, as well

as updating the `manifest-rev` branch in the local git clone of `P`. Confusingly, updating `baz` may result in the removal of `P` from `baz/west.yml`, which "should" cause `west update P` to fail with an unrecognized project!

For this reason, it's not possible to run `west update P` if `P` is defined in an imported manifest; you must update this project along with all the others with a plain `west update`.

By default, west won't fetch any project data over the network if a project's revision is a SHA or tag which is already available locally, so updating the extra projects shouldn't take too much time unless it's really needed. See the documentation for the *update.fetch* configuration option for more information.

Extensions All extension commands defined using `west-commands` keys discovered while handling imports are available in the resolved manifest.

If an imported manifest file has a `west-commands:` definition in its `self:` section, the extension commands defined there are added to the set of available extensions at the time the manifest is imported. They will thus take precedence over any extension commands with the same names added later on.

Group filters The resolved manifest has a `group-filter` value which is the result of concatenating the `group-filter` values in the top-level manifest and any imported manifests.

Manifest files which appear earlier in the import order have higher precedence and are therefore concatenated later into the final `group-filter`.

In other words, let:

- the submanifest resolved from `self-import` have group filter `self-filter`
- the top-level manifest file have group filter `top-filter`
- the submanifests resolved from `import-1` through `import-N` have group filters `filter-1` through `filter-N` respectively

The final resolved `group-filter` value is then `filterN + ... + filter-2 + filter-1 + top-filter + self-filter`, where + here refers to list concatenation.

> **Important**
>
> The order that filters appear in the above list matters.
>
> The last filter element in the final concatenated list "wins" and determines if the group is enabled or disabled.

For example, in `[-foo] + [+foo]`, group `foo` is *enabled*. However, in `[+foo] + [-foo]`, group `foo` is *disabled*.

For simplicity, west and this documentation may elide concatenated group filter elements which are redundant using these rules. For example, `[+foo] + [-foo]` could be written more simply as `[-foo]`, for the reasons given above. As another example, `[-foo] + [+foo]` could be written as the empty list `[]`, since all groups are enabled by default.

Manifest Command

The `west manifest` command can be used to manipulate manifest files. It takes an action, and action-specific arguments.

The following sections describe each action and provides a basic signature for simple uses. Run `west manifest --help` for full details on all options.

Resolving Manifests The `--resolve` action outputs a single manifest file equivalent to your current manifest and all its *imported manifests*:

```
west manifest --resolve [-o outfile]
```

The main use for this action is to see the "final" manifest contents after performing any `imports`.

To print detailed information about each imported manifest file and how projects are handled during manifest resolution, set the maximum verbosity level using `-v`:

```
west -v manifest --resolve
```

Freezing Manifests The `--freeze` action outputs a frozen manifest:

```
west manifest --freeze [-o outfile]
```

A "frozen" manifest is a manifest file where every project's revision is a SHA. You can use `--freeze` to produce a frozen manifest that's equivalent to your current manifest file. The `-o` option specifies an output file; if not given, standard output is used.

Validating Manifests The `--validate` action either succeeds if the current manifest file is valid, or fails with an error:

```
west manifest --validate
```

The error message can help diagnose errors.

Here, "invalid" means that the syntax of the manifest file doesn't follow the rules documented on this page.

If your manifest is valid but it's not working the way you want it to, turning up the verbosity with `-v` is a good way to get detailed information about what decisions west made about your manifest, and why:

```
west -v manifest --validate
```

Get the manifest path The `--path` action prints the path to the top level manifest file:

```
west manifest --path
```

The output is something like `/path/to/workspace/west.yml`. The path format depends on your operating system.

2.11.8 Configuration

This page documents west's configuration file system, the `west config` command, and configuration options used by built-in commands. For API documentation on the `west.configuration` module, see west-apis-configuration.

West Configuration Files

West's configuration file syntax is INI-like; here is an example file:

```
[manifest]
path = zephyr

[zephyr]
base = zephyr
```

Above, the `manifest` section has option `path` set to `zephyr`. Another way to say the same thing is that `manifest.path` is `zephyr` in this file.

There are three types of configuration file:

1. **System**: Settings in this file affect west's behavior for every user logged in to the computer. Its location depends on the platform:
   - Linux: `/etc/westconfig`
   - macOS: `/usr/local/etc/westconfig`
   - Windows: `%PROGRAMDATA%\west\config`
2. **Global** (per user): Settings in this file affect how west behaves when run by a particular user on the computer.
   - All platforms: the default is `.westconfig` in the user's home directory.
   - Linux note: if the environment variable `XDG_CONFIG_HOME` is set, then `$XDG_CONFIG_HOME/west/config` is used.
   - Windows note: the following environment variables are tested to find the home directory: `%HOME%`, then `%USERPROFILE%`, then a combination of `%HOMEDRIVE%` and `%HOMEPATH%`.
3. **Local**: Settings in this file affect west's behavior for the current *west workspace*. The file is `.west/config`, relative to the workspace's root directory.

A setting in a file which appears lower down on this list overrides an earlier setting. For example, if `color.ui` is `true` in the system's configuration file, but `false` in the workspace's, then the final value is `false`. Similarly, settings in the user configuration file override system settings, and so on.

west config

The built-in `config` command can be used to get and set configuration values. You can pass `west config` the options `--system`, `--global`, or `--local` to specify which configuration file to use. Only one of these can be used at a time. If none is given, then writes default to `--local`, and reads show the final value after applying overrides.

Some examples for common uses follow; run `west config -h` for detailed help, and see *Built-in Configuration Options* for more details on built-in options.

To set `manifest.path` to `some-other-manifest`:

```
west config manifest.path some-other-manifest
```

Doing the above means that commands like `west update` will look for the *west manifest* inside the `some-other-manifest` directory (relative to the workspace root directory) instead of the directory given to `west init`, so be careful!

To read `zephyr.base`, the value which will be used as `ZEPHYR_BASE` if it is unset in the calling environment (also relative to the workspace root):

```
west config zephyr.base
```

You can switch to another zephyr repository without changing `manifest.path` – and thus the behavior of commands like `west update` – using:

```
west config zephyr.base some-other-zephyr
```

This can be useful if you use commands like `git worktree` to create your own zephyr directories, and want commands like `west build` to use them instead of the zephyr repository specified in the manifest. (You can go back to using the directory in the upstream manifest by running `west config zephyr.base zephyr`.)

To set `color.ui` to `false` in the global (user-wide) configuration file, so that west will no longer print colored output for that user when run in any workspace:

```
west config --global color.ui false
```

To undo the above change:

```
west config --global color.ui true
```

Built-in Configuration Options

The following table documents configuration options supported by west's built-in commands. Configuration options supported by Zephyr's extension commands are documented in the pages for those commands.

Option	Description
`alias.ALIAS`	String. If non-empty the `<ALIAS>` can be used as a west command. See *West aliases*.
`color.ui`	Boolean. If `true` (the default), then west output is colorized when stdout is a terminal.
`commands.allow_extensions`	Boolean, default `true`, disables *Extensions* if `false`
`grep.color`	String, default empty. Set this to `never` to disable `west grep` color output. If set, `west grep` passes the value to the grep tool's `--color` option.
`grep.tool`	String, one of `"git-grep"` (default), `"ripgrep"`, or `"grep"`. The grep tool that `west grep` should use.
`grep.<TOOL>-args`	String, default empty. The `<TOOL>` part is a pattern that can be any `grep.tool` value, so `grep.ripgrep-args` is an example configuration option. If set, arguments that `west grep` should pass to the corresponding grep tool. Run `west help grep` for details.
`grep.<TOOL>-path`	String, default empty. The `<TOOL>` part is a pattern that can be any `grep.tool` value, so `grep.ripgrep-path` is an example configuration option. The path to the corresponding tool that `west grep` should use instead of searching for the command. Run `west help grep` for details.
`manifest.file`	String, default `west.yml`. Relative path from the manifest repository root directory to the manifest file used by `west init` and other commands which parse the manifest.
`manifest.group-filter`	String, default empty. A comma-separated list of project groups to enable and disable within the workspace. Prefix enabled groups with `+` and disabled groups with `-`. For example, the value `"+foo,-bar"` enables group `foo` and disables `bar`. See *Project Groups*.
`manifest.path`	String, relative path from the *west workspace* root directory to the manifest repository used by `west update` and other commands which parse the manifest. Set locally by `west init`.

`manifest.project-filter`	Comma-separated list of strings.
	The option's value is a comma-separated list of regular expressions, each prefixed with `+` or `-`, like this:
	`+re1,-re2,-re3`
	Project names are matched against each regular expression (`re1`, `re2`, `re3`, ...) in the list, in order. If the entire project name matches the regular expression, that element of the list either deactivates or activates the project. The project is deactivated if the element begins with `-`. The project is activated if the element begins with `+`. (Project names cannot contain `,` if this option is used, so the regular expressions do not need to contain a literal `,` character.)
	If a project's name matches multiple regular expressions in the list, the result from the last regular expression is used. For example, if `manifest.project-filter` is:
	`-hal_.*,+hal_foo`
	Then a project named `hal_bar` is inactive, but a project named `hal_foo` is active.
	If a project is made inactive or active by a list element, the project is active or not regardless of whether any or all of its groups are disabled. (This is currently the only way to make a project that has no groups inactive.)
	Otherwise, i.e. if a project does not match any regular expressions in the list, it is active or inactive according to the usual rules related to its groups (see *Project Group Examples* for examples in that case).
	Within an element of a `manifest.project-filter` list, leading and trailing whitespace are ignored. That means these example values are equivalent:
	`+foo,-bar`
	`+foo ,-bar`
	Any empty elements are ignored. That means these example values are equivalent:

2.11.9 West aliases

West allows to add alias commands to the local, global or system configuration files. These aliases make it easy to add shortcuts for frequently used, or hard to memorize commands for ease of development.

Similar to how `git` aliases work, the alias command is replaced with the alias' full text and parsed as a new shell argument list (using the Python function shlex.split() internally to split the value). This enables adding argument parameters as they were passed to the original command. Spaces are considered argument separators; use proper escaping if arguments shouldn't be split.

To add a new alias simply call the `west config` command:

```
west config alias.mylist "list -f '{name} {revision}'"
```

To list aliases, use `west help` *`some_alias`*.

Recursive aliases are allowed as an alias command can contain other aliases, effectively building more complex but easy-to-remember commands.

It is possible to override an existing command, for example to pass default arguments:

```
west config alias.update "update -o=--depth=1 -n"
```

Warning

Overriding/shadowing other or built-in commands is an advanced use case, it can lead to strange side-effects and should be done with great care.

Examples

Add `west run` and `west menuconfig` shortcuts to your global configuration to call `west build` with the corresponding CMake targets:

```
west config --global alias.run "build --pristine=never --target run"
west config --global alias.menuconfig "build --pristine=never --target menuconfig"
```

Create an alias for the sample you are actively developing with additional options:

```
west config alias.sample "build -b native_sim samples/hello_world -t run -- -
↪DCONFIG_ASSERT=y"
```

Override `west update` to check a local cache:

```
west config alias.update "update --path-cache $HOME/.cache/zephyrproject"
```

Automatically exclude the 32-bit native simulator target when running *Twister* via west. This is especially useful when running on hosts systems without a 32-bit host C library (i.e. Linux/AArch64):

```
west config alias.twister "twister --exclude-platform native_sim/native"
```

2.11.10 Extensions

West is "pluggable": you can add your own commands to west without editing its source code. These are called **west extension commands**, or just "extensions" for short. Extensions show up in the `west --help` output in a special section for the project which defines them. This page provides general information on west extension commands, and has a tutorial for writing your own.

Some commands you can run when using west with Zephyr, like the ones used to *build, flash, and debug* and the *ones described here*, are extensions. That's why help for them shows up like this in `west --help`:

```
extension commands from project manifest (path: zephyr):
  completion:           display shell completion scripts
  boards:               display information about supported boards
  shields:              display list of supported shields
  build:                compile a Zephyr application
  twister:              west twister wrapper
  sign:                 sign a Zephyr binary for bootloader chain-loading
  flash:                flash and run a binary on a board
  debug:                flash and interactively debug a Zephyr application
  debugserver:          connect to board and launch a debug server
  attach:               interactively debug a board
  ...
```

See `zephyr/scripts/west-commands.yml` and the `zephyr/scripts/west_commands` directory for the implementation details.

Disabling Extension Commands

To disable support for extension commands, set the `commands.allow_extensions` *configuration* option to `false`. To set this globally for whenever you run west, use:

```
west config --global commands.allow_extensions false
```

If you want to, you can then re-enable them in a particular *west workspace* with:

```
west config --local commands.allow_extensions true
```

Note that the files containing extension commands are not imported by west unless the commands are explicitly run. See below for details.

Adding a West Extension

There are three steps to adding your own extension:

1. Write the code implementing the command.
2. Add information about it to a `west-commands.yml` file.
3. Make sure the `west-commands.yml` file is referenced in the *west manifest*.

Note that west ignores extension commands whose names are the same as a built-in command.

Step 1: Implement Your Command Create a Python file to contain your command implementation (see the west project metadata on the west PyPI page for details on the currently supported versions of Python). You can put it in anywhere in any project tracked by your *west manifest*, or the manifest repository itself. This file must contain a subclass of the `west.commands.WestCommand` class. This class will be instantiated and used when your extension is run.

Here is a basic skeleton you can use to get started. It contains a subclass of `WestCommand`, with implementations for all the abstract methods. For more details on the west APIs you can use, see west-apis.

```
'''my_west_extension.py

Basic example of a west extension.'''
```

(continues on next page)

(continued from previous page)

```
from textwrap import dedent            # just for nicer code indentation

from west.commands import WestCommand  # your extension must subclass this

class MyCommand(WestCommand):

    def __init__(self):
        super().__init__(
            'my-command-name',  # gets stored as self.name
            '', # ignored self.help, will not be required by future west versions
            # self.description:
            description=dedent('''
            A multi-line description of my-command.

            You can split this up into multiple paragraphs and they'll get
            reflowed for you. You can also pass
            formatter_class=argparse.RawDescriptionHelpFormatter when calling
            parser_adder.add_parser() below if you want to keep your line
            endings.'''))

    def do_add_parser(self, parser_adder):
        # This is a bit of boilerplate, which allows you full control over the
        # type of argparse handling you want. The "parser_adder" argument is
        # the return value of an argparse.ArgumentParser.add_subparsers() call.
        parser = parser_adder.add_parser(self.name,
                                         description=self.description)

        # Add some example options using the standard argparse module API.
        parser.add_argument('-o', '--optional', help='an optional argument')
        parser.add_argument('required', help='a required argument')

        return parser           # gets stored as self.parser

    def do_run(self, args, unknown):
        # This gets called when the user runs the command, e.g.:
        #
        #   $ west my-command-name -o FOO BAR
        #   --optional is FOO
        #   required is BAR
        self.inf('--optional is', args.optional)
        self.inf('required is', args.required)
```

You can ignore the second argument to `do_run()` (`unknown` above), as `WestCommand` will reject unknown arguments by default. If you want to be passed a list of unknown arguments instead, add `accepts_unknown_args=True` to the `super().__init__()` arguments.

Step 2: Add or Update Your `west-commands.yml` You now need to add a `west-commands.yml` file to your project which describes your extension to west.

Here is an example for the above class definition, assuming it's in `my_west_extension.py` at the project root directory:

```
west-commands:
  - file: my_west_extension.py
    commands:
      - name: my-command-name
```

(continues on next page)

(continued from previous page)

```
        class: MyCommand
        help: one-line help for what my-command-name does
```

The top level of this YAML file is a map with a `west-commands` key. The key's value is a sequence of "command descriptors". Each command descriptor gives the location of a file implementing west extensions, along with the names of those extensions, and optionally the names of the classes which define them (if not given, the `class` value defaults to the same thing as `name`).

Some information in this file is redundant with definitions in the Python code. This is because west won't import `my_west_extension.py` until the user runs `west my-command-name`, since:

- It allows users to run `west update` with a manifest from an untrusted source, then use other west commands without your code being imported along the way. Since importing a Python module is shell-equivalent, this provides some peace of mind.
- It's a small optimization, since your code will only be imported if it is needed.

So, unless your command is explicitly run, west will just load the `west-commands.yml` file to get the basic information it needs to display information about your extension to the user in `west --help` output, etc.

If you have multiple extensions, or want to split your extensions across multiple files, your `west-commands.yml` will look something like this:

```
west-commands:
  - file: my_west_extension.py
    commands:
      - name: my-command-name
        class: MyCommand
        help: one-line help for what my-command-name does
  - file: another_file.py
    commands:
      - name: command2
        help: another cool west extension
      - name: a-third-command
        class: ThirdCommand
        help: a third command in the same file as command2
```

Above:

- `my_west_extension.py` defines extension `my-command-name` with class `MyCommand`
- `another_file.py` defines two extensions:
  1. `command2` with class `command2`
  2. `a-third-command` with class `ThirdCommand`

See the file `west-commands-schema.yml` in the west repository for a schema describing the contents of a `west-commands.yml`.

Step 3: Update Your Manifest Finally, you need to specify the location of the `west-commands.yml` you just edited in your west manifest. If your extension is in a project, add it like this:

```
manifest:
   # [... other contents ...]

   projects:
     - name: your-project
       west-commands: path/to/west-commands.yml
     # [... other projects ...]
```

Where `path/to/west-commands.yml` is relative to the root of the project. Note that the name `west-commands.yml`, while encouraged, is just a convention; you can name the file something else if you need to.

Alternatively, if your extension is in the manifest repository, just do the same thing in the manifest's `self` section, like this:

```
manifest:
  # [... other contents ...]

  self:
    west-commands: path/to/west-commands.yml
```

That's it; you can now run `west my-command-name`. Your command's name, help, and the project which contains its code will now also show up in the `west --help` output. If you share the updated repositories with others, they'll be able to use it, too.

2.11.11 Building, Flashing and Debugging

Zephyr provides several *west extension commands* for building, flashing, and interacting with Zephyr programs running on a board: `build`, `flash`, `debug`, `debugserver`, `rtt`, and `attach`.

For information on adding board support for the flashing and debugging commands, see *Flash and debug support* in the board porting guide.

Building: `west build`

> **Tip**
>
> Run `west build -h` for a quick overview.

The `build` command helps you build Zephyr applications from source. You can use *west config* to configure its behavior.

Its default behavior tries to "do what you mean":

- If there is a Zephyr build directory named `build` in your current working directory, it is incrementally re-compiled. The same is true if you run `west build` from a Zephyr build directory.
- Otherwise, if you run `west build` from a Zephyr application's source directory and no build directory is found, a new one is created and the application is compiled in it.

Basics The easiest way to use `west build` is to go to an application's root directory (i.e. the folder containing the application's `CMakeLists.txt`) and then run:

```
west build -b <BOARD>
```

Where `<BOARD>` is the name of the board you want to build for. This is exactly the same name you would supply to CMake if you were to invoke it with: `cmake -DBOARD=<BOARD>`.

> **Tip**
>
> You can use the *west boards* command to list all supported boards.

A build directory named `build` will be created, and the application will be compiled there after `west build` runs CMake to create a build system in that directory. If `west build` finds an existing

build directory, the application is incrementally re-compiled there without re-running CMake. You can force CMake to run again with `--cmake`.

You don't need to use the `--board` option if you've already got an existing build directory; `west build` can figure out the board from the CMake cache. For new builds, the `--board` option, *BOARD* environment variable, or `build.board` configuration option are checked (in that order).

Sysbuild (multi-domain builds) *Sysbuild (System build)* can be used to create a multi-domain build system combining multiple images for a single or multiple boards.

Use `--sysbuild` to select the *Sysbuild (System build)* build infrastructure with `west build` to build multiple domains.

More detailed information regarding the use of sysbuild can be found in the *Sysbuild (System build)* guide.

> **Tip**
>
> The `build.sysbuild` configuration option can be enabled to tell `west build` to default build using sysbuild. `--no-sysbuild` can be used to disable sysbuild for a specific build.

`west build` will build all domains through the top-level build folder of the domains specified by sysbuild.

A single domain from a multi-domain project can be built by using `--domain` argument.

Examples Here are some `west build` usage examples, grouped by area.

Forcing CMake to Run Again To force a CMake re-run, use the `--cmake` (or `-c`) option:

```
west build -c
```

Setting a Default Board To configure `west build` to build for the `reel_board` by default:

```
west config build.board reel_board
```

(You can use any other board supported by Zephyr here; it doesn't have to be `reel_board`.)

Setting Source and Build Directories To set the application source directory explicitly, give its path as a positional argument:

```
west build -b <BOARD> path/to/source/directory
```

To set the build directory explicitly, use `--build-dir` (or `-d`):

```
west build -b <BOARD> --build-dir path/to/build/directory
```

To change the default build directory from `build`, use the `build.dir-fmt` configuration option. This lets you name build directories using format strings, like this:

```
west config build.dir-fmt "build/{board}/{app}"
```

With the above, running `west build -b reel_board samples/hello_world` will use build directory `build/reel_board/hello_world`. See *Configuration Options* for more details on this option.

Setting the Build System Target To specify the build system target to run, use `--target` (or `-t`).

For example, on host platforms with QEMU, you can use the `run` target to build and run the hello_world sample for the emulated qemu_x86 board in one command:

```
west build -b qemu_x86 -t run samples/hello_world
```

As another example, to use `-t` to list all build system targets:

```
west build -t help
```

As a final example, to use `-t` to run the `pristine` target, which deletes all the files in the build directory:

```
west build -t pristine
```

Pristine Builds A *pristine* build directory is essentially a new build directory. All byproducts from previous builds have been removed.

To force `west build` make the build directory pristine before re-running CMake to generate a build system, use the `--pristine=always` (or `-p=always`) option.

Giving `--pristine` or `-p` without a value has the same effect as giving it the value `always`. For example, these commands are equivalent:

```
west build -p -b reel_board samples/hello_world
west build -p=always -b reel_board samples/hello_world
```

By default, `west build` makes no attempt to detect if the build directory needs to be made pristine. This can lead to errors if you do something like try to reuse a build directory for a different `--board`.

Using `--pristine=auto` makes `west build` detect some of these situations and make the build directory pristine before trying the build.

Tip

You can run `west config build.pristine always` to always do a pristine build, or `west config build.pristine never` to disable the heuristic. See the `west build` *Configuration Options* for details.

Verbose Builds To print the CMake and compiler commands run by `west build`, use the global west verbosity option, `-v`:

```
west -v build -b reel_board samples/hello_world
```

One-Time CMake Arguments To pass additional arguments to the CMake invocation performed by `west build`, pass them after a `--` at the end of the command line.

Important

Passing additional CMake arguments like this forces `west build` to re-run the CMake build configuration step, even if a build system has already been generated. This will make incremental builds slower (but still much faster than building from scratch).

After using `--` once to generate the build directory, use `west build -d <build-dir>` on subsequent runs to do incremental builds.

> Alternatively, make your CMake arguments permanent as described in the next section; it will not slow down incremental builds.

For example, to use the Unix Makefiles CMake generator instead of Ninja (which `west build` uses by default), run:

```
west build -b reel_board -- -G'Unix Makefiles'
```

To use Unix Makefiles and set CMAKE_VERBOSE_MAKEFILE to `ON`:

```
west build -b reel_board -- -G'Unix Makefiles' -DCMAKE_VERBOSE_MAKEFILE=ON
```

Notice how the `--` only appears once, even though multiple CMake arguments are given. All command-line arguments to `west build` after a `--` are passed to CMake.

To set *DTC_OVERLAY_FILE* to `enable-modem.overlay`, using that file as a *devicetree overlay*:

```
west build -b reel_board -- -DDTC_OVERLAY_FILE=enable-modem.overlay
```

To merge the `file.conf` Kconfig fragment into your build's `.config`:

```
west build -- -DEXTRA_CONF_FILE=file.conf
```

Permanent CMake Arguments The previous section describes how to add CMake arguments for a single `west build` command. If you want to save CMake arguments for `west build` to use every time it generates a new build system instead, you should use the `build.cmake-args` configuration option. Whenever `west build` runs CMake to generate a build system, it splits this option's value according to shell rules and includes the results in the `cmake` command line.

Remember that, by default, `west build` **tries to avoid generating a new build system if one is present** in your build directory. Therefore, you need to either delete any existing build directories or do a *pristine build* after setting `build.cmake-args` to make sure it will take effect.

For example, to always enable **CMAKE_EXPORT_COMPILE_COMMANDS**, you can run:

```
west config build.cmake-args -- -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

(The extra `--` is used to force the rest of the command to be treated as a positional argument. Without it, *west config* would treat the `-DVAR=VAL` syntax as a use of its `-D` option.)

To enable **CMAKE_VERBOSE_MAKEFILE**, so CMake always produces a verbose build system:

```
west config build.cmake-args -- -DCMAKE_VERBOSE_MAKEFILE=ON
```

To save more than one argument in `build.cmake-args`, use a single string whose value can be split into distinct arguments (`west build` uses the Python function shlex.split() internally to split the value).

For example, to enable both **CMAKE_EXPORT_COMPILE_COMMANDS** and **CMAKE_VERBOSE_MAKEFILE**:

```
west config build.cmake-args -- "-DCMAKE_EXPORT_COMPILE_COMMANDS=ON -
↪DCMAKE_VERBOSE_MAKEFILE=ON"
```

If you want to save your CMake arguments in a separate file instead, you can combine CMake's `-C <initial-cache>` option with `build.cmake-args`. For instance, another way to set the options used in the previous example is to create a file named `~/my-cache.cmake` with the following contents:

```
set(CMAKE_EXPORT_COMPILE_COMMANDS ON CACHE BOOL "")
set(CMAKE_VERBOSE_MAKEFILE ON CACHE BOOL "")
```

Then run:

```
west config build.cmake-args "-C ~/my-cache.cmake"
```

See the cmake(1) manual page and the set() command documentation for more details.

Build tool arguments Use `-o` to pass options to the underlying build tool.

This works with both `ninja` (*the default*) and `make` based build systems.

For example, to pass `-dexplain` to `ninja`:

```
west build -o=-dexplain
```

As another example, to pass `--keep-going` to `make`:

```
west build -o=--keep-going
```

Note that using `-o=--foo` instead of `-o --foo` is required to prevent `--foo` from being treated as a `west build` option.

Build parallelism By default, `ninja` uses all of your cores to build, while `make` uses only one. You can control this explicitly with the `-j` option supported by both tools.

For example, to build with 4 cores:

```
west build -o=-j4
```

The `-o` option is described further in the previous section.

Build a single domain In a multi-domain build with hello_world and MCUboot, you can use `--domain hello_world` to only build this domain:

```
west build --sysbuild --domain hello_world
```

The `--domain` argument can be combined with the `--target` argument to build the specific target for the target, for example:

```
west build --sysbuild --domain hello_world --target help
```

Use a snippet See *Using Snippets*.

Configuration Options You can *configure* `west build` using these options.

Option	Description
`build.board`	String. If given, this the board used by *west build* when `--board` is not given and `BOARD` is unset in the environment.
`build.board_warn`	Boolean, default `true`. If `false`, disables warnings when `west build` can't figure out the target board.
`build.cmake-args`	String. If present, the value will be split according to shell rules and passed to CMake whenever a new build system is generated. See *Permanent CMake Arguments*.
`build.dir-fmt`	String, default `build`. The build folder format string, used by west whenever it needs to create or locate a build folder. The currently available arguments are: • `west_topdir`: The absolute path to the west workspace, as returned by the `west_topdir` command • `board`: The board name • `source_dir`: Path to the CMake source directory, relative to the current working directory. If the current working directory is inside the source directory, this is an empty string, as it is on Windows for a source directory on another drive, which has no relative path to the current one. If no source directory is specified, it defaults to current working directory. E.g. if `west build ../app` is run from `<west_topdir>/app1`, `source_dir` resolves to `../app` (which is the relative path to the current working dir). • `source_dir_workspace`: Path to the source directory, relative to `west_topdir` (if it is inside the workspace). Otherwise, it is relative to the filesystem root (`/` on Unix, respectively `C:/` on Windows). E.g. if `west build ../app` is run from `<west_topdir>/app1`, `source_dir` resolves to `app` (which is the relative path to the west workspace dir). • `app`: The name of the source directory.
`build.generator`	String, default `Ninja`. The CMake Generator to use to create a build system. (To set a generator for a single build, see the *above example*)
`build.guess-dir`	String, instructs west whether to try to guess what build folder to use when `build.dir-fmt` is in use and not enough information is available to resolve the build folder name. Can take these values: • `never` (default): Never try to guess, bail out instead and require the user to provide a build folder with `-d`. • `runners`: Try to guess the folder when using any of the 'runner' commands. These are typically all commands that invoke an external tool, such as `flash` and `debug`.
`build.pristine`	String. Controls the way in which `west build` may clean the build folder before building. Can take the following values: • `never` (default): Never automatically make the build folder pristine. • `auto`: `west build` will automatically make the build folder pristine before building, if a build system is present and the build would fail otherwise (e.g. the user has specified a different board or application from the one previously used to make the build directory). • `always`: Always make the build folder pristine before building, if a build system is present.
`build.sysbuild`	Boolean, default `false`. If `true`, build application using the sysbuild infrastructure.

Flashing: `west flash`

Tip

Run `west flash -h` for additional help.

Basics From a Zephyr build directory, re-build the binary and flash it to your board:

```
west flash
```

To specify the build directory, use `--build-dir` (or `-d`):

```
west flash --build-dir path/to/build/directory
```

If you don't specify the build directory, `west flash` searches for one in `build`, then the current working directory. If you set the `build.dir-fmt` configuration option (see *Setting Source and Build Directories*), `west flash` searches there instead of `build`.

Choosing a Runner If your board's Zephyr integration supports flashing with multiple programs, you can specify which one to use using the `--runner` (or `-r`) option. For example, if West flashes your board with `nrfjprog` by default, but it also supports JLink, you can override the default with:

```
west flash --runner jlink
```

You can override the default flash runner at build time by using the `BOARD_FLASH_RUNNER` CMake variable, and the debug runner with `BOARD_DEBUG_RUNNER`.

For example:

```
# Set the default runner to "jlink", overriding the board's
# usual default.
west build [...] -- -DBOARD_FLASH_RUNNER=jlink
```

See *One-Time CMake Arguments* and *Permanent CMake Arguments* for more information on setting CMake arguments.

See *Configuration Options* below for more information on the `runner` library used by West. The list of runners which support flashing can be obtained with `west flash -H`; if run from a build directory or with `--build-dir`, this will print additional information on available runners for your board.

Configuration Overrides The CMake cache contains default values West uses while flashing, such as where the board directory is on the file system, the path to the zephyr binaries to flash in several formats, and more. You can override any of this configuration at runtime with additional options.

For example, to override the HEX file containing the Zephyr image to flash (assuming your runner expects a HEX file), but keep other flash configuration at default values:

```
west flash --hex-file path/to/some/other.hex
```

The `west flash -h` output includes a complete list of overrides supported by all runners.

Runner-Specific Overrides Each runner may support additional options related to flashing. For example, some runners support an `--erase` flag, which mass-erases the flash storage on your board before flashing the Zephyr image.

To view all of the available options for the runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west flash --context
```

Important

Note the capital H in the short option name. This re-runs the build in order to ensure the information displayed is up to date!

When running West outside of a build directory, `west flash -H` just prints a list of runners. You can use `west flash -H -r <runner-name>` to print usage information for options supported by that runner.

For example, to print usage information about the `jlink` runner:

```
west flash -H -r jlink
```

Multi-domain flashing When a *Sysbuild (multi-domain builds)* folder is detected, then `west flash` will flash all domains in the order defined by sysbuild.

It is possible to flash the image from a single domain in a multi-domain project by using `--domain`.

For example, in a multi-domain build with hello_world and MCUboot, you can use the `--domain hello_world` domain to only flash only the image from this domain:

```
west flash --domain hello_world
```

Configuration Options You can *configure* `west flash` using these options.

Option	Description
`flash.rebuild`	Boolean, default `true`. If `false`, do not rebuild on west flash.

Debugging: `west debug`, `west debugserver`

Tip

Run `west debug -h` or `west debugserver -h` for additional help.

Basics From a Zephyr build directory, to attach a debugger to your board and open up a debug console (e.g. a GDB session):

```
west debug
```

To attach a debugger to your board and open up a local network port you can connect a debugger to (e.g. an IDE debugger):

```
west debugserver
```

To specify the build directory, use `--build-dir` (or `-d`):

```
west debug --build-dir path/to/build/directory
west debugserver --build-dir path/to/build/directory
```

If you don't specify the build directory, these commands search for one in `build`, then the current working directory. If you set the `build.dir-fmt` configuration option (see *Setting Source and Build Directories*), `west debug` searches there instead of `build`.

Choosing a Runner If your board's Zephyr integration supports debugging with multiple programs, you can specify which one to use using the `--runner` (or `-r`) option. For example, if West debugs your board with `pyocd-gdbserver` by default, but it also supports JLink, you can override the default with:

```
west debug --runner jlink
west debugserver --runner jlink
```

See *Configuration Options* below for more information on the `runner` library used by West. The list of runners which support debugging can be obtained with `west debug -H`; if run from a build directory or with `--build-dir`, this will print additional information on available runners for your board.

Configuration Overrides The CMake cache contains default values West uses for debugging, such as where the board directory is on the file system, the path to the zephyr binaries containing symbol tables, and more. You can override any of this configuration at runtime with additional options.

For example, to override the ELF file containing the Zephyr binary and symbol tables (assuming your runner expects an ELF file), but keep other debug configuration at default values:

```
west debug --elf-file path/to/some/other.elf
west debugserver --elf-file path/to/some/other.elf
```

The `west debug -h` output includes a complete list of overrides supported by all runners.

Runner-Specific Overrides Each runner may support additional options related to debugging. For example, some runners support flags which allow you to set the network ports used by debug servers.

To view all of the available options for the runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west debug --context
```

(The command `west debugserver --context` will print the same output.)

> **Important**
>
> Note the capital H in the short option name. This re-runs the build in order to ensure the information displayed is up to date!

When running West outside of a build directory, `west debug -H` just prints a list of runners. You can use `west debug -H -r <runner-name>` to print usage information for options supported by that runner.

For example, to print usage information about the `jlink` runner:

```
west debug -H -r jlink
```

Multi-domain debugging `west debug` can only debug a single domain at a time. When a *Sysbuild (multi-domain builds)* folder is detected, `west debug` will debug the `default` domain specified by sysbuild.

The default domain will be the application given as the source directory. See the following example:

```
west build --sysbuild path/to/source/directory
```

For example, when building `hello_world` with MCUboot using sysbuild, `hello_world` becomes the default domain:

```
west build --sysbuild samples/hello_world
```

So to debug `hello_world` you can do:

```
west debug
```

or:

```
west debug --domain hello_world
```

If you wish to debug MCUboot, you must explicitly specify MCUboot as the domain to debug:

```
west debug --domain mcuboot
```

Configuration Options You can *configure* `west debug` and *configure* `west debugserver` using these options.

Option	Description
`debug.rebuild`	Boolean, default `true`. If `false`, do not rebuild on west debug.
`debugserver.rebuild`	Boolean, default `true`. If `false`, do not rebuild on west debugserver.

Flash and debug runners

The flash and debug commands use Python wrappers around various *Flash & Debug Host Tools*. These wrappers are all defined in a Python library at scripts/west_commands/runners. Each wrapper is called a *runner*. Runners can flash and/or debug Zephyr programs.

The central abstraction within this library is `ZephyrBinaryRunner`, an abstract class which represents runners. The set of available runners is determined by the imported subclasses of `ZephyrBinaryRunner`. `ZephyrBinaryRunner` is available in the `runners.core` module; individual runner implementations are in other submodules, such as `runners.nrfjprog`, `runners.openocd`, etc.

Running Robot Framework tests: `west robot`

> **Tip**
>
> Run `west robot -h` for additional help.

Basics Currently the command supports only one runner which is using `renode-test`, (essentially a wrapper for running Robot tests in Renode), but can be easily extended by adding other runners.

From a Zephyr build directory, to run a Robot test suite:

```
west robot --runner=renode-robot --testsuite path/to/testsuite.robot
```

This will run all tests from testsuite.robot and print output provided by Robot Framework.

To pass additional parameters to Renode use `--renode-robot-args` switch. For example to show Renode logs in addition to Robot Framework's output:

west robot –runner=renode-robot –testsuite path/to/testsuite.robot –renode-robot-arg="–show-log"

Runner-Specific Overrides To view all of the available options for the Robot runners your board supports, as well as their usage information, use `--context` (or `-H`):

```
west robot --runner=renode-robot --context
```

To view all available options "renode-test" runner supports, use:

```
west robot --runner=renode-robot --renode-robot-help
```

Simulating a board with: `west simulate`

Basics Currently the command supports only one runner which is using Renode, but can be easily extended by adding other runners.

From a Zephyr build directory, to run the built binary:

```
west simulate --runner=renode
```

This will start Renode and configure simulation based on a default `.resc` script for the current platform with the zephyr.elf file loaded by default. The simulation then can be started by typing "start" or "s" in Renode's Monitor. This can also be done by passing a command to Renode, using an argument provided by the runner:

west simulate –runner=renode –renode-command start

To pass an argument to Renode itself, for example to start Renode in console mode instead of a separate window:

west simulate –runner=renode –renode-arg="–console"

From that point on Renode can be used normally in both console and window modes. For details on using Renode see Renode - documentation.

Runner-Specific Overrides To view all of the available options supported by the runners, as well as their usage information, use `--context` (or `-H`):

```
west simulate --runner=renode --context
```

To view all available options Renode supports, use:

```
west simulate --runner=renode --renode-help
```

Out of tree runners

Zephyr modules can have external runners discovered by adding python files in their *module.yml*. Create an external runner class by inheriting from `ZephyrBinaryRunner` and implement all abstract methods.

> **Note**
>
> Support for custom out-of-tree runners makes the `runners.core` module part of the public API and backwards incompatible changes need to undergo the *deprecation process*.

Hacking

This section documents the `runners.core` module used by the flash and debug commands. This is the core abstraction used to implement support for these features.

Developers can add support for new ways to flash and debug Zephyr programs by implementing additional runners. To get this support into upstream Zephyr, the runner should be added into a new or existing `runners` module, and imported from `runners/__init__.py`.

> **Note**
>
> The test cases in scripts/west_commands/tests add unit test coverage for the runners package and individual runner classes.
>
> Please try to add tests when adding new runners. Note that if your changes break existing test cases, CI testing on upstream pull requests will fail.

Zephyr binary runner core interfaces

This provides the core ZephyrBinaryRunner class meant for public use, as well as some other helpers for concrete runner classes.

class `runners.core.`**`BuildConfiguration`**(*build_dir: str*)

This helper class provides access to build-time configuration.

Configuration options can be read as if the object were a dict, either object['CONFIG_FOO'] or object.get('CONFIG_FOO').

Kconfig configuration values are available (parsed from .config).

Devicetree configuration is available using the edtlib API as object.edt.

property edt

Get the EDT representing the devicetree. The EDT is lazily loaded on first access, and cached for subsequent accesses.

getboolean(*option*)

If a boolean option is explicitly set to y or n, returns its value. Otherwise, falls back to False.

class `runners.core.`**`DeprecatedAction`**(*option_strings, dest, nargs=None, const=None, default=None, type=None, choices=None, required=False, help=None, metavar=None*)

class `runners.core.`**`FileType`**(**values*)

exception `runners.core.`**`MissingProgram`**(*program*)

FileNotFoundError subclass for missing program dependencies.

No significant changes from the parent FileNotFoundError; this is useful for explicitly signaling that the file in question is a program that some class requires to proceed.

The filename attribute contains the missing program.

class `runners.core.`**`NetworkPortHelper`**

Helper class for dealing with local IP network ports.

get_unused_ports(*starting_from*)

Find unused network ports, starting at given values.

starting_from is an iterable of ports the caller would like to use.

The return value is an iterable of ports, in the same order, using the given values if they were unused, or the next sequentially available unused port otherwise.

Ports may be bound between this call's check and actual usage, so callers still need to handle errors involving returned ports.

`class runners.core.RunnerCaps`(*commands: set[str] = <factory>, dev_id: bool = False, mult_dev_ids: bool = False, flash_addr: bool = False, erase: bool = False, reset: bool = False, reset_types: bool = False, reset_types_supported: list[str] | None = None, extload: bool = False, tool_opt: bool = False, gdb_init: bool = False, file: bool = False, hide_load_files: bool = False, rtt: bool = False, dry_run: bool = False, skip_load: bool = False, batch_debug: bool = False*)

This class represents a runner class's capabilities.

Each capability is represented as an attribute with the same name. Flag attributes are True or False.

Available capabilities:

- commands: set of supported commands; default is {'flash', 'debug', 'debugserver', 'attach', 'simulate', 'robot', 'rtt'}.
- dev_id: whether the runner supports device identifiers, in the form of an -i, –dev-id option. This is useful when the user has multiple debuggers connected to a single computer, in order to select which one will be used with the command provided.
- mult_dev_ids: whether the runner supports multiple device identifiers for a single operation, allowing for bulk flashing of devices.
- flash_addr: whether the runner supports flashing to an arbitrary address. Default is False. If true, the runner must honor the –dt-flash option.
- erase: whether the runner supports an –erase option, which does a mass-erase of the entire addressable flash on the target before flashing. On multi-core SoCs, this may only erase portions of flash specific the actual target core. (This option can be useful for things like clearing out old settings values or other subsystem state that may affect the behavior of the zephyr image. It is also sometimes needed by SoCs which have flash-like areas that can't be sector erased by the underlying tool before flashing; UICR on nRF SoCs is one example.)
- reset: whether the runner supports a –reset option, which resets the device after a flash operation is complete.
- reset_types: whether the runner supports a –reset-type=x option, which specifies a runner specific value for selecting a specific type of reset to use when resetting the device.
- reset_types_supported: a list of values allowed to be passed with the –reset-type option. If the list of supported reset types is dynamic/device specific, this should be omitted, allowing any value to be set and passed directly to the underlying tool. In addition to these, each runner implementation must always be ready to accept *None* in a runner-specific fashion. For user-friendliness, it is recommended to place first in the list the value which has the behavior identical or most similar to *None*.
- extload: whether the runner supports a –extload option, which must be given one time and is passed on to the underlying tool that the runner wraps.
- tool_opt: whether the runner supports a –tool-opt (-O) option, which can be given multiple times and is passed on to the underlying tool that the runner wraps.
- gdb_init: whether the runner supports a –gdb-init option, which can be given multiple times and appends commands, in the order given, to the GDB client invocation, after the ones the runner issues itself.
- file: whether the runner supports a –file option, which specifies exactly the file that should be used to flash, overriding any default discovered in the build directory.

- hide_load_files: whether the elf/hex/bin file arguments should be hidden.
- rtt: whether the runner supports SEGGER RTT. This adds a –rtt-address option.
- skip_load: whether the runner supports the –load/–no-load option, which allows skipping the load of image on target before starting a debug session (this option only affects the 'debug' command)

class runners.core.**RunnerConfig**(*build_dir: str, board_dir: str, elf_file: str | None, exe_file: str | None, hex_file: str | None, bin_file: str | None, uf2_file: str | None, mot_file: str | None, file: str | None, file_type: FileType | None = FileType.OTHER, gdb: str | None = None, openocd: str | None = None, openocd_search: list[str] = [], rtt_address: int | None = None*)

Runner execution-time configuration.

This is a common object shared by all runners. Individual runners can register specific configuration options using their do_add_parser() hooks.

`bin_file: str | None`
Alias for field number 5

`board_dir: str`
Alias for field number 1

`build_dir: str`
Alias for field number 0

`elf_file: str | None`
Alias for field number 2

`exe_file: str | None`
Alias for field number 3

`file: str | None`
Alias for field number 8

`file_type: FileType | None`
Alias for field number 9

`gdb: str | None`
Alias for field number 10

`hex_file: str | None`
Alias for field number 4

`mot_file: str | None`
Alias for field number 7

`openocd: str | None`
Alias for field number 11

`openocd_search: list[str]`
Alias for field number 12

`rtt_address: int | None`
Alias for field number 13

`uf2_file: str | None`
Alias for field number 6

`class runners.core.SysbuildConfiguration(build_dir: str)`

This helper class provides access to sysbuild-time configuration.

Configuration options can be read as if the object were a dict, either object['SB_CONFIG_FOO'] or object.get('SB_CONFIG_FOO').

Kconfig configuration values are available (parsed from .config).

`class runners.core.ZephyrBinaryRunner(cfg: RunnerConfig)`

Abstract superclass for binary runners (flashers, debuggers).

Note: this class's API has changed relatively rarely since it as added, but it is not considered a stable Zephyr API, and may change without notice.

With some exceptions, boards supported by Zephyr must provide generic means to be flashed (have a Zephyr firmware binary permanently installed on the device for running) and debugged (have a breakpoint debugger and program loader on a host workstation attached to a running target).

This is supported by four top-level commands managed by the Zephyr build system:

- 'flash': flash a previously configured binary to the board, start execution on the target, then return.
- 'debug': connect to the board via a debugging protocol, program the flash, then drop the user into a debugger interface with symbol tables loaded from the current binary, and block until it exits.
- 'debugserver': connect via a board-specific debugging protocol, then reset and halt the target. Ensure the user is now able to connect to a debug server with symbol tables loaded from the binary.
- 'attach': connect to the board via a debugging protocol, then drop the user into a debugger interface with symbol tables loaded from the current binary, and block until it exits. Unlike 'debug', this command does not program the flash.

This class provides an API for these commands. Every subclass is called a 'runner' for short. Each runner has a name (like 'pyocd'), and declares commands it can handle (like 'flash'). Boards (like 'nrf52dk/nrf52832') declare which runner(s) are compatible with them to the Zephyr build system, along with information on how to configure the runner to work with the board.

The build system will then place enough information in the build directory to create and use runners with this class's create() method, which provides a command line argument parsing API. You can also create runners by instantiating subclasses directly.

In order to define your own runner, you need to:

1. Define a ZephyrBinaryRunner subclass, and implement its abstract methods. You may need to override capabilities().
2. Make sure the Python module defining your runner class is imported, e.g. by editing this package's __init__.py (otherwise, get_runners() won't work).
3. Give your runner's name to the Zephyr build system in your board's board.cmake.

Additional advice:

- If you need to import any non-standard-library modules, make sure to catch ImportError and defer complaints about it to a RuntimeError if one is missing. This avoids affecting users that don't require your runner, while still making it clear what went wrong to users that do require it that don't have the necessary modules installed.
- If you need to ask the user something (e.g. using input()), do it in your create() classmethod, not do_run(). That ensures your __init__() really has everything it needs to call do_run(), and also avoids calling input() when not instantiating within a command line application.

- Use self.logger to log messages using the standard library's logging API; your logger is named "runner.<your-runner-name()>"

For command-line invocation from the Zephyr build system, runners define their own argparse-based interface through the common add_parser() (and runner-specific do_add_parser() it delegates to), and provide a way to create instances of themselves from a RunnerConfig and parsed runner-specific arguments via create().

Runners use a variety of host tools and configuration values, the user interface to which is abstracted by this class. Each runner subclass should take any values it needs to execute one of these commands in its constructor. The actual command execution is handled in the run() method.

`classmethod add_parser`(*parser*)

Adds a sub-command parser for this runner.

The given object, parser, is a sub-command parser from the argparse module. For more details, refer to the documentation for argparse.ArgumentParser.add_subparsers().

The lone common optional argument is:

- –dt-flash (if the runner capabilities includes flash_addr)

Runner-specific options are added through the do_add_parser() hook.

`classmethod args_from_previous_runner`(*previous_runner*, *args: Namespace*)

Update arguments from a previously created runner.

This is intended for propagating relevant user responses between multiple runs of the same runner, for example a JTAG serial number.

`property build_conf: BuildConfiguration`

Get a BuildConfiguration for the build directory.

`call`(*cmd: list[str]*, ***kwargs*) → int

Subclass subprocess.call() wrapper.

Subclasses should use this method to run command in a subprocess and get its return code, rather than using subprocess directly, to keep accurate debug logs.

`classmethod capabilities`() → *RunnerCaps*

Returns a RunnerCaps representing this runner's capabilities.

This implementation returns the default capabilities.

Subclasses should override appropriately if needed.

`cfg`

RunnerConfig for this instance.

`check_call`(*cmd: list[str]*, ***kwargs*)

Subclass subprocess.check_call() wrapper.

Subclasses should use this method to run command in a subprocess and check that it executed correctly, rather than using subprocess directly, to keep accurate debug logs.

`check_call_ignore_sigint`(*client*, ***kwargs*)

Run a command that ignores SIGINT.

`check_output`(*cmd: list[str]*, ***kwargs*) → bytes

Subclass subprocess.check_output() wrapper.

Subclasses should use this method to run command in a subprocess and check that it executed correctly, rather than using subprocess directly, to keep accurate debug logs.

`classmethod create`(*cfg:* RunnerConfig, *args: Namespace*) → *ZephyrBinaryRunner*

Create an instance from command-line arguments.

- `cfg`: runner configuration (pass to superclass __init__)
- `args`: arguments parsed from execution environment, as specified by `add_parser()`.

`classmethod dev_id_help`() → str

Get the ArgParse help text for the –dev-id option.

`abstractmethod classmethod do_add_parser`(*parser*)

Hook for adding runner-specific options.

`abstractmethod classmethod do_create`(*cfg:* RunnerConfig, *args: Namespace*) → *ZephyrBinaryRunner*

Hook for instance creation from command line arguments.

`abstractmethod do_run`(*command: str*, ***kwargs*)

Concrete runner; run() delegates to this. Implement in subclasses.

In case of an unsupported command, raise a ValueError.

`dry_run`

log commands instead of executing them. Can be set by subclasses

`ensure_output`(*output_type: str*) → None

Ensure self.cfg has a particular output artifact.

For example, ensure_output('bin') ensures that self.cfg.bin_file refers to an existing file. Errors out if it's missing or undefined.

Parameters

`output_type` – string naming the output type

`classmethod extload_help`() → str

Get the ArgParse help text for the –extload option.

`static flash_address_from_build_conf`(*build_conf:* BuildConfiguration)

If CONFIG_FLASH_USES_MAPPED_PARTITION is y in build_conf, return the reg address of the zephyr,code-partition devicetree node. Else, if CONFIG_HAS_FLASH_LOAD_OFFSET is n in build_conf, return the CONFIG_FLASH_BASE_ADDRESS value. Otherwise, return CONFIG_FLASH_BASE_ADDRESS + CONFIG_FLASH_LOAD_OFFSET.

`static gdb_ex_args`(*commands: list[str] | None*) → list[str]

Turn GDB commands into '-ex', <command> pairs for a GDB argv.

The commands are passed on verbatim.

`static get_flash_address`(*args: Namespace*, *build_conf:* BuildConfiguration, *default: int = 0*) → int

Helper method for extracting a flash address.

If args.dt_flash is true, returns the address obtained from ZephyrBinaryRunner.flash_address_from_build_conf(build_conf).

Otherwise (when args.dt_flash is False), the default value is returned.

`get_rtt_address`() → int | None

Helper method for extracting a the RTT control block address.

If args.rtt_address was supplied, returns that.

Otherwise, attempt to locate an rtt block in the elf file. If this is not found, None is returned

`static get_runners()` → list[type[*ZephyrBinaryRunner*]]

Get a list of all currently defined runner classes.

`logger`

logging.Logger for this instance.

`abstractmethod classmethod name()` → str

Return this runner's user-visible name.

When choosing a name, pick something short and lowercase, based on the name of the tool (like openocd, jlink, etc.) or the target architecture/board (like xtensa etc.).

`static normalize_chip`(*name: str*) → str

Normalize a chip name for case- and dash-insensitive matching.

`popen_ignore_int`(*cmd: list[str], **kwargs*) → Popen

Spawn a child command, ensuring it ignores SIGINT.

The returned subprocess.Popen object must be manually terminated.

`static require`(*program: str, path: str | None = None*) → str

Require that a program is installed before proceeding.

Parameters

- `program` – name of the program that is required, or path to a program binary.
- `path` – PATH where to search for the program binary. By default check on the system PATH.

If `program` is an absolute path to an existing program binary, this call succeeds. Otherwise, try to find the program by name on the system PATH or in the given PATH, if provided.

If the program can be found, its path is returned. Otherwise, raises MissingProgram.

`static resolve_port_by_chip`(*detect_func, target, logger, priority_vids=None*)

Probe serial ports and return the first one whose connected chip matches the normalized target name.

Parameters

- `detect_func` – callable taking a port device path and returning the chip name reported by that port, or None if no chip could be identified. It is responsible for opening and closing the port.
- `target` – normalized chip name to match against.
- `logger` – logger for progress and diagnostics.
- `priority_vids` – optional list of USB VIDs whose ports are probed first, to avoid resetting unrelated serial adapters.

Returns the matching port device path, or None when no match is found.

`run`(*command: str, **kwargs*)

Runs command ('flash', 'debug', 'debugserver', 'attach').

This is the main entry point to this runner.

`run_server_and_client`(*server, client, **kwargs*)

Run a server that ignores SIGINT, and a client that handles it.

This routine portably:

- creates a Popen object for the `server` command which ignores SIGINT
- runs `client` in a subprocess while temporarily ignoring SIGINT

- cleans up the server after the client exits.
- the keyword arguments, if any, will be passed down to both server and client subprocess calls

It's useful to e.g. open a GDB server and client.

`run_telnet_client`(*host: str*, *port: int*, *active_sock=None*, *send_on_connect: str | None = None*) → None

Run a telnet client for user interaction.

Parameters

`send_on_connect` – Send the given string right after connecting to the given socket. If provided, it will result in avoiding using *nc* as the client.

`static sram_address_from_build_conf`(*build_conf:* BuildConfiguration)

return SRAM address.

`property sysbuild_conf: SysbuildConfiguration`

Get a SysbuildConfiguration for the sysbuild directory.

`property thread_info_enabled: bool`

Returns True if self.build_conf has CONFIG_DEBUG_THREAD_INFO enabled.

`classmethod tool_opt_help`() → str

Get the ArgParse help text for the –tool-opt option.

Doing it By Hand

If you prefer not to use West to flash or debug your board, simply inspect the build directory for the binaries output by the build system. These will be named something like `zephyr/zephyr.elf`, `zephyr/zephyr.hex`, etc., depending on your board's build system integration. These binaries may be flashed to a board using alternative tools of your choice, or used for debugging as needed, e.g. as a source of symbol tables.

By default, these West commands rebuild binaries before flashing and debugging. This can of course also be accomplished using the usual targets provided by Zephyr's build system (in fact, that's how these commands do it).

2.11.12 Signing Binaries

The `west sign` *extension* command can be used to sign a Zephyr application binary for consumption by a bootloader using an external tool. In some configurations, `west sign` is also used to invoke an external, post-processing tool that "stitches" the final components of the image together. Run `west sign -h` for command line help.

rimage

rimage configuration uses an approach that does not rely on Kconfig or CMake but on *west config*, similar to *Permanent CMake Arguments*.

Signing involves a number of "wrapper" scripts stacked on top of each other: `west flash` invokes `west build` which invokes `cmake` and `ninja` which invokes `west sign` which invokes `imgtool` or rimage. As long as the signing parameters desired are the default ones and fairly static, these indirections are not a problem. On the other hand, passing `imgtool` or `rimage` options through all these layers can causes issues typical when the layers don't abstract anything. First, this usually requires boilerplate code in each layer. Quoting whitespace or other special characters through all the wrappers can be difficult. Reproducing a lower `west sign` command to debug some build-time issue can be very time-consuming: it requires at least enabling and searching verbose build logs to find which exact options were used. Copying these options from the build logs can be unreliable: it may produce different results because of subtle environment differences. Last

and worst: new signing feature and options are impossible to use until more boilerplate code has been added in each layer.

To avoid these issues, `rimage` parameters can bet set in `west config`. Here's a `workspace/.west/config` example:

```
[sign]
# Not needed when invoked from CMake
tool = rimage

[rimage]
# Quoting is optional and works like in Unix shells
# Not needed when rimage can be found in the default PATH
path = "/home/me/zworkspace/build-rimage/rimage"

# Not needed when using the default development key
extra-args = -i 4 -k 'keys/key argument with space.pem'
```

In order to support quoting, values are parsed by Python's `shlex.split()` like in *One-Time CMake Arguments*.

The `extra-args` are passed directly to the `rimage` command. The example above has the same effect as appending them on command line after `--` like this: `west sign --tool rimage -- -i 4 -k 'keys/key argument with space.pem'`. In case both are used, the command-line arguments go last.

silabs_commander

The `silabs_commander` tool is used to apply sign or MIC or encrypt binaries for Silicon Labs devices. It can be invoked either by `west sign` when the `sign.tool` configuration is set to `silabs_commander` or by `west build` if `CONFIG_SIWX91X_SIGN_KEY` or `CONFIG_SIWX91X_MIC_KEY` is set.

If one of `CONFIG_SIWX91X_SIGN_KEY` or `CONFIG_SIWX91X_MIC_KEY` is set, `west flash` will automatically flash the signed version of the binary.

`silabs_commander` require Simplicity Commander to be install on the host. The provisionning of the key on the device is described in UG574 SiWx917 SoC Manufacturing Utility User Guide.

2.11.13 Additional Zephyr extension commands

This page documents miscellaneous *Zephyr Extensions*.

Listing boards: `west boards`

The `boards` command can be used to list the boards that are supported by Zephyr without having to resort to additional sources of information.

It can be run by typing:

```
west boards
```

This command lists all supported boards in a default format. If you prefer to specify the display format yourself you can use the `--format` (or `-f`) flag:

```
west boards -f "{arch}:{name}"
```

Additional help about the formatting options can be found by running:

```
west boards -h
```

Shell completion scripts: `west completion`

The `completion` extension command outputs shell completion scripts that can then be used directly to enable shell completion for the supported shells.

It currently supports the following shells:

- bash
- zsh
- fish
- powershell (board qualifiers only)

Additional instructions are available in the command's help:

```
west help completion
```

Installing CMake packages: `west zephyr-export`

This command registers the current Zephyr installation as a CMake config package in the CMake user package registry.

In Windows, the CMake user package registry is found in `HKEY_CURRENT_USER\Software\Kitware\CMake\Packages`.

In Linux and MacOS, the CMake user package registry is found in. `~/.cmake/packages`.

You may run this command when setting up a Zephyr workspace. If you do, application CMakeLists.txt files that are outside of your workspace will be able to find the Zephyr repository with the following:

```
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

See share/zephyr-package/cmake for details.

Software bill of materials: `west spdx`

This command generates a Software Bill of Materials (SBOM) for a Zephyr build as a set of SPDX documents. It records the source files that went into the build, the build artifacts they produced, and the relationships between them. The license and copyright of every file are scanned and filled into the documents, together with file hashes.

Choosing an SPDX version `west spdx` can emit either of the two major SPDX specification families. SPDX 2.3 is the default; select another version with the `--spdx-version` option.

SPDX 2.3 is a superset of 2.2 and adds fields such as `PrimaryPackagePurpose`. Pick SPDX 2.x for compatibility with tooling that does not yet understand SPDX 3; pick SPDX 3.0 or above for the richer, machine-readable build provenance described in *Build profile (SPDX 3.0)*.

> **Note**
>
> SPDX 3.1 support is experimental: the SPDX 3.1 specification is still in development.

Generating SPDX documents

1. Enable `CONFIG_BUILD_OUTPUT_META` in your project, so that the build records what `west spdx` needs.
2. Build your application:

```
west build -d BUILD_DIR [...]
```

3. Generate SPDX documents using this build directory:

```
west spdx -d BUILD_DIR
```

This generates SPDX 2.3 tag-value documents by default. To generate SPDX 2.2 or 3.0 documents instead, pass `--spdx-version`:

```
west spdx -d BUILD_DIR --spdx-version 3.0
```

> **Note**
>
> When building with *Sysbuild (System build)*, make sure you target the actual application which you want to generate the SBOM for. For example, if the application is named `hello_world`:
>
> ```
> west build --sysbuild -d BUILD_DIR
> west spdx -d BUILD_DIR/hello_world
> ```

Output documents The documents are written to `BUILD_DIR/spdx/` (override with `-s`). The same set of bill-of-materials (BOM) documents is produced regardless of the SPDX version; only the file extension differs: `.spdx` for the SPDX 2.x tag-value format and `.jsonld` for the SPDX 3.0 JSON-LD format:

- `app`: BOM for the application source files used for the build
- `zephyr`: BOM for the specific Zephyr source code files used for the build
- `build`: BOM for the built output files
- `modules-deps`: BOM for modules dependencies. Check *modules* for more details.

For SPDX 3.0, every document declares conformance to the Core, Software and Simple Licensing profiles, and `build.jsonld` additionally declares the *Build profile* that captures how the artifacts were produced. Element IDs are full IRIs under the namespace prefix (`-n`), so the documents of separate builds, such as the images of a *sysbuild* build, can be combined into one graph as long as each build has its own prefix, which the default prefix guarantees.

Every file in the bill-of-materials is scanned so that its hashes (SHA512, SHA256 and SHA1) can be recorded, together with its license and copyright.

Both are resolved with the third-party `reuse` tool from the REUSE group, so `west spdx` honours every way the REUSE specification allows them to be declared: an `SPDX-License-Identifier` and `SPDX-FileCopyrightText` comment in the file itself, a `.license` file sitting next to it, or a `REUSE.toml` at the root of the repository annotating whole sets of paths at once. That last form is the only way to license a file that cannot hold a comment of its own, binary blobs in particular.

What is found is written out as `LicenseInfoInFile`, `LicenseConcluded` and `FileCopyrightText` (SPDX 2.x), or as the element's license and copyright properties (SPDX 3.0).

> **Note**
>
> Copyright extraction uses heuristics that may not capture complete notice text, so `FileCopyrightText` content is best-effort. This aligns with SPDX specification recommendations.

Relationships SPDX relationships are created to indicate dependencies between CMake build targets, build targets that are linked together, and source files that are compiled to generate the built library files.

The two specification families express build provenance differently:

- In **SPDX 2.x**, each generated artifact carries file-level `GENERATED_FROM` relationships pointing back at the source (and, with `--analyze-includes`, header) files it was compiled from.
- In **SPDX 3.0**, that provenance is carried by the *Build profile* instead, using `hasInput`/`hasOutput`/`usesTool` relationships scoped to the `build` lifecycle.

Build profile (SPDX 3.0) When generating SPDX 3.0 documents, `west spdx` populates the SPDX 3.0 Build profile so the SBOM records not just *what* was built but *how* it was built. The information is collected automatically from the CMake file-API, and lives in `build.jsonld`.

The profile adds a `build_Build` element describing the overall build: its build type, the CMake generator and build configuration, and selected environment variables such as `BOARD` and `ARCH`. The toolchain (CMake, the compilers, assembler, linker and archiver) is recorded as `Tool` elements, each with its path and version. Build-scoped relationships then link the build to its inputs (the source packages and compiled files), its outputs (the final image) and the tools it used.

Each intermediate target, such as a static library, also gets its own sub-build capturing the exact sources, tools and compile flags that produced its artifact, so any output can be traced back to how it was built.

Command-line options `west spdx` accepts these additional options:

- `-i, --init`: create the CMake file-based API query in a build directory before it is configured. Deprecated, and to be removed in Zephyr 5.0: a build with `CONFIG_BUILD_OUTPUT_META` now requests the query itself.
- `-n PREFIX`: a prefix for the Document Namespaces that will be included in the generated SPDX documents. See SPDX specification clause 6 for details. If `-n` is omitted, a default namespace will be generated according to the default format described in section 2.5 using a random UUID.
- `-s SPDX_DIR`: specifies an alternate directory where the SPDX documents should be written instead of `BUILD_DIR/spdx/`.
- `--spdx-version {2.2,2.3,3.0,3.1}`: specifies which SPDX specification version to use. Defaults to `2.3`. See *Choosing an SPDX version* for the differences between the versions.
- `--analyze-includes`: in addition to recording the compiled source code files (e.g. `.c`, `.S`) in the bills-of-materials, also attempt to determine the specific header files that are included for each `.c` file.

  This takes longer, as it performs a dry run using the C compiler for each `.c` file using the same arguments that were passed to it for the actual build.
- `--include-sdk`: with `--analyze-includes`, also create a fourth SPDX document, `sdk.spdx` (or `sdk.jsonld`), which lists header files included from the SDK.

Warning

The generation of SBOM documents for the `native_sim` platform is currently not supported.

Working with binary blobs: `west blobs`

The `blobs` command allows users to interact with *binary blobs* declared in one or more *modules* via their *module.yml* file.

The `blobs` command has three sub-commands, used to list, fetch or clean (i.e. delete) the binary blobs themselves.

You can list binary blobs while specifying the format of the output:

```
west blobs list -f '{module}: {type} {path}'
```

For the full set of variables available in `-f/--format` run `west blobs -h`.

Fetching blobs works in a similar manner:

```
west blobs fetch
```

Note that, as described in *the modules section*, fetched blobs are stored in a `zephyr/blobs/` folder relative to the root of the corresponding module repository.

As does deleting them:

```
west blobs clean
```

Additionally the tool allows you to specify the modules you want to list, fetch or clean blobs for by typing the module names as a command-line parameter.

The argument `--allow-regex` can be passed `west blobs fetch` to restrict the specific blobs that are fetched, by passing a regular expression:

```
# For example, only download esp32 blobs, skip the other variants
west blobs fetch hal_espressif --allow-regex 'lib/esp32/.*'
```

An auto-cache directory can be provided via the `--auto-cache` cli argument or via the `blobs.auto-cache` config option. When enabled, the auto-cache directory is automatically populated whenever a blob is missing and downloaded.

One or more additional cache directories (separated by `;`) can be provided in `--cache-dirs` cli argument or `blobs.cache-dirs` config option.

`west blobs fetch` searches all configured cache directories (including the auto-cache) for a matching blob filename. Cached files may be stored either under their original filename or with a SHA-256 suffix (`<filename>.<sha>`). If found, the blob is copied from the cache to the blob path; otherwise it is downloaded from its URL(s) to the blob path.

One or more download mirrors can be configured via the `blobs.mirrors` config option. Its value is a JSON object that contains one or more key-value pairs, where each key is a remote URL prefix (a string) and each value is either a single mirror URL prefix (a string) or a list of mirror URL prefixes (an array of strings).

A single mirror for a remote URL prefix:

```
west config blobs.mirrors '{"https://github.com/":
↪"https://example.com/github-mirror/"}'
```

More than one mirror for the same remote URL prefix, tried in list order:

```
west config blobs.mirrors '{
  "https://github.com/": [
    "https://example.com/github1-mirror/",
    "https://example.com/github2-mirror/"
  ]
}'
```

For each blob URL, every mirror whose remote URL prefix matches is tried, ordered so that the *longest* (most specific) matching remote URL prefix is tried first (like git's `insteadOf`); mirrors for shorter, less specific matches are tried afterward. Mirrors configured for the same remote URL prefix are tried in the order they are listed. The original URL is tried last, as a fallback if every mirror fails.

Twister wrapper: `west twister`

This command is a wrapper for *twister*.

Twister can then be invoked via west as follows:

```
west twister -help
west twister -T tests/ztest/base
```

Working with binary descriptors: `west bindesc`

The `bindesc` command allows users to read *binary descriptors* of executable files. It currently supports `.bin`, `.hex`, `.elf` and `.uf2` files as input.

You can search for a specific descriptor in an image, for example:

```
west bindesc search KERNEL_VERSION_STRING build/zephyr/zephyr.bin
```

You can search for a custom descriptor by type and ID, for example:

```
west bindesc custom_search STR 0x200 build/zephyr/zephyr.bin
```

You can dump all of the descriptors in an image using:

```
west bindesc dump build/zephyr/zephyr.bin
```

You can extract the descriptor data area of the image to a file using:

```
west bindesc extract
```

You can list all known standard descriptor names using:

```
west bindesc list
```

You can print the offset of the descriptors inside the image using:

```
west bindesc get_offset
```

Indexing the sources with GNU Global: `west gtags`

> **Important**
>
> You must install the `gtags` and `global` programs provided by GNU Global to use this command.

The `west gtags` command lets you create a GNU Global tags file for the entire west workspace:

```
west gtags
```

This will create a tags file named `GTAGS` in the workspace *topdir* (it will also create other Global-related metadata files named `GPATH` and `GRTAGS` in the same place).

You can then run the `global` command anywhere inside the workspace to search for symbol locations using this tags file.

For example, to search for definitions of the `arch_system_halt()` function, starting from the `zephyr/drivers` directory:

```
$ cd zephyr/drivers
$ global -x arch_system_halt
arch_system_halt   65 ../arch/arc/core/fatal.c FUNC_NORETURN void␣
↪arch_system_halt(unsigned int reason)
arch_system_halt  455 ../arch/arm64/core/fatal.c FUNC_NORETURN void␣
↪arch_system_halt(unsigned int reason)
arch_system_halt  137 ../arch/nios2/core/fatal.c FUNC_NORETURN void␣
↪arch_system_halt(unsigned int reason)
arch_system_halt   18 ../arch/posix/core/fatal.c FUNC_NORETURN void␣
↪arch_system_halt(unsigned int reason)
arch_system_halt   17 ../arch/x86/core/fatal.c FUNC_NORETURN void␣
↪arch_system_halt(unsigned int reason)
arch_system_halt  126 ../arch/xtensa/core/fatal.c FUNC_NORETURN void␣
↪arch_system_halt(unsigned int reason)
arch_system_halt   21 ../kernel/fatal.c FUNC_NORETURN __weak void␣
↪arch_system_halt(unsigned int reason)
```

This prints the search symbol, the line it is defined on, a relative path to the file it is defined in, and the line itself, for all places where the symbol is defined.

Additional tips:

- This can also be useful to search for vendor HAL function definitions.
- See the `global` command's manual page for more information on how to use this tool.
- You should run `global`, **not** `west global`. There is no need for a separate `west global` command since `global` already searches for the `GTAGS` file starting from your current working directory. This is why you need to run `global` from inside the workspace.

Working with patches: `west patch`

The `patch` command allows users to apply patches to Zephyr or Zephyr modules in a controlled manner that makes automation and tracking easier for external applications that use the *T2 star topology*. The *patches.yml* file stores metadata about patch files and fills-in the gaps between official Zephyr releases, so that users can easily see the status of any upstreaming efforts, and determine which patches to drop before upgrading to the next Zephyr release.

There are several sub-commands available to manage patches for Zephyr or other modules in the workspace:

- `apply`: apply patches listed in `patches.yml`
- `reverse`: reverse patches listed in `patches.yml` that have been previously applied
- `clean`: remove all patches that have been applied, and reset to the manifest checkout state
- `list`: list all patches in `patches.yml`
- `gh-fetch`: fetch patches from a GitHub pull request

```
west-workspace/
└── application/
    ...
    ├── west.yml
    └── zephyr
        ├── module.yml
        ├── patches
        │   ├── bootloader
        │   │   └── mcuboot
        │   │       └── my-tweak-for-mcuboot.patch
        │   └── zephyr
```

(continues on next page)

(continued from previous page)

```
      |       └── my-zephyr-change.patch
      └── patches.yml
```

In this example, the *west manifest* file, `west.yml`, would pin to a specific Zephyr revision (e.g. `v4.1.0`) and apply patches against that revision of Zephyr and the specific revisions of other modules used in the application. However, this application needs two changes in order to meet requirements; one for Zephyr and another for MCUBoot.

```yaml
patches:
  - path: zephyr/my-zephyr-change.patch
    sha256sum: c676cd376a4d19dc95ac4e44e179c253853d422b758688a583bb55c3c9137035
    module: zephyr
    author: Obi-Wan Kenobi
    email: obiwan@jedi.org
    date: 2025-05-04
    upstreamable: false
    comments: |
      An application-specific change we need for Zephyr.
  - path: bootloader/mcuboot/my-tweak-for-mcuboot.patch
    sha256sum: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
    module: mcuboot
    author: Darth Sidious
    email: sidious@sith.org
    date: 2025-05-04
    merge-pr: https://github.com/zephyrproject-rtos/zephyr/pull/<pr-number>
    issue: https://github.com/zephyrproject-rtos/zephyr/issues/<issue-number>
    merge-status: true
    merge-commit: 1234567890abcdef1234567890abcdef12345678
    merge-date: 2025-05-06
    apply-command: git apply
    comments: |
      A change to mcuboot that has been merged already. We can remove this
      patch when we are ready to upgrade to the next Zephyr release.
```

Patches can easily be applied in an automated manner. For example:

```
west init -m <manifest repo> <workspace>
cd <workspace>
west update
west patch apply
```

When it is time to update to a newer version of Zephyr, the `west.yml` file can be updated to point at the next Zephyr release, e.g. `v4.2.0`. Patches that are no longer needed, like `my-tweak-for-mcuboot.patch` in the example above, can be removed from `patches.yml` and from the external application repository, and then the following commands can be run.

```
west patch clean
west update
west patch apply --roll-back # roll-back all patches if one does not apply cleanly
```

Optionally, patches can be reversed rather than cleaning all modules. This leaves non-conflicting, unrelated edits to modules in place, but removes only the changes made by patches. This is useful when developing patches and testing them in an application, but not wanting to clean all patches and lose any manual edits made to the module.

```
west patch reverse
```

If a patch needs to be reworked, remember to update the `patches.yml` file with the new SHA256

checksum.

```
sha256sum zephyr/patches/zephyr/my-zephyr-change.patch
7d57ca78d5214f422172cc47fed9d0faa6d97a0796c02485bff0bf29455765e9
```

It is also possible to use `west patch gh-fetch` to fetch patches from a GitHub pull request and automatically create or update the `patches.yml` file. This can be useful when the author already has a number of changes captured in existing upstream pull requests.

```
west patch gh-fetch --owner zephyrproject-rtos --repo zephyr --pull-request
↪<pr-number> \
  --module zephyr --split-commits
```

The above command will create the directory and file structure below, which includes patches for each individual commit associated with the given pull request.

```
zephyr
├── patches
|   ├── first-commit-from-pr.patch
|   ├── second-commit-from-pr.patch
|   └── third-commit-from-pr.patch
└── patches.yml
```

Working with the Zephyr SDK: `west sdk`

The `west sdk` command is a Zephyr-specific west command used to list and install the Zephyr SDK and its toolchains.

Listing SDKs and toolchains To list installed Zephyr SDKs as well as available SDK releases and toolchains, run:

```
west sdk list
```

This command shows:

- Installed SDK versions
- Available SDK releases
- Toolchains included in each SDK

Installing the Zephyr SDK To install the Zephyr SDK, run:

```
west sdk install
```

This command may run in interactive mode and prompt for SDK or toolchain selection. When specific toolchains are provided via `--toolchains`, the command runs non-interactively, which is recommended for automation.

To install specific toolchains only, use the `--toolchains` option:

```
west sdk install --toolchains arm-zephyr-eabi riscv64-zephyr-elf
```

If you are unsure which toolchains you need, run `west sdk list` first to see the available options and avoid downloading unnecessary toolchains, which can save gigabytes of disk space and download time.

2.11.14 History and Motivation

West was added to the Zephyr project to fulfill two fundamental requirements:

- The ability to work with multiple Git repositories
- The ability to provide an extensible and user-friendly command-line interface for basic Zephyr workflows

During the development of west, a set of *Design Constraints* were identified to avoid the common pitfalls of tools of this kind.

Requirements

Although the motivation behind splitting the Zephyr codebase into multiple repositories is outside of the scope of this page, the fundamental requirements, along with a clear justification of the choice not to use existing tools and instead develop a new one, do belong here.

The basic requirements are:

- **R1**: Keep externally maintained code in separately maintained repositories outside of the main zephyr repository, without requiring users to manually clone each of the external repositories
- **R2**: Provide a tool that both Zephyr users and distributors can make use of to benefit from and extend
- **R3**: Allow users and downstream distributions to override or remove repositories without having to make changes to the zephyr repository
- **R4**: Support both continuous tracking and commit-based (bisectable) project updating

Rationale for a custom tool

Some of west's features are similar to those provided by Git Submodules and Google's repo.

Existing tools were considered during west's initial design and development. None were found suitable for Zephyr's requirements. In particular, these were examined in detail:

- Google repo
  - Does not cleanly support using zephyr as the manifest repository (**R4**)
  - Python 2 only
  - Does not play well with Windows
  - Assumes Gerrit is used for code review
- Git submodules
  - Does not fully support **R1**, since the externally maintained repositories would still need to be inside the main zephyr Git tree
  - Does not support **R3**, since downstream copies would need to either delete or replace submodule definitions
  - Does not support continuous tracking of the latest `HEAD` in external repositories (**R4**)
  - Requires hardcoding of the paths/locations of the external repositories

Multiple Git Repositories

Zephyr intends to provide all required building blocks needed to deploy complex IoT applications. This in turn means that the Zephyr project is much more than an RTOS kernel, and is instead a collection of components that work together. In this context, there are a few reasons to work with multiple Git repositories in a standardized manner within the project:

- Clean separation of Zephyr original code and imported projects and libraries
- Avoidance of license incompatibilities between original and imported code
- Reduction in size and scope of the core Zephyr codebase, with additional repositories containing optional components instead of being imported directly into the tree
- Safety and security certifications
- Enforcement of modularization of the components
- Out-of-tree development based on subsets of the supported boards and SoCs

See *Basics* for information on how west workspaces manage multiple git repositories.

Design Constraints

West is:

- **Optional**: it is always *possible* to drop back to "raw" command-line tools, i.e. use Zephyr without using west (although west itself might need to be installed and accessible to the build system). It may not always be *convenient* to do so, however. (If all of west's features were already conveniently available, there would be no reason to develop it.)
- **Compatible with CMake**: building, flashing and debugging, and emulator support will always remain compatible with direct use of CMake.
- **Cross-platform**: West is written in Python 3, and works on all platforms supported by Zephyr.
- **Usable as a Library**: whenever possible, west features are implemented as libraries that can be used standalone in other programs, along with separate command line interfaces that wrap them. West itself is a Python package named `west`; its libraries are implemented as subpackages.
- **Conservative about features**: no features will be accepted without strong and compelling motivation.
- **Clearly specified**: West's behavior in cases where it wraps other commands is clearly specified and documented. This enables interoperability with third party tools, and means Zephyr developers can always find out what is happening "under the hood" when using west.

See Zephyr issue #6205 and for more details and discussion.

`west update` detached HEADs

The use of detached git `HEAD` revisions documented in the *update procedure* confuses and even frustrates some users.

Specifically, users often ask why `west update` doesn't leave existing local branches checked out by default.

This section explains why `west update` behaves the way it does by default, and mentions some other options you have for managing projects that you modified locally.

Two core requirements for `west update` are:

1. safety: the command should not lose any of the user's work
2. determinism: two users running `west update` on the same manifest should get the exact same *workspace* contents

Using detached HEADs helps preserve safety. As documented in the update procedure, using `git checkout --detach` on the updated project to get a detached `HEAD` is a generally safe operation that won't lose the user's work:

- if your work is all safely committed in a local git branch, that branch will be left as-is (see the `git help checkout` output for details)
- if you have uncommitted work, it will be safely kept in your working tree by git if that's possible
- if that's not possible, the entire command will fail, and you can decide what to do with your work before re-running `west update`

Detached HEADs also help ensure determinism:

- Checking out exactly the project revision that's specified in your manifest file is needed to make sure your workspace's files match what's specified in the main manifest. If west did not check out the manifest revision by default, your workspace could be invisibly different from what you would expect given the contents of the manifest files in your working copy. Even worse, the way it was different would depend on exactly what was in your branch before you ran `west update`.
- There is a subtle interaction with *Manifest Imports*. If your project itself has one or more manifest files that are being imported by `west update`, then west needs to check out those files into the working tree so it can resolve the entire imported manifest. If `west update` left a branch checked out in your project, the manifest files in your working tree could be out of date. If west then used those out-of-date files to finish the import, the process could fail unexpectedly or produce other nondeterministic results.

West does not check out the *manifest-rev* branch directly for reasons explained in the documentation for that branch.

This default behavior was chosen a long time ago and it's too late to change it now, since users are depending on it. However, there are other ways to run `west update` if this default doesn't work for you. To learn more, run `west help update` and read the `checked out branch behavior` option group text. For example:

- if you're actively writing some code in your project and you want to keep it up to date after potential changes in the upstream manifest, use `west update --rebase`
- if you've got some local code you'd like to keep around as long as the upstream revision for your project hasn't moved on to something newer, use `west update --keep-descendants`

You can use *West aliases* if you want to set up a shorter way to type one of these commands in your workspace. You can also contribute new options to this option group by sending a pull request to:

https://github.com/zephyrproject-rtos/west

2.11.15 Using Zephyr without west

This page provides information on using Zephyr without west. This is not recommended for beginners due to the extra effort involved. In particular, you will have to do work "by hand" to replace these features:

- cloning the additional source code repositories used by Zephyr in addition to the main zephyr repository, and keeping them up to date
- specifying the locations of these repositories to the Zephyr build system
- flashing and debugging without understanding detailed usage of the relevant host tools

Note

If you have previously installed west and want to stop using it, uninstall it first:

```
pip3 uninstall west
```

Otherwise, Zephyr's build system will find it and may try to use it.

Getting the Source

In addition to downloading the zephyr source code repository itself, you will need to manually clone the additional projects listed in the *west manifest* file inside that repository.

```
mkdir zephyrproject
cd zephyrproject
git clone https://github.com/zephyrproject-rtos/zephyr
# clone additional repositories listed in zephyr/west.yml,
# and check out the specified revisions as well.
```

As you pull changes in the zephyr repository, you will also need to maintain those additional repositories, adding new ones as necessary and keeping existing ones up to date at the latest revisions.

Building applications

You can build a Zephyr application using CMake and Ninja (or make) directly without west installed if you specify any modules manually.

```
cmake -Bbuild -GNinja -DZEPHYR_MODULES=module1;module2;... samples/hello_world
ninja -Cbuild
```

When building with west installed, the Zephyr build system will use it to set *ZEPHYR_MODULES*.

If you don't have west installed and your application does not need any of these repositories, the build will still work.

If you don't have west installed and your application *does* need one of these repositories, you must set `ZEPHYR_MODULES` yourself as shown above.

See *Modules (External projects)* for more details.

Similarly, if your application requires binary blobs and you are not using west, you will need to download and place those blobs in the right places instead of using `west blobs`. See *Binary Blobs* for more details.

Flashing and Debugging

Flashing and debugging are done with the `west flash`, `west debug`, `west debugserver`, `west attach` and `west rtt` commands, which are documented in *Building, Flashing and Debugging*. These commands require west, so they are not available if you are using Zephyr without it.

Without west, you can still flash and debug using any of the *Flash & Debug Host Tools* which work for your board (and which those west commands wrap), but you will have to invoke them yourself, with the right options for your board and application.

For details on west's Python APIs, see west-apis.

2.12 Test Runner (Twister)

Twister scans for the set of test applications in the git repository and attempts to execute them. By default, it tries to build each test application on boards marked as default in the board definition file.

The default options will build the majority of the test applications on a defined set of boards and will run in an emulated environment if available for the architecture or configuration being tested.

Because of the limited test execution coverage, twister cannot guarantee local changes will succeed in the full build environment, but it does sufficient testing by building samples and tests for different boards and different configurations to help keep the complete code tree buildable.

When using (at least) one `-v` option, twister's console output shows for every test application how the test is run (qemu, native_sim, etc.) or whether the binary was just built. The resultant *status* of a test is likewise reported in the `twister.json` and other report files. There are a few reasons why twister only builds a test and doesn't run it:

- The test is marked as `build_only: true` in its `.yaml` configuration file.
- The test configuration has defined a `harness` but you don't have it or haven't set it up.
- The target device is not connected and not available for flashing
- You or some higher level automation invoked twister with `--build-only`.

To run Twister in the local tree, follow the steps below:

```
$ west twister
```

Note

The examples in this document invoke Twister as `west twister`, the *west* extension command, which works the same on all host operating systems. The following invocations are equivalent:

- `west twister ...` (recommended).
- `python .\scripts\twister ...` (Windows): invoking the script directly. This requires the Zephyr environment to be set up first (`source zephyr-env.sh` or `zephyr-env.cmd`).

All forms accept the same command line options.

If you want to run tests on one or more specific platforms, you can use the `--platform` option, it is a platform filter for testing, with this option, test suites will only be built/run on the platforms specified. This option also supports different revisions of one same board, you can use `--platform board@revision` to test on a specific revision.

The list of command line options supported by twister can be viewed with `west twister --help`. See *Command Line Options* for the full set of options.

The following pages cover additional Twister topics:

2.12.1 Command Line Options

The list of command line options supported by Twister can be viewed with `west twister --help`. The full set of options is shown below (rendered at documentation build time):

```
$ $ZEPHYR_BASE/scripts/twister --help
usage: twister [-h] [-E FILENAME] [-F FILENAME] [-T TESTSUITE_ROOT] [-f]
               [--list-tests] [--test-tree] [-G | --emulation-only]
               [--simulation
↪{mdb-nsim,nsim,renode,qemu,tsim,armfvp,xt-sim,native,custom,simics,whisper}]
               [--device-testing] [--pre-script PRE_SCRIPT]
               [--device-serial DEVICE_SERIAL]
               [--device-serial-baud DEVICE_SERIAL_BAUD]
               [--device-serial-pty DEVICE_SERIAL_PTY]
               [--generate-hardware-map GENERATE_HARDWARE_MAP]
               [--hardware-map HARDWARE_MAP] [--persistent-hardware-map]
               [--device-flash-with-test]
```

(continues on next page)

(continued from previous page)

```
                [--device-flash-timeout DEVICE_FLASH_TIMEOUT] [--flash-before]
                [--flash-command FLASH_COMMAND] [--west-flash [WEST_FLASH]]
                [--west-runner WEST_RUNNER] [--west-flash-cmd {flash,debug}]
                [-a ARCH] [--vendor VENDOR] [-p PLATFORM] [-P EXCLUDE_PLATFORM]
                [--platform-pattern PLATFORM_PATTERN]
                [--test-pattern TEST_PATTERN] [--filter {buildable,runnable}]
                [-t TAG] [-e EXCLUDE_TAG] [-s TEST | --sub-test SUB_TEST] [-K]
                [--ignore-platform-key] [--level LEVEL] [-b]
                [--prep-artifacts-for-testing]
                [--package-artifacts PACKAGE_ARTIFACTS] [--test-only]
                [--post-build-checks] [--timeout-multiplier TIMEOUT_MULTIPLIER]
                [--pytest-args PYTEST_ARGS] [--ctest-args CTEST_ARGS]
                [--enable-valgrind | --enable-asan] [-A BOARD_ROOT]
                [--allow-installed-plugin] [-B SUBSET] [--shuffle-tests]
                [--shuffle-tests-seed SHUFFLE_TESTS_SEED] [-c] [--cmake-only]
                [--enable-coverage] [-C] [--gcov-tool GCOV_TOOL]
                [--coverage-basedir COVERAGE_BASEDIR]
                [--coverage-platform COVERAGE_PLATFORM]
                [--coverage-tool {lcov,gcovr}]
                [--coverage-formats COVERAGE_FORMATS] [--coverage-per-instance]
                [--disable-coverage-aggregation] [--coverage-per-test]
                [--test-config TEST_CONFIG] [--disable-suite-name-check]
                [--enable-lsan] [--enable-ubsan] [--force-color]
                [--force-toolchain] [--create-rom-ram-report]
                [--footprint-report [{all,ROM,RAM}]] [--enable-size-report]
                [--footprint-from-buildlog]
                [-m | --compare-report COMPARE_REPORT] [--show-footprint]
                [-H FOOTPRINT_THRESHOLD] [-D] [-z FILENAME] [-i] [-j JOBS] [-l]
                [--list-tags] [--log-file FILENAME] [-M [{pass,all}]]
                [--keep-artifacts KEEP_ARTIFACTS] [-N | -k] [-n]
                [--aggressive-no-clean] [--detailed-test-id]
                [--no-detailed-test-id] [--detailed-skipped-report] [-O OUTDIR]
                [-o REPORT_DIR] [--overflow-as-errors] [--report-filtered]
                [--platform-reports] [--quarantine-list FILENAME]
                [--quarantine-verify] [--quit-on-failure]
                [--report-name REPORT_NAME] [--report-summary [REPORT_SUMMARY]]
                [--report-suffix REPORT_SUFFIX] [--report-all-options]
                [--retry-failed RETRY_FAILED] [--retry-interval RETRY_INTERVAL]
                [--retry-build-errors] [-S] [--enable-slow-only] [--seed SEED]
                [--short-build-path] [--timestamps] [-u] [-v]
                [--console-monitor]
                [-ll {NOTSET,DEBUG,INFO,WARNING,ERROR,CRITICAL}] [-W]
                [-X FIXTURE] [-x EXTRA_ARGS] [-y]
                [--alt-config-root ALT_CONFIG_ROOT]
                ...

positional arguments:
  extra_test_args       Additional args following a '--' are passed to the
                        test binary

options:
  -h, --help            show this help message and exit
  -G, --integration     Run integration tests
  --emulation-only      Only build and run emulation platforms
  --simulation
↪{mdb-nsim,nsim,renode,qemu,tsim,armfvp,xt-sim,native,custom,simics,whisper}
```

(continues on next page)

(continued from previous page)

```
                        Selects which simulation to use. Must match one of the
                        names defined in the board's manifest. If multiple
                        simulator are specified in the selected board and this
                        argument is not passed, then the first simulator is
                        selected.
  --device-testing      Test on device directly. Specify the serial device to
                        use with the --device-serial option.
  --pre-script PRE_SCRIPT
                        specify a pre script. This will be executed before
                        device handler open serial port and invoke runner.
  --device-serial DEVICE_SERIAL
                        Serial device for accessing the board (e.g.,
                        /dev/ttyACM0)
  --device-serial-baud DEVICE_SERIAL_BAUD
                        Serial device baud rate (default 115200)
  --device-serial-pty DEVICE_SERIAL_PTY
                        Script for controlling pseudoterminal. Twister
                        believes that it interacts with a terminal when it
                        actually interacts with the script. E.g "twister
                        --device-testing --device-serial-pty <script>
  --generate-hardware-map GENERATE_HARDWARE_MAP
                        Probe serial devices connected to this platform and
                        create a hardware map file to be used with --device-
                        testing
  --hardware-map HARDWARE_MAP
                        Load hardware map from a file. This will be used for
                        testing on hardware that is listed in the file.
  --persistent-hardware-map
                        With --generate-hardware-map, tries to use persistent
                        names for serial devices on platforms that support
                        this feature (currently only Linux).
  --device-flash-with-test
                        Add a test case timeout to the flash operation timeout
                        when flash operation also executes test case on the
                        platform.
  --device-flash-timeout DEVICE_FLASH_TIMEOUT
                        Set timeout for the device flash operation in seconds.
  --flash-before        Flash device before attaching to serial port. This is
                        useful for devices that share the same port for
                        programming and serial console, or use soft-USB, where
                        flash must come first. Also, it skips reading
                        remaining logs from the old image run.
  --flash-command FLASH_COMMAND
                        Instead of 'west flash', uses a custom flash command
                        to flash when running with --device-testing. Supports
                        comma-separated argument list, the script is also
                        passed a --build-dir flag with the build directory as
                        an argument, and a --board-id flag with the board or
                        probe id if available.
  --west-flash [WEST_FLASH]
                        Comma separated list of additional flags passed to
                        west when running with --device-testing. E.g "twister
                        --device-testing --device-serial /dev/ttyACM0 --west-
                        flash="--board-id=foobar,--erase" will translate to
                        "west flash -- --board-id=foobar --erase"
  --west-runner WEST_RUNNER
```

(continues on next page)

(continued from previous page)

```
                        Uses the specified west runner instead of default when
                        running with --west-flash. E.g "twister --device-
                        testing --device-serial /dev/ttyACM0 --west-flash
                        --west-runner=pyocd" will translate to "west flash
                        --runner pyocd"
  --west-flash-cmd {flash,debug}
                        Uses the specified west command. twister will use
                        flash if not set E.g "twister --device-testing
                        --device-serial /dev/ttyACM0 --west-flash-cmd="flash"
                        will translate to "west flash ..."
  -a ARCH, --arch ARCH  Arch filter for testing. Takes precedence over
                        --platform. If unspecified, test all arches. Multiple
                        invocations are treated as a logical 'or' relationship
  --vendor VENDOR       Vendor filter for testing
  -p PLATFORM, --platform PLATFORM
                        Platform filter for testing. This option may be used
                        multiple times. Test suites will only be built/run on
                        the platforms specified. If this option is not used,
                        then platforms marked as default in the platform
                        metadata file will be chosen to build and test.
  -P EXCLUDE_PLATFORM, --exclude-platform EXCLUDE_PLATFORM
                        Exclude platforms and do not build or run any tests on
                        those platforms. This option can be called multiple
                        times.
  --platform-pattern PLATFORM_PATTERN
                        Platform regular expression filter for testing. This
                        option may be used multiple times. Test suites will
                        only be built/run on the platforms matching the
                        specified patterns. If this option is not used, then
                        platforms marked as default in the platform metadata
                        file will be chosen to build and test.
  --test-pattern TEST_PATTERN
                        Run only the tests matching the specified pattern. The
                        pattern can include regular expressions.
  --filter {buildable,runnable}
                        Filter tests to be built and executed. By default
                        everything is built and if a test is runnable
                        (emulation or a connected device), it is run. This
                        option allows for example to only build tests that can
                        actually be run. Runnable is a subset of buildable.
  -t TAG, --tag TAG     Specify tags to restrict which tests to run by tag
                        value. Default is to not do any tag filtering.
                        Multiple invocations are treated as a logical 'or'
                        relationship.
  -e EXCLUDE_TAG, --exclude-tag EXCLUDE_TAG
                        Specify tags of tests that should not run. Default is
                        to run all tests with all tags.
  -K, --force-platform  Force testing on selected platforms, even if they are
                        excluded in the test configuration (testcase.yaml).
  --ignore-platform-key
                        Do not filter based on platform key
  --level LEVEL         Test level to be used. By default, no levels are used
                        for filtering and do the selection based on existing
                        filters.
  -b, --build-only      Only build the code, do not attempt to run the code on
                        targets.
```

(continues on next page)

(continued from previous page)

```
  --prep-artifacts-for-testing
                        Generate artifacts for testing, do not attempt to run
                        the code on targets.
  --package-artifacts PACKAGE_ARTIFACTS
                        Package artifacts needed for flashing in a file to be
                        used with --test-only
  --test-only           Only run device tests with current artifacts, do not
                        build the code
  --post-build-checks   Run post-build checks on each build directory once a
                        build completes, for example detecting git
                        repositories accidentally cloned into a build
                        directory. Disabled by default; intended to be enabled
                        in CI.
  --timeout-multiplier TIMEOUT_MULTIPLIER
                        Globally adjust tests timeouts by specified
                        multiplier. The resulting test timeout would be
                        multiplication of test timeout value, board-level
                        timeout multiplier and global timeout multiplier (this
                        parameter)
  --pytest-args PYTEST_ARGS
                        Pass additional arguments to the pytest subprocess.
                        This parameter will extend the pytest_args from the
                        harness_config in YAML file.
  --ctest-args CTEST_ARGS
                        Pass additional arguments to the ctest subprocess.
                        This parameter will extend the ctest_args from the
                        harness_config in YAML file.
  --enable-valgrind     Run binary through valgrind and check for several
                        memory access errors. Valgrind needs to be installed
                        on the host. This option only works with host binaries
                        such as those generated for the native_sim
                        configuration and is mutual exclusive with --enable-
                        asan.
  --enable-asan         Enable address sanitizer to check for several memory
                        access errors. Libasan needs to be installed on the
                        host. This option only works with host binaries such
                        as those generated for the native_sim configuration
                        and is mutual exclusive with --enable-valgrind.
  -A BOARD_ROOT, --board-root BOARD_ROOT
                        Directory to search for board configuration files. All
                        .yaml files in the directory will be processed. The
                        directory should have the same structure in the main
                        Zephyr tree: boards/<vendor>/<board_name>/
  --allow-installed-plugin
                        Allow to use pytest plugin installed by pip for pytest
                        tests.
  -B SUBSET, --subset SUBSET
                        Only run a subset of the tests, 1/4 for running the
                        first 25%, 3/5 means run the 3rd fifth of the total.
                        This option is useful when running a large number of
                        tests on different hosts to speed up execution time.
  --shuffle-tests       Shuffle test execution order to get randomly
                        distributed tests.
  --shuffle-tests-seed SHUFFLE_TESTS_SEED
                        Seed value for random generator used to shuffle tests.
                        If not provided, seed in generated by system. Used
```

(continues on next page)

(continued from previous page)

```
                        only when --shuffle-tests is provided.
  -c, --clobber-output  Cleaning the output directory will simply delete it
                        instead of the default policy of renaming.
  --cmake-only          Only run cmake, do not build or run.
  --test-config TEST_CONFIG
                        Path to file with plans and test configurations.
  --disable-suite-name-check
                        Disable extended test suite name verification at the
                        beginning of Ztest test. This option could be useful
                        for tests or platforms, which from some reasons cannot
                        print early logs.
  --enable-lsan         Enable leak sanitizer to check for heap memory leaks.
                        Libasan needs to be installed on the host. This option
                        only works with host binaries such as those generated
                        for the native_sim configuration and when --enable-
                        asan is given.
  --enable-ubsan        Enable undefined behavior sanitizer to check for
                        undefined behaviour during program execution. It uses
                        an optional runtime library to provide better error
                        diagnostics. This option only works with host binaries
                        such as those generated for the native_sim
                        configuration.
  --force-color         Always output ANSI color escape sequences even when
                        the output is redirected (not a tty)
  --force-toolchain     Do not filter based on toolchain, use the set
                        toolchain unconditionally
  -i, --inline-logs     Upon test failure, print relevant log data to stdout
                        instead of just a path to it.
  -j JOBS, --jobs JOBS  Number of jobs for building, defaults to number of CPU
                        threads, overcommitted by factor 2 when --build-only.
  -l, --all             Build/test on all platforms. Any --platform arguments
                        ignored.
  --log-file FILENAME   Specify a file where to save logs.
  -M [{pass,all}], --runtime-artifact-cleanup [{pass,all}]
                        Cleanup test artifacts. The default behavior is 'pass'
                        which only removes artifacts of passing tests. If you
                        wish to remove all artificats including those of
                        failed tests, use 'all'.
  --keep-artifacts KEEP_ARTIFACTS
                        Keep specified artifacts when cleaning up at runtime.
                        Multiple invocations are possible.
  -n, --no-clean        Re-use the outdir before building. Will result in
                        faster compilation since builds will be incremental.
  --aggressive-no-clean
                        Re-use the outdir before building and do not re-run
                        cmake. Will result in much faster compilation since
                        builds will be incremental. This option might result
                        in build failures and inconsistencies if dependencies
                        change or when applied on a significantly changed code
                        base. Use on your own risk. It is recommended to only
                        use this option for local development and when testing
                        localized change in a subsystem.
  --detailed-test-id    Compose each test Suite name from its configuration
                        path (relative to root) and the appropriate Scenario
                        name using PATH_TO_TEST_CONFIG/SCENARIO_NAME schema.
  --no-detailed-test-id
```

(continues on next page)

(continued from previous page)

```
                        Don't prefix each test Suite name with its
                        configuration path, so it is the same as the
                        appropriate Scenario name.
  --detailed-skipped-report
                        Generate a detailed report with all skipped test cases
                        including those that are filtered based on testsuite
                        definition.
  -O OUTDIR, --outdir OUTDIR
                        Output directory for logs and binaries. Default is
                        'twister-out' in the current directory. This directory
                        will be cleaned unless '--no-clean' is set. The '--
                        clobber-output' option controls what cleaning does.
  -o REPORT_DIR, --report-dir REPORT_DIR
                        Output reports containing results of the test run into
                        the specified directory. The output will be both in
                        JSON and JUNIT format (twister.json and twister.xml).
  --overflow-as-errors  Treat RAM/SRAM overflows as errors.
  --report-filtered     Include filtered tests in the reports.
  --platform-reports    Create individual reports for each platform.
  --quarantine-list FILENAME
                        Load list of test scenarios under quarantine. The
                        entries in the file need to correspond to the test
                        scenarios names as in corresponding tests .yaml files.
                        These scenarios will be skipped with quarantine as the
                        reason.
  --quarantine-verify   Use the list of test scenarios under quarantine and
                        run them to verify their current status.
  --quit-on-failure     quit twister once there is build / run failure
  --report-name REPORT_NAME
                        Create a report with a custom name.
  --report-summary [REPORT_SUMMARY]
                        Show failed/error report from latest run. Default
                        shows all items found. However, you can specify the
                        number of items (e.g. --report-summary 15). It also
                        works well with the --outdir switch.
  --report-suffix REPORT_SUFFIX
                        Add a suffix to all generated file names, for example
                        to add a version or a commit ID.
  --report-all-options  Show all command line options applied, including
                        defaults, as environment.options object in
                        twister.json. Default: show only non-default settings.
  --retry-failed RETRY_FAILED
                        Retry failing tests again, up to the number of times
                        specified.
  --retry-interval RETRY_INTERVAL
                        Retry failing tests after specified period of time.
  --retry-build-errors  Retry build errors as well.
  -S, --enable-slow     Execute time-consuming test cases that have been
                        marked as 'slow' in testcase.yaml. Normally these are
                        only built.
  --enable-slow-only    Execute time-consuming test cases that have been
                        marked as 'slow' in testcase.yaml only. This also set
                        the option --enable-slow
  --seed SEED           Seed for native_sim pseudo-random number generator
  --short-build-path    Create shorter build directory paths based on symbolic
                        links. The shortened build path will be used by CMake
```

(continues on next page)

(continued from previous page)

```
                        for generating the build system and executing the
                        build. Use this option if you experience build
                        failures related to path length, for example on
                        Windows OS. This option can be used only with '--
                        ninja' argument (to use Ninja build generator).
  --timestamps          Print all messages with time stamps.
  -u, --no-update       Do not update the results of the last run. This option
                        is only useful when reusing the same output directory
                        of twister, for example when re-running failed tests
                        with --only-failed or --no-clean. This option is for
                        debugging purposes only.
  -v, --verbose         Call multiple times to increase verbosity.
  --console-monitor     Show a live full-screen dashboard for the run in this
                        terminal instead of the normal scrolling output:
                        progress, in-flight jobs with their pipeline stage,
                        and a filterable instance table with per-instance
                        details and logs. Requires an interactive terminal
                        (ignored otherwise). Press q to leave the dashboard;
                        while the run is still going, normal console output
                        resumes, and after it finishes, reports are written as
                        usual. Normal log output goes to twister.log while the
                        dashboard is shown.
  -ll {NOTSET,DEBUG,INFO,WARNING,ERROR,CRITICAL}, --log-level
↪{NOTSET,DEBUG,INFO,WARNING,ERROR,CRITICAL}
                        Select the threshold event severity for which you'd
                        like to receive logs in console. Default is INFO.
  -W, --disable-warnings-as-errors
                        Do not treat warning conditions as errors.
  -X FIXTURE, --fixture FIXTURE
                        Specify a fixture that a board might support.
  -x EXTRA_ARGS, --extra-args EXTRA_ARGS
                        Extra CMake cache entries to define when building test
                        cases. May be called multiple times. The key-value
                        entries will be prefixed with -D before being passed
                        to CMake. E.g "twister -x=USE_CCACHE=0" will translate
                        to "cmake -DUSE_CCACHE=0" which will ultimately
                        disable ccache.
  -y, --dry-run         Create the filtered list of test cases, but don't
                        actually run them. Useful if you're just interested in
                        the test plan generated for every run and saved in the
                        specified output directory (testplan.json).
  --alt-config-root ALT_CONFIG_ROOT
                        Alternative test configuration root/s. When a test is
                        found, Twister will check if a test configuration file
                        exist in any of the alternative test configuration
                        root folders. For example, given
                        $test_root/tests/foo/testcase.yaml, Twister will use
                        $alt_config_root/tests/foo/testcase.yaml if it exists

Test case selection:

  Artificially long but functional example:
      $ ./scripts/twister -v     \
        --testsuite-root tests/ztest/base    \
        --testsuite-root tests/kernel   \
        --test      tests/ztest/base/testing.ztest.verbose_0  \
```

(continues on next page)

(continued from previous page)

```
        --test      tests/kernel/fifo/fifo_api/kernel.fifo

     "kernel.fifo.poll" is one of the test section names in
                                   __/fifo_api/testcase.yaml


  -F FILENAME, --load-tests FILENAME
                        Load a list of tests and platforms to be run from a
                        JSON file ('testplan.json' schema).
  -T TESTSUITE_ROOT, --testsuite-root TESTSUITE_ROOT
                        Base directory to recursively search for test cases.
                        All testcase.yaml files under here will be processed.
                        May be called multiple times. Defaults to the
                        'samples/' and 'tests/' directories at the base of the
                        Zephyr tree.
  -f, --only-failed     Run only those tests that failed the previous twister
                        run invocation.
  -s TEST, --test TEST, --scenario TEST
                        Run only the specified test suite scenario. These are
                        named by 'path/relative/to/Zephyr/base/section.subsect
                        ion_in_testcase_yaml', or just 'section.subsection'
                        identifier. With '--testsuite-root' option the
                        scenario will be found faster.
  --sub-test SUB_TEST   Recursively find sub-test functions (test cases) and
                        run the entire test scenario (section.subsection)
                        where they were found, including all sibling test
                        functions. Sub-tests are named by: 'section.subsection
                        _in_testcase_yaml.ztest_suite.ztest_without_test_prefi
                        x'. Example_1: 'kernel.fifo.fifo_api_1cpu.fifo_loop'
                        where 'kernel.fifo' is a test scenario name
                        (section.subsection) and 'fifo_api_1cpu.fifo_loop' is
                        a Ztest 'suite_name.test_name'. Example_2:
                        'debug.coredump.logging_backend' is a standalone test
                        scenario name. Note: This selection mechanism works
                        only for Ztest suite and test function names in the
                        source files which are not generated by macro-
                        substitutions.
  -N, --ninja           Use the Ninja generator with CMake. (This is the
                        default)
  -k, --make            Use the unix Makefile generator with CMake.

Test plan reporting:
  Report the composed test plan details and exit (dry-run).

  -E FILENAME, --save-tests FILENAME
                        Write a list of tests and platforms to be run to
                        FILENAME file and stop execution. The resulting file
                        will have the same content as 'testplan.json'.
  --list-tests          List of all sub-test functions recursively found in
                        all --testsuite-root arguments. The output is
                        flattened and reports detailed sub-test names without
                        their directories. Note: sub-test names can share the
                        same test scenario identifier prefix
                        (section.subsection) even if they are from different
                        test projects.
  --test-tree           Output the test plan in a tree form.
```

(continues on next page)

```
  --list-tags           List all tags occurring in selected tests.

Memory footprint:
  Collect and report ROM/RAM size footprint for the test instance images built.

  --create-rom-ram-report
                        Generate detailed json reports with ROM/RAM symbol
                        sizes for each test image built using additional build
                        option `--target footprint`.
  --footprint-report [{all,ROM,RAM}]
                        Select which memory area symbols' data to collect as
                        'footprint' property of each test suite built, and
                        report in 'twister_footprint.json' together with the
                        relevant execution metadata the same way as in
                        `twister.json`. Implies '--create-rom-ram-report' to
                        generate the footprint data files. No value means
                        'all'. Default: None
  --enable-size-report  Collect and report ROM/RAM section sizes for each test
                        image built.
  --footprint-from-buildlog
                        Take ROM/RAM sections footprint summary values from
                        the 'build.log' instead of 'objdump' results used
                        otherwise.Requires --enable-size-report or one of the
                        baseline comparison modes.Warning: the feature will
                        not work correctly with sysbuild.
  -m, --last-metrics    Compare footprints to the previous twister invocation
                        as a baseline running in the same output directory.
                        Implies --enable-size-report option.
  --compare-report COMPARE_REPORT
                        Use this report file as a baseline for footprint
                        comparison. The file should be of 'twister.json'
                        schema. Implies --enable-size-report option.
  --show-footprint      With footprint comparison to a baseline, log ROM/RAM
                        section deltas.
  -H FOOTPRINT_THRESHOLD, --footprint-threshold FOOTPRINT_THRESHOLD
                        With footprint comparison to a baseline, warn the user
                        for any of the footprint metric change which is
                        greater or equal to the specified percentage value.
                        Default is 5.0 for 5.0% delta from the new footprint
                        value. Use zero to warn on any footprint metric
                        increase.
  -D, --all-deltas      With footprint comparison to a baseline, warn on any
                        footprint change, increase or decrease. Implies
                        --footprint-threshold=0
  -z FILENAME, --size FILENAME
                        Ignore all other command line options and just produce
                        a report to stdout with ROM/RAM section sizes on the
                        specified binary images.

Code coverage:
  Build with code coverage support, collect code coverage statistics executing␣
↪tests, compose code coverage report at the end.
  Effective for devices with 'HAS_COVERAGE_SUPPORT'.

  --enable-coverage     Enable code coverage collection using gcov.
  -C, --coverage        Generate coverage reports. Implies --enable-coverage
                        to collect the coverage data first.
```

(continues on next page)

(continued from previous page)

```
  --gcov-tool GCOV_TOOL
                        Path to the 'gcov' tool to use for code coverage
                        reports. By default it will be chosen in the following
                        order: using ZEPHYR_TOOLCHAIN_VARIANT ('llvm': from
                        LLVM_TOOLCHAIN_PATH), gcov installed on the host - for
                        'native' or 'unit' platform types, using
                        ZEPHYR_SDK_INSTALL_DIR.
  --coverage-basedir COVERAGE_BASEDIR
                        Base source directory for coverage report.
  --coverage-platform COVERAGE_PLATFORM
                        Platforms to run coverage reports on. This option may
                        be used multiple times. Default to what was selected
                        with --platform.
  --coverage-tool {lcov,gcovr}
                        Tool to use to generate coverage reports (gcovr -
                        default).
  --coverage-formats COVERAGE_FORMATS
                        Output formats to use for generated coverage reports
                        as a comma-separated list without spaces. Valid
                        options for 'gcovr' tool are:
                        html,xml,csv,txt,coveralls,sonarqube (html - default).
                        Valid options for 'lcov' tool are: html,lcov
                        (html,lcov - default).
  --coverage-per-instance
                        Compose individual coverage reports for each test
                        suite when coverage reporting is enabled; it might run
                        in addition to the default aggregation mode which
                        produces one coverage report for all executed test
                        suites. Default: False
  --disable-coverage-aggregation
                        Don't aggregate coverage report statistics for all the
                        test suites selected to run with enabled coverage.
                        Requires another reporting mode to be active
                        (`--coverage-per-instance`) to have at least one of
                        these reporting modes. Default: False
  --coverage-per-test   Collect a separate coverage artifact for every
                        individual Ztest test case, enabling a per-test to
                        source-code coverage matrix. Requires the 'lcov'
                        coverage tool. Implies --coverage. On platforms that
                        support semihosting (e.g. mps2/*) the per-test data is
                        written straight to the host filesystem, avoiding
                        console traffic. Default: False
```

2.12.2 Integration with pytest test framework

Please mind that integration of twister with pytest is still work in progress. Not every platform type is supported in pytest (yet). If you find any issue with the integration or have an idea for an improvement, please, let us know about it and open a GitHub issue/enhancement.

Introduction

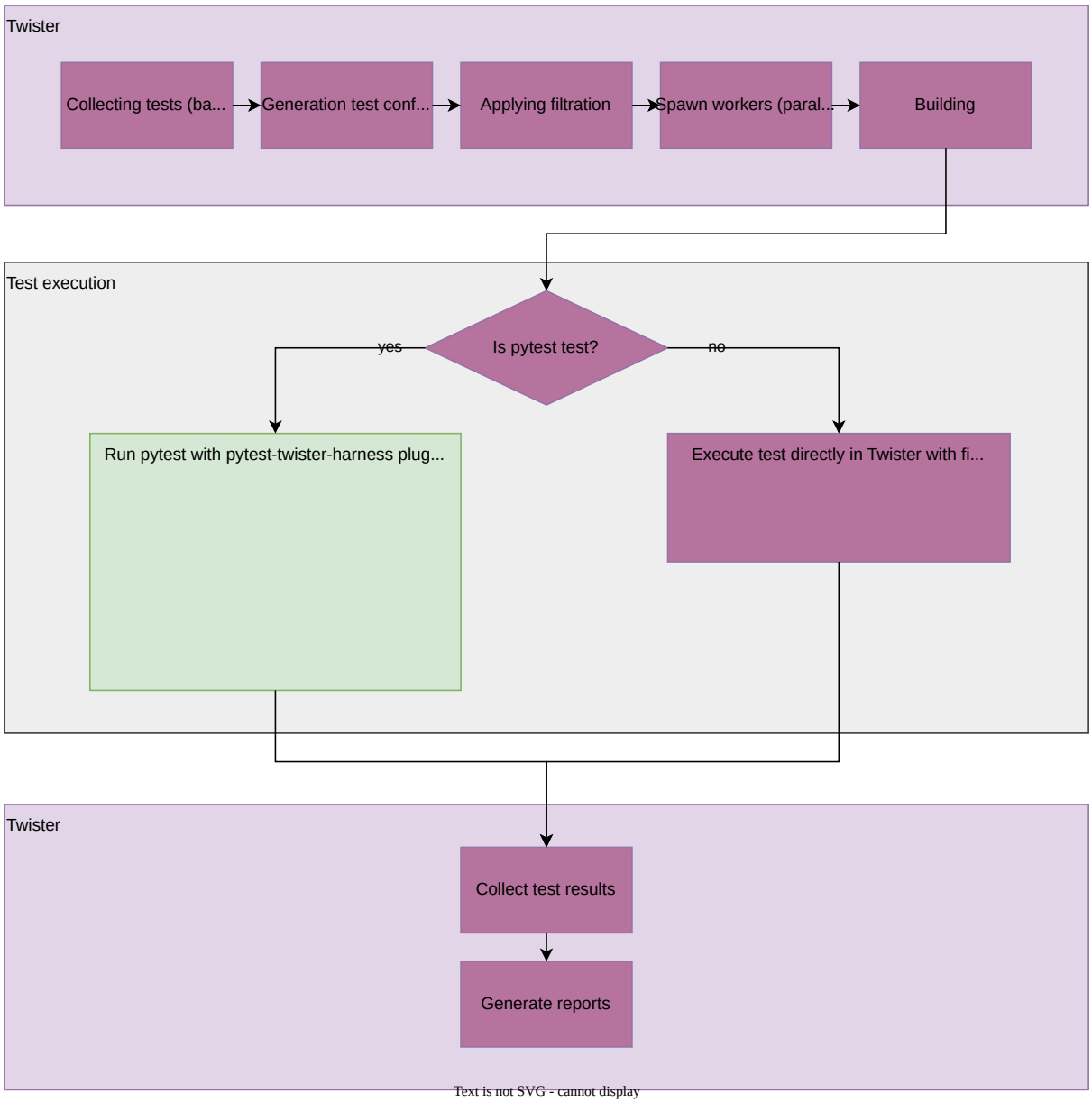
Pytest is a python framework that *"makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries"* (https://docs.pytest.org/en/7.3.x/). Python is known for its free libraries and ease of using it for scripting. In addition, pytest utilizes the concept of plugins and fixtures, increasing its expendability and reusability. A pytest plugin `pytest-twister-harness` was introduced to provide an integration between pytest and

twister, allowing Zephyr's community to utilize pytest functionality with keeping twister as the main framework.

Integration with twister

By default, there is nothing to be done to enable pytest support in twister. The plugin is developed as a part of Zephyr's tree. To enable install-less operation, twister first extends `PYTHONPATH` with path to this plugin, and then during pytest call, it appends the command with `-p twister_harness.plugin` argument. If one prefers to use the installed version of the plugin, they must add `--allow-installed-plugin` flag to twister's call.

Pytest-based test suites are discovered the same way as other twister tests, i.e., by a presence of tests.yaml file. Inside, a keyword `harness` tells twister how to handle a given test. In the case of `harness: pytest`, most of twister workflow (test suites discovery, parallelization, building and reporting) remains the same as for other harnesses. The change happens during the execution step. The below picture presents a simplified overview of the integration.



If `harness: pytest` is used, twister delegates the test execution to pytest, by calling it as a subprocess. Required parameters (such as build directory, device to be used, etc.) are passed through a

YAML configuration file. When pytest is done, twister looks for a pytest report (results.xml) and sets the test result accordingly.

How to create a pytest test

An example folder containing a pytest test, application source code and Twister configuration .yaml file can look like the following:

```
test_foo/
├── pytest/
│   └── test_foo.py
├── src/
│   └── main.c
├── CMakeList.txt
├── prj.conf
└── testcase.yaml
```

An example of a pytest test is given at samples/subsys/testsuite/pytest/shell/pytest/test_shell.py. Using the configuration provided in the `testcase.yaml` file, Twister builds the application from `src` and then, if the .yaml file contains a `harness: pytest` entry, it calls pytest in a separate subprocess. A sample configuration file may look like this:

```yaml
tests:
  some.foo.test:
    harness: pytest
    tags: foo
```

By default, pytest tries to look for tests in a `pytest` directory located next to a directory with binary sources. A keyword `pytest_root` placed under `harness_config` section in .yaml file can be used to point to other files, directories or subtests (more info *here*).

Pytest scans the given locations looking for tests, following its default discovery rules.

Passing extra arguments There are two ways for passing extra arguments to the called pytest subprocess:

1. From .yaml file, using `pytest_args` placed under `harness_config` section - more info *here*.
2. Through Twister command line interface as `--pytest-args` argument. This can be particularly useful when one wants to select a specific testcase from a test suite. For instance, one can use a command:

   ```
   $ west twister --platform native_sim -T samples/subsys/testsuite/pytest/shell␣
   ↪\
   -s samples/subsys/testsuite/pytest/shell/sample.pytest.shell \
   --pytest-args='-k test_shell_print_version'
   ```

   The command line arguments will extend those from the .yaml file. If the same argument is present in both places, the one from the command line will take precedence.

Fixtures

dut Give access to a *DeviceAdapter* type object, that represents Device Under Test. This fixture is the core of pytest harness plugin. It is required to launch DUT (initialize logging, flash device, connect serial etc). This fixture yields a device prepared according to the requested type (`native`, `qemu`, `hardware`, etc.). All types of devices share the same API. This allows for writing tests which are device-type-agnostic. Scope of this fixture is determined by the `pytest_dut_scope` keyword placed under `harness_config` section (more info *here*).

```
from twister_harness import DeviceAdapter

def test_sample(dut: DeviceAdapter):
   dut.readlines_until(regex='Hello world')
```

shell Provide a *Shell* class object with methods used to interact with shell application. It calls `wait_for_promt` method, to not start scenario until DUT is ready. The shell fixture calls `dut` fixture, hence has access to all its methods. The `shell` fixture adds methods optimized for interactions with a shell. It can be used instead of `dut` for tests. Scope of this fixture is determined by the `pytest_dut_scope` keyword placed under `harness_config` section (more info *here*).

```
from twister_harness import Shell

def test_shell(shell: Shell):
   shell.exec_command('help')
```

mcumgr Sample fixture to wrap `mcumgr` command-line tool used to manage remote devices. More information about MCUmgr can be found here *MCUmgr*.

> **Note**
>
> This fixture requires the `mcumgr` available in the system PATH

Only selected functionality of MCUmgr is wrapped by this fixture. For example, here is a test with a fixture `mcumgr`

```
from twister_harness import DeviceAdapter, Shell, McuMgr

def test_upgrade(dut: DeviceAdapter, shell: Shell, mcumgr: McuMgr):
   # free the serial port for mcumgr
   dut.disconnect()
   # upload the signed image
   mcumgr.image_upload('path/to/zephyr.signed.bin')
   # obtain the hash of uploaded image from the device
   second_hash = mcumgr.get_hash_to_test()
   # test a new upgrade image
   mcumgr.image_test(second_hash)
   # reset the device remotely
   mcumgr.reset_device()
   # continue test scenario, check version etc.
```

unlaunched_dut Similar to the `dut` fixture, but it does not initialize the device. It can be used when a finer control over the build process is needed. It becomes responsibility of the test to initialize the device.

```
from twister_harness import DeviceAdapter

def test_sample(unlaunched_dut: DeviceAdapter):
   unlaunched_dut.launch()
   unlaunched_dut.readlines_until(regex='Hello world')
```

Fixtures for Multi-DUT scenarios

Multi-DUT fixtures mirror the single-DUT fixtures (`unlaunched_dut`, `dut`, `shell`) but operate on lists of devices. They work for both hardware and `native_sim` targets; QEMU is not supported. They require `duts` entries to be present in `twister_pytest_config.yaml`. Twister generates this file automatically — for simulation targets it creates placeholder DUT entries so no hardware map is needed — and then calls pytest. When re-running pytest manually without Twister (see FAQ), that same config file is reused. See *Multi-DUTs testing support* section in the Twister documentation for configuration details.

unlaunched_duts Analogous to `unlaunched_dut`, but returns a list of *DeviceAdapter* objects - one per reserved DUT - with log files initialized but devices not yet launched.

```
from twister_harness import DeviceAdapter

def test_sample(unlaunched_duts: list[DeviceAdapter]):
   for dut in unlaunched_duts:
      dut.launch()
   unlaunched_duts[0].readlines_until(regex='Hello world')
```

duts Analogous to `dut`, but returns a list of launched *DeviceAdapter* objects. All devices are flashed concurrently to reduce setup time. If sequential flashing is required, override this fixture in your `conftest.py`.

```
from twister_harness import DeviceAdapter

def test_sample(duts: list[DeviceAdapter]):
   assert len(duts) > 1
   duts[0].readlines_until(regex='Hello world')
   duts[1].readlines_until(regex='Hello world')
```

shells Analogous to `shell`, but returns a list of *Shell* objects - one per reserved DUT - each already waiting for the shell prompt.

```
from twister_harness import Shell

def test_sample(shells: list[Shell]):
   assert len(shells) > 1
   shells[0].exec_command('help')
   shells[1].exec_command('help')
```

Classes

DeviceAdapter

`class twister_harness.DeviceAdapter(device_config: DeviceConfig)`

This class defines a common interface for all device types (hardware, simulator, QEMU) used in tests to gathering device output and send data to it.

`launch() → None`

Launch the test application on the target device.

This method performs the complete device initialization sequence:

- Close any previously running application (cleanup)
- Generate the execution command if not already set
- Add any extra test arguments to the command

- Start reader threads to capture device output
- Launch the application (flash for hardware, execute for simulators)

The launch process varies by device type:

- Hardware devices: Flash the application and establish serial communication. May connect before or after flashing depending on device configuration.
- QEMU/simulators: Start subprocess and establish FIFO/pipe communication
- Native simulators: Execute binary and connect via process pipes

`close()` → None

Disconnect, close device and close reader threads.

`readline`(*connection_index: int = 0, **kwargs*) → str

Read a single line from device output.

Parameters

- `connection_index` – Connection port to read from (0=main UART, 1=second core, etc.)
- `kwargs` – Additional keyword arguments (timeout, print_output)

Returns

Single line of text without trailing newlines

`readlines`(*connection_index: int = 0, **kwargs*) → list[str]

Read all available output lines produced by device from internal buffer.

`readlines_until`(*connection_index: int = 0, **kwargs*) → list[str]

Read lines from device output until a specific condition is met.

This method provides flexible ways to collect device output: wait for a specific pattern, collect a fixed number of lines, or get all currently available lines.

Parameters

- `connection_index` – Index of the connection/port to read from (default: 0). For hardware devices: 0 = main UART, 1 = second core UART, etc. For QEMU and native: always use 0 (only one connection available)
- `kwargs` – Keyword arguments passed to underlying connection including:
  - regex - Regular expression pattern to search for
  - num_of_lines - Exact number of lines to read
  - timeout - Maximum time in seconds to wait (uses base_timeout if not provided)
  - print_output - Whether to log each line as it's read (default: True)

Returns

List of output lines without trailing newlines

Raises

`AssertionError` – If timeout expires before condition is met

`write`(*data: bytes, connection_index: int = 0*) → None

Write data bytes to the target device.

Sends raw bytes to the device through the appropriate communication channel. The underlying transport mechanism varies by device type: Hardware devices write to serial/UART port, QEMU devices write to FIFO queue, and Native simulators write to process stdin pipes.

Parameters

- **data** – Raw bytes to send to the device
- **connection_index** – Index of the connection/port to write to (default: 0)

Shell

class twister_harness.**Shell**(*device:* DeviceAdapter, *prompt: str = 'uart:~$', timeout: float | None = None*)

Helper class that provides methods used to interact with shell application.

exec_command(*command: str, timeout: float | None = None, print_output: bool = True, * (Keyword-only parameters separator (PEP 3102)), get_full_output: bool = False, full_output_timeout: float | None = None*) → list[str]

Send shell command to a device and return response. Passed command is extended by double enter sings - first one to execute this command on a device, second one to receive next prompt what is a signal that execution was finished. Method returns printout of the executed command.

Parameters

- **get_full_output** – If True, after the shell prompt is seen, keep reading lines until `full_output_timeout` elapses (wall clock). Use this when output (e.g. decoded logs) continues to arrive after the prompt.
- **full_output_timeout** – Seconds to spend draining additional lines after the prompt. When `get_full_output` is True and this is omitted, `timeout` (or `base_timeout`) is used.

wait_for_prompt(*timeout: float | None = None*) → bool

Send every 0.5 second "enter" command to the device until shell prompt statement will occur (return True) or timeout will be exceeded (return False).

get_filtered_output(*command_lines: list[str]*) → list[str]

Filter out prompts and log messages

Take the output of exec_command, which can contain log messages and command prompts, and filter them to obtain only the command output.

Example:

```
>>> # equivalent to `lines = shell.exec_command("kernel version")`
>>> lines = [
>>>    'uart:~$',                    # filter prompts
>>>    'Zephyr version 3.6.0',       # keep this line
>>>    'uart:~$ <dbg> debug message' # filter log messages
>>> ]
>>> filtered_output = shell.get_filtered_output(output)
>>> filtered_output
['Zephyr version 3.6.0']
```

Parameters

command_lines – List of strings i.e. the output of *exec_command*.

Returns

A list of strings containing, excluding prompts and log messages.

Examples of pytest tests in the Zephyr project

- pytest_shell
- MCUmgr tests - tests/boot/with_mcumgr
- LwM2M tests - tests/net/lib/lwm2m/interop

- GDB stub tests - tests/subsys/debug/gdbstub

FAQ

How to flash/run application only once per pytest session?

dut is a fixture responsible for flashing/running application. By default, its scope is set as `function`. This can be changed by adding to .yaml file `pytest_dut_scope` keyword placed under `harness_config` section:

```
harness: pytest
harness_config:
   pytest_dut_scope: session
```

More info can be found *here*.

How to run only one particular test from a python file?

This can be achieved in several ways. In .yaml file it can be added using a `pytest_root` entry placed under `harness_config` with list of tests which should be run:

```
harness: pytest
harness_config:
   pytest_root:
      - "pytest/test_shell.py::test_shell_print_help"
```

Particular tests can be also chosen by pytest `-k` option (more info about pytest keyword filter can be found here). It can be applied by adding `-k` filter in `pytest_args` in .yaml file:

```
harness: pytest
harness_config:
   pytest_args:
      - "-k test_shell_print_help"
```

or by adding it to Twister command overriding parameters from the .yaml file:

```
$ west twister ... --pytest-args='-k test_shell_print_help'
```

How to get information about used device type in test?

This can be taken from `dut` fixture (which represents *DeviceAdapter* object):

```
device_type: str = dut.device_config.type
if device_type == 'hardware':
   ...
elif device_type == 'native':
   ...
```

How to rerun locally pytest tests without rebuilding application by Twister?

This can be achieved by running Twister once again with `--test-only` argument added to Twister command. Another way is running Twister with highest verbosity level (`-vv`) and then copy-pasting from logs command dedicated for spawning pytest (log started by `Running pytest command: ...`).

First, run scenario with Twister:

```
$ west twister -vv -ll debug -T samples/subsys/testsuite/pytest/shell \
-s sample.pytest.shell --device-testing -p nrf54l15dk/nrf54l15/cpuapp --
↪device-serial \
/dev/ttyACM1 --west-flash=--erase
```

Twister automatically generates a configuration file `twister_pytest_config.yaml` in the build directory containing all test parameters including device configuration etc.

Then export the required PYTHONPATH environment variable if not already set:

```
$ export PYTHONPATH=$ZEPHYR_BASE/scripts/pylib/pytest-twister-harness/src
```

Finally, run pytest command:

```
$ pytest -s -v -p twister_harness.plugin \
--
↪twister-config=twister-out/.../sample.pytest.shell/twister_pytest_config.yaml␣
↪\
samples/subsys/testsuite/pytest/shell/pytest
```

Is this possible to run pytest tests in parallel?

Basically `pytest-harness-plugin` wasn't written with intention of running pytest tests in parallel. Especially those one dedicated for hardware. There was assumption that parallelization of tests is made by Twister, and it is responsible for managing available sources (jobs and hardwares). If anyone is interested in doing this for some reasons (for example via pytest-xdist plugin) they do so at their own risk.

Limitations

- Not every platform type is supported in the plugin (yet).

2.12.3 Twister Status

What is a Twister Status?

Twister Status formulates the current state of

- `Harness`
- `TestCase`
- `TestSuite`
- `TestInstance`

in a comprehensive and easy-to understand way. In practice, most users will be interested in Statuses of Instances and Cases after the conclusion of their Twister runs.

Tip

Nomenclature reminder:

`Harness`

`Harness` is a Python class inside Twister that allows us to capture and analyse output from a program external to Twister. It has been excised from this page for clarity, as it does not appear in final reports.

`TestCase`

`TestCase`, also called Case, is a piece of code that aims to verify some assertion. It is the smallest subdivision of testing in Zephyr.

`TestSuite`

`TestSuite`, also called Suite, is a grouping of Cases. One can modify Twister's behaviour on a per-Suite basis via `testcase.yaml` files. Such grouped Cases should have enough in common for it to make sense to treat them all the same by Twister.

`TestInstance`

`TestInstance`, also called Instance, is a Suite on some platform. Twister typically reports its results for Instances, despite them being called "Suites" there. If a status is marked as applicable for Suites, it is also applicable for Instances. As the distinction between them is not useful in this section, whenever you read "Suite", assume the same for Instances.

More detailed explanation can be found *here*.

Possible Twister Statuses

Table 6: Twister Statuses

In-code	In-text	Description	Suite	Case
FILTER	filtered	A static or runtime filter has eliminated the test from the list of tests to use.	✓	✓
NOTRUN	not run	The test wasn't run because it was not runnable in current configuration. It was, however, built correctly.	✓	✓
BLOCK	blocked	The test was not run because of an error or crash in the test suite.	×	✓
SKIP	skipped	Test was skipped by some other reason than previously delineated.	✓	✓
ERROR	error	The test produced an error in running the test itself.	✓	✓
FAIL	failed	The test produced results different than expected.	✓	✓
PASS	passed	The test produced results as expected.	✓	✓

In-code and **In-text** are the naming contexts of a given status: the former is rather internal for Twister and appears in logs, whereas the latter is used in the JSON reports.

> **Note**
>
> There are two more Statuses that are internal to Twister. `NONE` (A starting status for Cases and Suites) and `STARTED` (Indicating an in-progress Case). Those should not appear in the final Twister report. Appearance of those non-terminal Statuses in one's report files indicates a problem with Twister.

Case and Suite Status combinations

Table 7: Case and Suite Status combinations

Case\Suite	FILTER	ERROR	FAIL	PASS	NOTRUN	SKIP
FILTER	✓	×	×	×	×	×
ERROR	×	✓	×	×	×	×
BLOCK	×	✓	✓	×	×	×
FAIL	×	✓	✓	×	×	×
PASS	×	✓	✓	✓	×	×
NOTRUN	×	×	×	×	✓	×
SKIP	×	✓	✓	✓	×	✓

× indicates that such a combination should not happen in a proper Twister run. In other words, no Suite of a status indicated by the table column should contain any Cases of a status indicated by the table row.

✓ indicates a proper combination.

Detailed explanation, per Suite Status

`FILTER`:
This status indicates that the whole Suite has been statically filtered out of a given Twister run. Thus, any Case within it should also have such a status.

`ERROR`:
Suite encountered a problem when running the test. It requires at least one case with `ERROR` or `BLOCK` status. As this takes precedence over all other Case statuses, all valid terminal Case statuses can be within such a Suite.

`FAIL`:
Suite has at least one Case that did not meet its assertions. This takes precedence over all other Case statuses, given that the conditions for an ERROR status have not been met.

`PASS`:
Suite has passed properly. It cannot contain any Cases with `BLOCK`, `ERROR`, or `FAIL` statuses, as those indicate a problem when running the Suite.

`NOTRUN`:
Whole suite was not run, but only built. It requires than all Cases within were not run. As runnability is decided on a per-Suite basis, only `NOTRUN` is applicable for its Cases.

`SKIP`:
Whole Suite has been skipped at runtime. All Cases need to have `SKIP` status as well.

2.12.4 Twister blackbox tests

This guide aims to explain the structure of a test file so the reader will be able to understand existing files and create their own. All developers should fix any tests they break and create new ones when introducing new features, so this knowledge is important for any Twister developer.

Basics

Twister blackbox tests are written in python, using the `pytest` library. Read up on it *here* . Auxiliary test data follows whichever format it was in originally. Tests and data are wholly contained in the scripts/tests/twister_blackbox directory and prepended with `test_`.

Blackbox tests should not be aware of the internal twister code. Instead, they should call twister as user would and check the results.

Sample test file

```python
#!/usr/bin/env python3
# Copyright (c) 2024 Intel Corporation
#
# SPDX-License-Identifier: Apache-2.0

import importlib
import json
import os
import sys
from unittest import mock

import pytest
from conftest import TEST_DATA, ZEPHYR_BASE, test_filename_mock
from twisterlib.testplan import TestPlan


class TestDummy:
    TESTDATA_X = [
        ("smoke", 5),
        ("acceptance", 6),
    ]

    @classmethod
    def setup_class(cls):
        apath = os.path.join(ZEPHYR_BASE, "scripts", "twister")
        cls.loader = importlib.machinery.SourceFileLoader("__main__", apath)
        cls.spec = importlib.util.spec_from_loader(cls.loader.name, cls.loader)
        cls.twister_module = importlib.util.module_from_spec(cls.spec)

    @classmethod
    def teardown_class(cls):
        pass

    @pytest.mark.parametrize(
        "level, expected_tests", TESTDATA_X, ids=["smoke", "acceptance"]
    )
    @mock.patch.object(TestPlan, "TEST_DEFINITION_FILENAME", test_filename_mock)
    def test_level(self, capfd, out_path, level, expected_tests):
        # Select platforms used for the tests
        test_platforms = ["qemu_x86", "frdm_k64f"]
        # Select test root
        path = os.path.join(TEST_DATA, "tests")
        config_path = os.path.join(TEST_DATA, "test_config.yaml")

        # Set flags for our Twister command as a list of strs
        args = (
            # Flags related to the generic test setup:
```

(continues on next page)

(continued from previous page)

```
            # * Control the level of detail in stdout/err
            # * Establish the output directory
            # * Select Zephyr tests to use
            # * Control whether to only build or build and run aforementioned tests
            ["-i", "--outdir", out_path, "-T", path, "-y"]
            # Flags under test
            + ["--level", level]
            # Flags required for the test
            + ["--test-config", config_path]
            # Flags related to platform selection
            + [
                val
                for pair in zip(["-p"] * len(test_platforms), test_platforms,␣
↪strict=False)
                for val in pair
            ]
        )

        # First, provide the args variable as our Twister command line arguments.
        # Then, catch the exit code in the sys_exit variable.
        with mock.patch.object(sys, "argv", [sys.argv[0]] + args), pytest.raises(
            SystemExit
        ) as sys_exit:
            # Execute the Twister call itself.
            self.loader.exec_module(self.twister_module)

        # Check whether the Twister call succeeded
        assert str(sys_exit.value) == "0"

        # Access to the test file output
        with open(os.path.join(out_path, "testplan.json")) as f:
            j = json.load(f)
        filtered_j = [
            (ts["platform"], ts["name"], tc["identifier"])
            for ts in j["testsuites"]
            for tc in ts["testcases"]
            if "reason" not in tc
        ]

        # Read stdout and stderr to out and err variables respectively
        out, err = capfd.readouterr()
        # Rewrite the captured buffers to stdout and stderr so the user can still␣
↪read them
        sys.stdout.write(out)
        sys.stderr.write(err)

        # Test-relevant checks
        assert expected_tests == len(filtered_j)
```

Comparison with CLI

Test above runs the command

```
twister -i --outdir $OUTDIR -T $TEST_DATA/tests -y --level $LEVEL
--test-config $TEST_DATA/test_config.yaml -p qemu_x86 -p frdm_k64f
```

It presumes a CLI with the `zephyr-env.sh` or `zephyr-env.cmd` already run.

Such a test provides us with all the outputs we typically expect of a Twister run thanks to `importlib` 's `exec_module()`[1] . We can easily set up all flags that we expect from a Twister call via `args` variable[2] . We can check the standard output or stderr in `out` and `err` variables.

Beside the standard outputs, we can also investigate the file outputs, normally placed in `twister-out` directories. Most of the time, we will use the `out_path` fixture in conjunction with `--outdir` flag (L52) to keep test-generated files in temporary directories. Typical files read in blackbox tests are `testplan.json` , `twister.xml` and `twister.log` .

Other functionalities

Decorators

- `@pytest.mark.usefixtures('clear_log')`
  - allows us to use `clear_log` fixture from `conftest.py` . The fixture is to become `autouse` in the future. After that, this decorator can be removed.
- `@pytest.mark.parametrize('level, expected_tests', TESTDATA_X, ids=['smoke', 'acceptance'])`
  - this is an example of `pytest` 's test parametrization. Read up on it here. TESTDATAs are most often declared as class fields.
- `@mock.patch.object(TestPlan, 'TEST_DEFINITION_FILENAME', test_filename_mock)`
  - this decorator allows us to use only tests defined in the `test_data` and ignore the Zephyr testcases in the `tests` directory. **Note that all "test_data" tests use `test_data.yaml` as a filename, not `testcase.yaml` !** Read up on the `mock` library here.

Fixtures Blackbox tests use `pytest` 's fixtures, further reading on which is available here.

If you would like to add your own fixtures, consider whether they will be used in just one test file, or in many.

- If in many, create such a fixture in the scripts/tests/twister_blackbox/conftest.py file.
  - scripts/tests/twister_blackbox/conftest.py already contains some fixtures - take a look there for an example.
- If in just one, declare it in that file.
  - Consider using class fields instead - look at TESTDATAs for an example.

How do I...

Call Twister multiple times in one test? Sometimes we want to test something that requires prior Twister use. `--test-only` flag would be a typical example, as it is to be coupled with previous `--build-only` Twister call. How should we approach that?

If we just call the `importlib` 's `exec_module` two times, we will experience log duplication. `twister.log` will duplicate every line (triplicate if we call it three times, etc.) instead of overwriting the log or appending to the end of it.

It is caused by the use of logger module variables in the Twister files. Thus us executing the module again causes the loggers to have multiple handles.

To overcome this, between the calls you ought to use

[1] Take note of the `setup_class()` class function, which allows us to run `twister` python file as if it were called directly (bypassing the `__name__ == '__main__'` check).

[2] We advise you to keep the first section of `args` definition intact in almost all of your tests, as it is used for the common test setup.

```
capfd.readouterr()   # To remove output from the buffer
                     # Note that if you want output from all runs after each other,
                     # skip this line.
clear_log_in_test()  # To remove log duplication
```

2.12.5 Board Configuration

To build tests for a specific board and to execute some of the tests on real hardware or in an emulation environment such as QEMU a board configuration file is required which is generic enough to be used for other tasks that require a board inventory with details about the board and its configuration that is only available during build time otherwise.

The board metadata file is located in the board directory and is structured using the YAML markup language. The example below shows a board with a data required for best test coverage for this specific board:

```yaml
identifier: frdm_k64f
name: NXP FRDM-K64F
type: mcu
arch: arm
toolchain:
  - zephyr
  - gnuarmemb
supported:
  - arduino_gpio
  - arduino_i2c
  - netif:eth
  - adc
  - i2c
  - nvs
  - spi
  - gpio
  - usb_device
  - watchdog
  - can
  - pwm
testing:
  default: true
```

identifier:
A string that matches how the board is defined in the build system. This same string is used when building, for example when calling `west build` or `cmake`:

```
# with west
west build -b reel_board
# with cmake
cmake -DBOARD=reel_board ..
```

name:
The actual name of the board as it appears in marketing material.

vendor:
The board vendor. Used by the `vendor_allow` and `vendor_exclude` test scenario filters.

tier:
An optional integer indicating the board support tier. Used for reporting and to group platforms by their level of support.

type:
Type of the board or configuration. One of `mcu`, `qemu`, `sim`, `unit` or `native`.

simulation:
Simulator(s) used to simulate the platform, e.g. qemu.

```
simulation:
  - name: qemu
  - name: armfvp
    exec: FVP_Some_Platform
  - name: custom
    exec: AnotherBinary
```

By default, tests will be executed using the first entry in the simulation array. Another simulation can be selected with `--simulation <simulation_name>`. The `exec` attribute is optional. If it is set but the required simulator is not available, the tests will be built only. If it is not set and the required simulator is not available the tests will fail to run. The simulation name must match one of the elements of `SUPPORTED_EMU_PLATFORMS`.

arch:
Architecture of the board

toolchain:
The list of supported toolchains that can build this board. This should match one of the values used for *`ZEPHYR_TOOLCHAIN_VARIANT`* when building on the command line. Twister filters out any test instance whose toolchain is not in this list, unless `--force-toolchain` is given. This list says which toolchains *may* build the board, it does not select one; see *Toolchain Selection*.

preferred_toolchain:
The toolchain Twister should use for this platform when nothing else selects one. This is useful for boards that are nominally buildable with several toolchains but should be tested with a specific one. See *Toolchain Selection*.

build_toolchains:
An optional list of toolchains that every test assigned to this platform should be built with. Twister creates one test instance per toolchain in the list, each in its own build directory, instead of picking a single toolchain for the platform. For example, to build everything on `native_sim` with both GCC and Clang:

```
build_toolchains:
  - host/gnu
  - host/llvm
```

Because this multiplies build time, it is usually better to leave it out of the board definition and enable it only for CI, using the `build_toolchains` option of the *Twister configuration file*. See *Toolchain Selection*.

ram:
Available RAM on the board (specified in KB). This is used to match test scenario requirements. If not specified we default to 128KB.

flash:
Available FLASH on the board (specified in KB). This is used to match test scenario requirements. If not specified we default to 512KB.

sysbuild: [True|False] (default False)
If true, applications for this platform are built using *sysbuild* by default.

twister: [True|False] (default True)
If false, Twister ignores this platform entirely and never builds or runs tests on it.

supported:
A list of features this board supports. This can be specified as a single word feature or as a

variant of a feature class. For example:

```
supported:
  - pci
```

This indicates the board does support PCI. You can make a test scenario build or run only on such boards, or:

```
supported:
  - netif:eth
  - sensor:bmi16
```

A test scenario can depend on 'eth' to only test ethernet or on 'netif' to run on any board with a networking interface.

testing:
testing relating keywords to provide best coverage for the features of this board.

binaries:
A list of custom binaries to be kept for device testing.

default: [True|False]:
This is a default board, it will tested with the highest priority and is covered when invoking the simplified twister without any additional arguments.

ignore_tags:
Do not attempt to build (and therefore run) tests marked with this list of tags.

only_tags:
Only execute tests with this list of tags on a specific platform.

timeout_multiplier: <float> (default 1)
Multiply each test scenario timeout by specified ratio. This option allows to tune timeouts only for required platform. It can be useful in case naturally slow platform I.e.: HW board with power-efficient but slow CPU or simulation platform which can perform instruction accurate simulation but does it slowly.

flash_before: [True|False] (default False)
For pytest/shell harness hardware testing, flash the device before opening the serial port. This prevents serial port disconnection issues during flashing on some boards (e.g., those with USB CDC that reset during flash operations).

renode:
Configuration for the Renode simulator. Supports two keys: `uart`, the UART peripheral the harness connects to (e.g. `sysbus.uart0`), and `resc`, the Renode script (`.resc`) used to set up the simulated machine.

env:
A list of environment variables. Twister will check if all these environment variables are set, and otherwise skip this platform. This allows the user to define a platform which should be used, for example, only if some required software or hardware is present, and to signal that presence to twister using these environment variables.

variants:
A mapping of board variant (qualifier) names to per-variant overrides. Each entry is itself a platform definition and may override any of the keys above for that specific variant, while inheriting the remaining values from the top-level definition.

2.12.6 Toolchain Selection

Several options influence which toolchain a test is built with. They fall into three groups: options that *select* a toolchain, options that *filter* out test instances whose toolchain is not usable, and options that *multiply* a test into several builds.

Twister first determines a default toolchain for the whole run by invoking `cmake/verify-toolchain.cmake`, which honors the *ZEPHYR_TOOLCHAIN_VARIANT* environment variable. This value is reported at the start of the run as `Using '<toolchain>' toolchain variant.`

For every test scenario and platform pair, the toolchain is then selected using the first of the following that applies:

1. The test scenario's `integration_toolchains`, if set. The test is built once per listed toolchain.
2. The platform's `build_toolchains`, if set, either in the board configuration or in the *Twister configuration file*. The test is built once per listed toolchain.
3. For `posix` and `unit` platforms, `host/llvm` if the run default is `host/llvm`, otherwise `host/gnu`.
4. The platform's `preferred_toolchain`, if set.
5. The run default described above, or `zephyr` if it could not be determined.

Note that *ZEPHYR_TOOLCHAIN_VARIANT* only changes the run default, which is the last entry in this list. It does not override a platform's `preferred_toolchain` or `build_toolchains`, nor a scenario's `integration_toolchains`.

Once a toolchain is selected, the resulting test instance can still be filtered out:

- If the toolchain is not in the platform's `toolchain` list of supported toolchains, the instance is filtered. `--force-toolchain` disables this check and uses the selected toolchain unconditionally. The comparison also succeeds on the part before the `/`, so `host/gnu` matches a platform listing `host`.
- The scenario's `toolchain_allow` and `toolchain_exclude` options filter instances by the selected toolchain.

Because `integration_toolchains` and `build_toolchains` produce one test instance per toolchain, they multiply build time. `build_toolchains` is therefore normally left out of the board configuration and enabled only for the configuration file used by CI.

2.12.7 Tests

Tests are detected by the presence of a `tests.yaml` (`sample.yaml` and `testcase.yaml` support is deprecated) files in the application's project directory. This test application configuration file may contain one or more entries in the `tests:` section each identifying a Test Scenario.

Test application configurations are written using the YAML syntax and share the same structure as samples.

A Test Scenario is a set of conditions and variables defined in a Test Scenario entry, under which a set of Test Suites will be built and executed.

A Test Suite is a collection of Test Cases which are intended to be used to test a software program to ensure it meets certain requirements. The Test Cases in a Test Suite are either related or meant to be executed together.

Test Scenario, Test Suite, and Test Case names must follow to these basic rules:

1. The format of the Test Scenario identifier shall be a string without any spaces or special characters (allowed characters: alphanumeric and [_=]) consisting of multiple sections delimited with a dot (`.`).
2. Each Test Scenario identifier shall start with a section name followed by a subsection names delimited with a dot (`.`). For example, a test scenario that covers semaphores in the kernel shall start with `kernel.semaphore`.
3. All Test Scenario names must be unique for the Twister execution scope.
4. The full canonical name of a Test Suite is: `<Test Application Project path>/<Test Scenario identifier>`

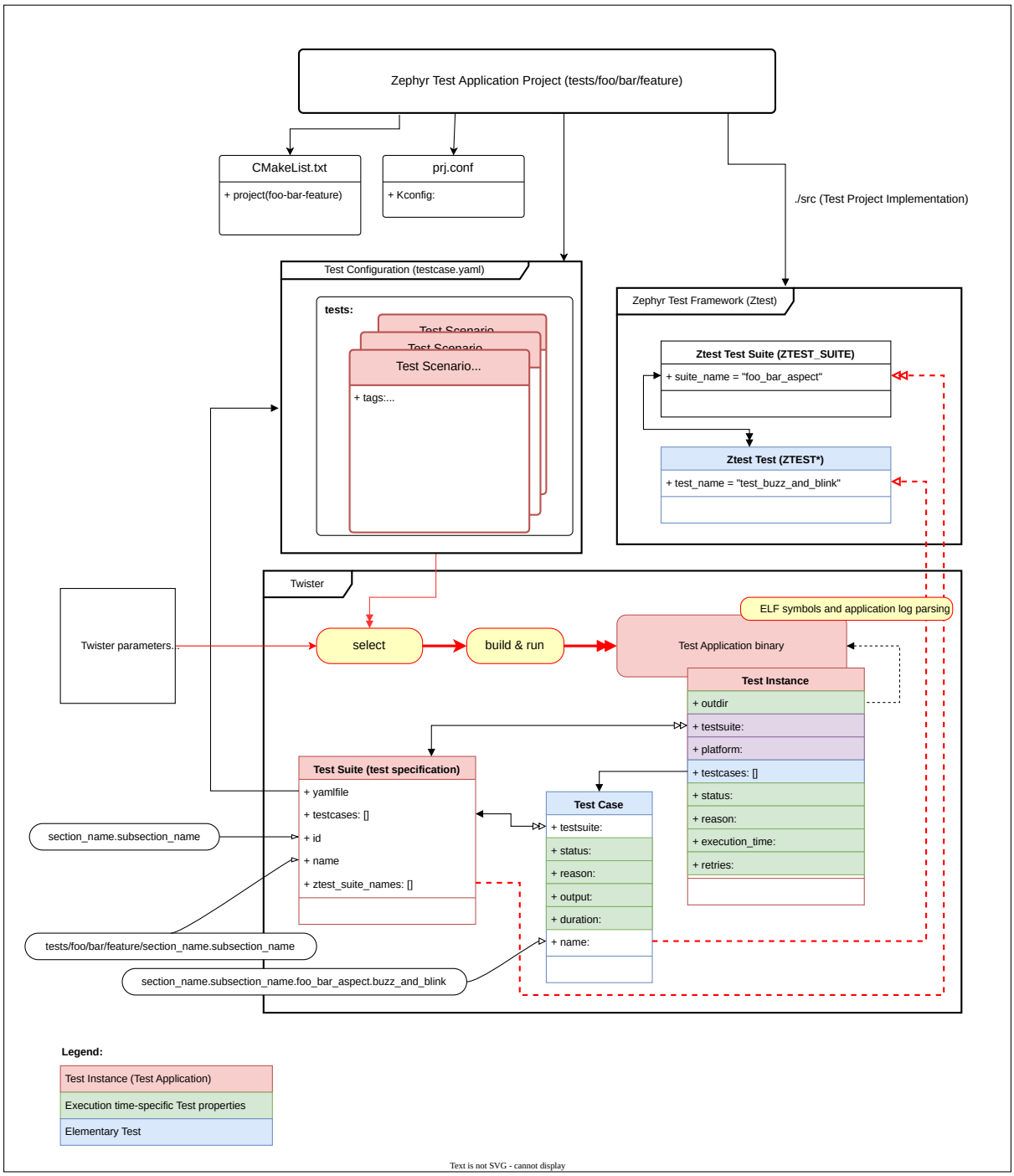


Fig. 4: Twister and a Test application project.

5. Depending on the Test Suite implementation, its Test Case identifiers consist of **at least three sections** delimited with a dot (`.`):
   - **Ztest tests**: a Test Scenario identifier from the corresponding `testcase.yaml` file, a Ztest suite name, and a Ztest test name: `<Test Scenario identifier>.<Ztest suite name>.<Ztest test name>`
   - **Standalone tests and samples**: a Test Scenario identifier from the corresponding `tests.yaml` file where the last section signifies the standalone Test Case name, for example: `debug.coredump.logging_backend`.

The following is an example test configuration with a few options that are explained in this document.

```yaml
tests:
  bluetooth.gatt:
    build_only: true
    platform_allow:
      - qemu_cortex_m3
      - qemu_x86
    tags:
      - bluetooth
  bluetooth.gatt.br:
    build_only: true
    extra_args:
      -CONF_FILE="prj_br.conf"
    filter: not CONFIG_DEBUG
    platform_exclude:
      -up_squared
    platform_allow:
      - qemu_cortex_m3 qemu_x86
    tags:
      - bluetooth
```

A sample with tests will have the same structure with additional information related to the sample and what is being demonstrated:

```yaml
sample:
  name: hello world
  description: Hello World sample, the simplest Zephyr application
tests:
  sample.basic.hello_world:
    build_only: true
    tags:
      - tests
    min_ram: 16
  sample.basic.hello_world.singlethread:
    build_only: true
    extra_args: CONF_FILE=prj_single.conf
    filter: not CONFIG_BT
    tags:
      - tests
    min_ram: 16
```

A Test Scenario entry in the `tests:` YAML dictionary has its Test Scenario identifier as a key.

Each Test Scenario entry in the Test Application configuration can define the following key/value pairs:

tags: <list of tags> (required)
A set of string tags for the test scenario. Usually pertains to functional domains but can be

anything. Command line invocations of this script can filter the set of tests to run based on tag.

skip: <True|False> (default False)
skip test scenario unconditionally. This can be used for broken tests for example.

slow: <True|False> (default False)
Don't run this test scenario unless `--enable-slow` or `--enable-slow-only` was passed in on the command line. Intended for time-consuming test scenarios that are only run under certain circumstances, like daily builds. These test scenarios are still compiled.

extra_args: <list of extra arguments>
Extra arguments to pass to build tool when building or running the test scenario.

Using namespacing, it is possible to apply extra_args only to some hardware. Currently architectures/platforms/simulation are supported:

```
common:
  tags:
   - drivers
   - adc
tests:
  test:
    depends_on: adc
  test_async:
    extra_args:
      - arch:x86:CONFIG_ADC_ASYNC=y
      - platform:qemu_x86:CONFIG_DEBUG=y
      - platform:mimxrt1060_evk:SHIELD=rk043fn66hs_ctg
      - simulation:qemu:CONFIG_MPU=y
```

extra_configs: <list of extra configurations>
Extra configuration options to be merged with a main prj.conf when building or running the test scenario. For example:

```
common:
  tags:
    - drivers
    - adc
tests:
  test:
    depends_on: adc
  test_async:
    extra_configs:
      - CONFIG_ADC_ASYNC=y
```

Using namespacing, it is possible to apply a configuration only to some hardware. Currently both architectures and platforms are supported:

```
common:
  tags:
    - drivers
    - adc
tests:
  test:
    depends_on: adc
  test_async:
    extra_configs:
      - arch:x86:CONFIG_ADC_ASYNC=y
      - platform:qemu_x86:CONFIG_DEBUG=y
```

extra_conf_files: <list of configuration files>
Extra Kconfig fragment files to merge into the build, as an alternative to passing `CONF_FILE=` through `extra_args`. Entries from `common` and the test scenario are concatenated. Prefer this field over `extra_args` for configuration files.

extra_overlay_confs: <list of overlay configuration files>
Extra Kconfig overlay fragments to merge into the build, as an alternative to passing `OVERLAY_CONFIG=` through `extra_args`. Entries from `common` and the test scenario are concatenated.

extra_dtc_overlay_files: <list of devicetree overlay files>
Extra devicetree overlay files to apply to the build, as an alternative to passing `DTC_OVERLAY_FILE=` through `extra_args`. Entries from `common` and the test scenario are concatenated.

build_only: <True|False> (default False)
If true, twister will not try to run the test even if the test is runnable on the platform.

This keyword is reserved for tests that are used to test if some code actually builds. A `build_only` test is not designed to be run in any environment and should not be testing any functionality, it only verifies that the code builds.

This option is often used to test drivers and the fact that they are correctly enabled in Zephyr and that the code builds, for example sensor drivers. Such test shall not be used to verify the functionality of the driver.

build_on_all: <True|False> (default False)
If true, attempt to build test scenario on all available platforms. This is mostly used in CI for increased coverage. Do not use this flag in new tests.

depends_on: <list of features>
A board or platform can announce what features it supports, this option will enable the test only those platforms that provide this feature.

levels: <list of levels>
Test levels this test should be part of. If a level is present, this test will be selectable using the command line option `--level <level name>`

min_ram: <integer>
estimated minimum amount of RAM in KB needed for this test to build and run. This is compared with information provided by the board metadata.

min_flash: <integer>
estimated minimum amount of ROM in KB needed for this test to build and run. This is compared with information provided by the board metadata.

timeout: <number of seconds>
Length of time to run test before automatically killing it. Default to 60 seconds.

arch_allow: <list of arches, such as x86, arm, arc>
Set of architectures that this test scenario should only be run for.

arch_exclude: <list of arches, such as x86, arm, arc>
Set of architectures that this test scenario should not run on.

toolchain_allow: <list of toolchain variants>
Set of toolchain variants that this test scenario should only be run for. The toolchain is the one configured for the run (see *ZEPHYR_TOOLCHAIN_VARIANT*). Platforms built with any other toolchain are filtered out.

toolchain_exclude: <list of toolchain variants>
Set of toolchain variants that this test scenario should not be run for.

vendor_allow: <list of vendors>
Set of platform vendors that this test scenario should only be run for. The vendor is defined

as part of the board definition. Boards associated with this vendors will be included. Other boards, including those without a vendor will be excluded.

vendor_exclude: <list of vendors>
Set of platform vendors that this test scenario should not run on. The vendor is defined as part of the board. Boards associated with this vendors will be excluded.

platform_allow: <list of platforms>
Set of platforms that this test scenario should only be run for. Do not use this option to limit testing or building in CI due to time or resource constraints, this option should only be used if the test or sample can only be run on the allowed platform and nothing else.

integration_platforms: <YML list of platforms/boards>
This option limits the scope to the listed platforms when twister is invoked with the `--integration` option. Use this instead of platform_allow if the goal is to limit scope due to timing or resource constraints.

integration_toolchains: <YML list of toolchain variants>
This option expands the scope to all the listed toolchains variants and adds another vector of testing where desired. By default, test configurations are generated based on the toolchain configured in the environment:

test scenario -> platforms1 -> toolchain1 test scenario -> platforms2 -> toolchain1

When a platform supports multiple toolchains that are available during the twister run, it is possible to expand the test configurations to include additional tests for each toolchain. For example, if a platform supports toolchains `toolchain1` and `toolchain2`, and the test scenario includes:

```
integration_toolchains:
  - toolchain1
  - toolchain2
```

the following configurations are generated:

test scenario -> platforms1 -> toolchain1 test scenario -> platforms1 -> toolchain2 test scenario -> platforms2 -> toolchain1 test scenario -> platforms2 -> toolchain2

> **Note**
> This functionality is evaluated always and is not limited to the `--integration` option.

This option takes precedence over a platform's `build_toolchains`. To expand the toolchain scope for every test on a platform instead of per test scenario, use `build_toolchains`. See *Toolchain Selection*.

platform_exclude: <list of platforms>
Set of platforms that this test scenario should not run on.

platform_type: <list of platform types>
Restrict this test scenario to platforms of the given type(s). A platform's type is declared via the `type:` key in its board metadata. Supported values are `mcu`, `qemu`, `sim`, `unit` and `native`. Platforms whose type is not in this list are filtered out.

simulation_exclude: <list of simulators>
Set of simulators that this test scenario should not run on. Supported values are `qemu`, `simics`, `xt-sim`, `renode`, `nsim`, `mdb-nsim`, `tsim`, `armfvp`, `native` and `custom`.

extra_sections: <list of extra binary sections>
When computing sizes, twister will report errors if it finds extra, unexpected sections in the Zephyr binary unless they are named here. They will not be included in the size calculation.

sysbuild: <True|False> (default False)
Build the project using sysbuild infrastructure. Only the main project's generated device-tree and Kconfig will be used for filtering tests. on device testing must use the hardware map, or west flash to load the images onto the target. The `--erase` option of west flash is not supported with this option. Usage of unsupported options will result in tests requiring sysbuild support being skipped.

harness: <string>
A harness keyword in the `testcase.yaml` file identifies a Twister harness needed to run a test successfully. A harness is a feature of Twister and implemented by Twister, some harnesses are defined as placeholders and have no implementation yet.

A harness can be seen as the handler that needs to be implemented in Twister to be able to evaluate if a test passes criteria. For example, a keyboard harness is set on tests that require keyboard interaction to reach verdict on whether a test has passed or failed, however, Twister lack this harness implementation at the moment.

Supported harnesses:

- ztest
- test
- console
- pytest
- gtest
- robot
- ctest
- shell
- power
- display_capture
- script
- bsim

See *Harnesses* for more information.

platform_key: <list of platform attributes>
Often a test needs to only be built and run once to qualify as passing. Imagine a library of code that depends on the platform architecture where passing the test on a single platform for each arch is enough to qualify the tests and code as passing. The platform_key attribute enables doing just that.

For example to key on (arch, simulation) to ensure a test is run once per arch and simulation (as would be most common):

```
platform_key:
  - arch
  - simulation
```

Adding platform (board) attributes to include things such as soc name, soc family, and perhaps sets of IP blocks implementing each peripheral interface would enable other interesting uses. For example, this could enable building and running SPI tests once for each unique IP block.

harness_config: <harness configuration options>
Extra harness configuration options to be used to select a board and/or for handling generic Console with regex matching. Config can announce what features it supports. This option will enable the test to run on only those platforms that fulfill this external dependency.

fixture: <string or list>
Specify a test scenario dependency on an external device(e.g., sensor), and identify setups that fulfill this dependency. It depends on specific test setup and board selection logic to pick the particular board(s) out of multiple boards that fulfill the dependency in an automation setup based on `fixture` keyword. Some sample fixture names are i2c_hts221, i2c_bme280, i2c_FRAM, ble_fw and gpio_loop.

ztest_suite_repeat: <int> (default 1)
This parameter specifies the number of times the entire test suite should be repeated.

ztest_test_repeat: <int> (default 1)
This parameter specifies the number of times each individual test within the test suite should be repeated.

ztest_test_shuffle: <True|False> (default False)
This parameter indicates whether the order of the tests within the test suite should be shuffled. When set to `true`, the tests will be executed in a random order.

The following is an example yaml file with robot harness_config options.

```
tests:
  robot.example:
    harness: robot
    harness_config:
      robot_testsuite: [robot file path]
```

It can be more than one test suite using a list.

```
tests:
  robot.example:
    harness: robot
    harness_config:
      robot_testsuite:
        - [robot file path 1]
        - [robot file path 2]
        - [robot file path n]
```

One or more options can be passed to robotframework.

```
tests:
  robot.example:
    harness: robot
    harness_config:
      robot_testsuite: [robot file path]
      robot_option:
        - --exclude tag
        - --stop-on-error
```

filter: <expression>
Filter whether the test scenario should be run by evaluating an expression against an environment containing the following values:

```
{ ARCH : <architecture>,
  PLATFORM : <platform>,
  <all CONFIG_* key/value pairs in the test's generated defconfig>,
  *<env>: any environment variable available
}
```

Twister will first evaluate the expression to find if a "limited" cmake call, i.e. using package_helper cmake script, can be done.

Existence of "dt_*" entries indicates devicetree is needed. Refer to *Devicetree Filtering Expressions* for detailed description of the different DT expressions available.

Existence of "CONFIG*" entries indicates kconfig is needed. If there are no other types of entries in the expression a filtration can be done without creating a complete build system. If there are entries of other types a full cmake is required.

The grammar for the expression language is as follows:

```
expression : expression 'and' expression
           | expression 'or' expression
           | 'not' expression
           | '(' expression ')'
           | symbol '==' constant
           | symbol '!=' constant
           | symbol '<' NUMBER
           | symbol '>' NUMBER
           | symbol '>=' NUMBER
           | symbol '<=' NUMBER
           | symbol 'in' list
           | symbol ':' STRING
           | symbol
           ;

list : '[' list_contents ']';

list_contents : constant (',' constant)*;

constant : NUMBER | STRING;
```

For the case where `expression ::= symbol`, it evaluates to `true` if the symbol is defined to a non-empty string.

Operator precedence, starting from lowest to highest:

- or (left associative)
- and (left associative)
- not (right associative)
- all comparison operators (non-associative)

`arch_allow`, `arch_exclude`, `platform_allow`, `platform_exclude` are all syntactic sugar for these expressions. For instance:

```
arch_exclude = x86 arc
```

Is the same as:

```
filter = not ARCH in ["x86", "arc"]
```

The `:` operator compiles the string argument as a regular expression, and then returns a true value only if the symbol's value in the environment matches. For example, if `CONFIG_SOC="stm32f107xc"` then

```
filter = CONFIG_SOC : "stm.*"
```

Would match it.

required_snippets: <list of needed snippets>

Snippets are supported in twister for test scenarios that require them. As with normal applications, twister supports using the base zephyr snippet directory and test application

directory for finding snippets. Listed snippets will filter supported tests for boards (snippets must be compatible with a board for the test to run on them, they are not optional).

The following is an example yaml file with 2 required snippets.

```
tests:
  snippet.example:
    required_snippets:
      - cdc-acm-console
      - user-snippet-example
```

required_applications: <list of required applications> (default empty)

Specify a list of test applications that must be built before current test can run. It enables sharing of built applications between test scenarios, allowing tests to access build artifacts from other applications.

Each required application entry supports:

- `application`: Test scenario identifier (required)
- `name`: Deprecated alias for `application` (still accepted for backward compatibility, but `application` should be used in new configurations)
- `platform`: Target platform (optional, defaults to current test's platform)
- `path`: Directory path where Twister should search for the application (optional). Can be an absolute path or a path relative to the directory containing the test's YAML file. Environment variables and Zephyr module directory variables are expanded (see *Expanding paths with module directory variables*). If not specified, Twister searches in the same directory as the referring test's YAML file.

Required applications are automatically discovered and built by Twister. If a required application is not already loaded, Twister searches for it in the directory specified by `path` or, if `path` is not set, in the same directory as the referring test's YAML file. When reusing build directories (e.g., with `--no-clean`), Twister can find required applications in the current build directory.

How it works:

- Twister builds the required applications first
- The main test application waits for required applications to complete
- Build directories of required applications are made available to the test harness
- For pytest harness, build directories are passed via `--required-build` arguments and accessible through the `required_build_dirs` fixture

When combined with `build: false`, the current test scenario skips its own build step entirely and uses the first required application's build artifacts as its image. This is useful for scenarios that serve purely as test harnesses for an image built elsewhere.

Example configuration:

```
tests:
  # Requires two applications, second one from a different path and with a␣
↪fixed platform
  sample.required_app_demo:
    harness: pytest
    required_applications:
      - application: sample.shared_app
      - application: other.app
        path: ../other_app
        platform: native_sim
  # No self build, use the first required application as the test image
```

(continues on next page)

(continued from previous page)

```
sample.no_self_build:
  build: false
  harness: pytest
  required_applications:
    - application: sample.basic.helloworld
      path: $ZEPHYR_BASE/samples/hello_world
sample.shared_app:
  build_only: true
```

Limitations:

- Not supported with `--runtime-artifact-cleanup`, as build artifacts of required applications must be retained for use by the main test application.
- Not supported with `--subset`: a required application and the test depending on it may be assigned to different subsets, making build artifacts unavailable at test execution time.

build: <True|False> (default True)
If false, the test scenario skips its own build step and uses the build artifacts from the first entry in `required_applications` as its image. This is useful for scenarios that serve purely as a test harness for an image built by another scenario.

Constraints:

- `required_applications` must be non-empty.
- Supported harnesses: pytest-based (e.g. `pytest`, `shell`) and `bsim`.
- QEMU platforms are not supported.

expect_reboot: <True|False> (default False)
Notify twister that the test scenario is expected to reboot while executing. When enabled, twister will suppress warnings about unexpected multiple runs of a testsuite or testcase.

modules: <list of module names>
Build and run this test scenario only when all of the listed *modules* are present in the workspace. Scenarios that require a module which is not available are filtered out.

type: <string> (default integration)
Test type of the scenario. Set to `unit` for unit tests that are built for the *unit_testing board* and run on the host without the full Zephyr build system.

testcases: <list of test case names>
Explicitly declare the list of test case names that make up this scenario. This is normally detected automatically (for example from the ztest source) and only needs to be set for harnesses that cannot be introspected.

ignore_faults: <True|False> (default False)
Do not mark the test scenario as failed if a fault is detected in the output while the test is running.

ignore_qemu_crash: <True|False> (default False)
Do not mark the test scenario as failed if QEMU crashes while the test is running.

The set of test scenarios that actually run depends on directives in the test scenario file and options passed in on the command line. If there is any confusion, running with `-v` or examining the *test plan* (`testplan.json`) can help show why particular test scenarios were filtered out.

To load arguments from a file, add + before the file name, e.g., `+file_name`. File content must be one or more valid arguments separated by line break instead of white spaces.

Most everyday users will run with no arguments.

Expanding paths with module directory variables

Path options in the test scenario file (e.g. `required_applications`, `harness_config: pytest_root`) are expanded before use. In addition to environment variables, Twister expands Zephyr module directory variables, which mirror the CMake variables defined for every module:

- `ZEPHYR_<MODULE>_MODULE_DIR` - absolute path to the module's root.
- `ZEPHYR_<MODULE>_MODULE_NAME` - the module's name.

`<MODULE>` is upper-cased with non-alphanumeric characters replaced by _, exactly as CMake does (for example `$ZEPHYR_HAL_NORDIC_MODULE_DIR` for the `hal_nordic` module). Unknown references are left unchanged.

Managing tests timeouts

There are several parameters which control tests timeouts on various levels:

- `timeout` option in each test scenario. See *here* for more details.
- `timeout_multiplier` option in board configuration. See *here* for more details.
- `--timeout-multiplier` twister option which can be used to adjust timeouts in exact twister run. It can be useful in case of simulation platform as simulation time may depend on the host speed & load or we may select different simulation method (i.e. cycle accurate but slower one), etc...

Overall test scenario timeout is a multiplication of these three parameters.

Devicetree Filtering Expressions

Expressions starting with "dt_*" are used to filter boards based on specific devicetree properties, such as compatibles, aliases, node labels, node properties, chosen nodes, etc. when selecting test scenarios.

> **Note**
>
> The source code for these expressions can be found at scripts/pylib/twister/expr_parser.py.

Expressions

`dt_compat_enabled(compat)`

Purpose:
Checks if any DT node with the specified compatible string (`compat`) is enabled.

Parameters:

- `compat`: The compatible string to match.

`dt_alias_exists(alias)`

Purpose:
Checks if any DT node with the specified alias exists and is enabled.

Parameters:

- `alias`: The alias (defined in `aliases` node) to match.

`dt_enabled_alias_with_parent_compat(alias, compat)`

Purpose:
Checks if the DT has an enabled alias node whose parent has the specified compatible string. Useful for nodes like `gpio-leds` child nodes, which may not have their own compatible.

Parameters:

- `alias`: The alias (defined in `aliases` node) to match.
- `compat`: The parent node's compatible string to match.

`dt_label_with_parent_compat_enabled(label, compat)`

Purpose:
Checks if a DT node with the specified label exists, is enabled, and its parent has the specified compatible string.

Parameters:

- `label`: The node label to match.
- `compat`: The parent node's compatible string to match.

`dt_label_compat_enabled(label, compat)`

Purpose:
Checks if a DT node with the specified label exists, is enabled, and has the specified compatible string.

Parameters:

- `label`: The node label to match.
- `compat`: The node compatible string to match.

`dt_chosen_enabled(chosen)`

Purpose:
Checks if a DT chosen property with the specified name exists and the node assigned to it is enabled.

Parameters:

- `chosen`: The name of the chosen property.

`dt_nodelabel_enabled(label)`

Purpose:
Checks if a DT node with the specified label exists and is enabled.

Parameters:

- `label`: The node label to match.

`dt_nodelabel_prop_enabled(label, prop)`

Purpose:
Checks if a DT node with the specified label exists, is enabled, and has the specified property with a non-empty value.

Parameters:

- `label`: The node label to match.
- `prop`: The node's property to check.

`dt_node_has_prop(node_id, prop)`

Purpose:
Checks if a DT node (specified by alias or path) has the specified property, regardless of its status. Useful for nodes that do not have a status, like `zephyr,user` node.

Parameters:

- `node_id`: The node alias (defined in `aliases` node) or node path to match.
- `prop`: The node's property to check.

Usage These expressions are used in Twister's test scenarios filtering logic to select boards that match specific DT conditions. For example:

```
tests:
  - test: my_test
    filter: dt_compat_enabled("my-compat-string")
```

The test scenario `my_test` will only build for boards where a DT node with `my-compat-string` is enabled.

2.12.8 Harnesses

A *harness* is the mechanism Twister uses to run a test and decide whether it passed or failed. After a test image is built and started on its target (real hardware, an emulator, or the host), the harness drives the interaction with the running image – providing input, capturing output, or handing execution over to an external test runner – and interprets the result to assign a *status* to each test case.

A test scenario selects a harness with the `harness:` entry in its `tests.yaml` and tunes its behavior through `harness_config`. When no harness is specified, the default `test` harness is used. Different harnesses serve different needs: some parse the device's console output against expected patterns, while others delegate execution to an external framework such as pytest, Robot Framework, or ctest. The pages linked below describe each supported harness and its `harness_config` options.

Harnesses `ztest`, `gtest` and `console` are based on parsing of the output and matching certain phrases. `ztest` and `gtest` harnesses look for pass/fail/etc. frames defined in those frameworks.

Some widely used harnesses that are not supported yet:

- keyboard
- net
- bluetooth

The following is an example yaml file with a few harness_config options.

```
sample:
  name: HTS221 Temperature and Humidity Monitor
common:
  tags:
    - sensor
  harness: console
  harness_config:
    type: multi_line
    ordered: false
    regex:
      - "Temperature:(.*)C"
      - "Relative Humidity:(.*)%"
    fixture: i2c_hts221
```

(continues on next page)

(continued from previous page)

```yaml
tests:
  test:
    tags:
      - sensors
    depends_on: i2c
```

Ctest

ctest_args: <list of arguments> (default empty)
Specify a list of additional arguments to pass to `ctest` e.g.: `ctest_args: ['--repeat until-pass:5']`. Note that `--ctest-args` can be passed multiple times to pass several arguments to the ctest.

Gtest

Use `gtest` harness if you've already got tests written in the gTest framework and do not wish to update them to zTest.

Pytest

The *pytest harness* is used to execute pytest test suites in the Zephyr test. The following options apply to the pytest harness:

pytest_root: <list of pytest testpaths> (default pytest)
Specify a list of pytest directories, files or subtests that need to be executed when a test scenario begins to run. The default pytest directory is `pytest`. After the pytest run is finished, Twister will check if the test scenario passed or failed according to the pytest report. Environment variables and Zephyr module directory variables are expanded (see *Expanding paths with module directory variables*). As an example, a list of valid pytest roots is presented below:

```yaml
harness_config:
  pytest_root:
    - "pytest/test_shell_help.py"
    - "../shell/pytest/test_shell.py"
    - "/tmp/test_shell.py"
    - "~/tmp/test_shell.py"
    - "$ZEPHYR_BASE/samples/subsys/testsuite/pytest/shell/pytest/test_shell.py"
    - "$ZEPHYR_HAL_NORDIC_MODULE_DIR/tests/pytest/test_hal.py"  # path inside a module
    - "pytest/test_shell_help.py::test_shell2_sample"  # select pytest subtest
    - "pytest/test_shell_help.py::test_shell2_sample[param_a]"  # select pytest parametrized subtest
```

pytest_args: <list of arguments> (default empty)
Specify a list of additional arguments to pass to `pytest` e.g.: `pytest_args: ['-k=test_method', '--log-level=DEBUG']`. Note that `--pytest-args` can be passed multiple times to pass several arguments to the pytest.

pytest_dut_scope: <function|class|module|package|session> (default function)

The scope for which `dut` and `shell` pytest fixtures are shared. If the scope is set to `function`, DUT is launched for every test case in python script. For `session` scope, DUT is launched only once.

The following is an example yaml file with pytest harness_config options, default pytest_root name "pytest" will be used if pytest_root not specified. please refer the examples in samples/subsys/testsuite/pytest/.

```yaml
common:
  harness: pytest
tests:
  pytest.example.directories:
    harness_config:
      pytest_root:
        - pytest_dir1
        - $ENV_VAR/samples/test/pytest_dir2
  pytest.example.files_and_subtests:
    harness_config:
      pytest_root:
        - pytest/test_file_1.py
        - test_file_2.py::test_A
        - test_file_2.py::test_B[param_a]
```

required_devices: <list of required device entries> (default empty)

Specify additional DUTs required for a multi-DUT test scenario. Each entry configures one extra device to reserve and flash alongside the main DUT. An empty entry `{}` reserves a second device with the same platform and application as the main DUT.

Multi-DUT testing is supported for hardware device testing and `native_sim` simulation. QEMU is not supported. For simulation targets, Twister automatically creates the required placeholder DUT entries, no hardware map is needed. For hardware testing, provide a hardware map file (`--hardware-map`) with a matching entry per required device. More details in *Multi-DUTs testing support* section.

Each entry supports the following optional fields:

platform: <string> (optional, defaults to current test's platform)

Platform to use for this required device. If not specified, the same platform as the main DUT is used.

application: <string> (optional, defaults to current test application)

Test application ID to flash on this required device. When specified, Twister builds the application separately and assigns its build directory to the reserved DUT before flashing. If not specified, the same application as the main DUT is used.

It uses same mechanism as *required_applications*.

path: <string> (optional)

Directory path where Twister should search for the application specified in `application`. Can be an absolute path or a path relative to the directory containing the test's YAML file. Environment variables and Zephyr module directory variables are expanded (see *Expanding paths with module directory variables*). If not specified, Twister searches in the same directory as the referring test's YAML file.

fixture: <list of fixture names> (optional, defaults to empty)

List of fixture names that must be present on the reserved device. See *Fixtures* for details.

Fixtures support an optional parameter suffix using the `name:param` syntax (e.g., `io_adapter:channel_a`). When the main DUT has a fixture with a parameter, Twister uses that parameter value to match required devices - only devices carrying the **same parameter** for that fixture name are considered. This ensures that physically connected DUT pairs (e.g., two boards wired together and registered with the same fixture parameter in the hardware map) are always reserved together and not mixed with unrelated boards.

Example hardware map entries for a paired setup:

```yaml
- id: "01"
  platform: nrf52840dk/nrf52840
  serial: /dev/ttyACM0
  fixtures:
    - io_adapter:channel_a
- id: "02"
  platform: nrf52840dk/nrf52840
  serial: /dev/ttyACM1
  fixtures:
    - io_adapter:channel_a
```

With `fixture: [io_adapter]` on both the main DUT and the required device, Twister selects only boards sharing the same `channel_a` parameter, guaranteeing the physically paired boards are picked.

Example configurations with multiple required devices:

```yaml
tests:
  # Two DUTs, same platform and application
  multidut.basic:
    harness_config:
      required_devices:
        - {}
  # Second DUT fixed to a specific platform
  multidut.fixed_platform:
    harness_config:
      required_devices:
        - platform: nrf52840dk/nrf52840
  # Second DUT flashed with a different application
  multidut.other_app:
    harness_config:
      required_devices:
        - application: multidut.basic
```

Console

The `console` harness tells Twister to parse a test's text output for a regex defined in the test's YAML file.

The following options are currently supported:

type: <one_line|multi_line> (required)
Depends on the regex string to be matched

regex: <list of regular expressions> (required)
Strings with regular expressions to match with the test's output to confirm the test runs as expected.

ordered: <True|False> (default False)
Check the regular expression strings in orderly or randomly fashion

record: <recording options> (optional)

regex: <list of regular expressions> (required)
Regular expressions with named subgroups to match data fields found in the test instance's output lines where it provides some custom data for further analysis. These records will be written into the build directory `recording.csv` file as well as `recording` property of the test suite object in `twister.json`.

With several regular expressions given, each of them will be applied to each output line producing either several different records from the same output line, or different

records from different lines, or similar records from different lines.

The .CSV file will have as many columns as there are fields detected in all records; missing values are filled by empty strings.

For example, to extract three data fields `metric`, `cycles`, `nanoseconds`:

```
record:
  regex:
    - "(?P<metric>.*):(?P<cycles>.*) cycles, (?P<nanoseconds>.*) ns"
```

merge: <True|False> (default False)
Allows to keep only one record in a test instance with all the data fields extracted by the regular expressions. Fields with the same name will be put into lists ordered as their appearance in recordings. It is possible for such multi value fields to have different number of values depending on the regex rules and the test's output.

as_json: <list of regex subgroup names> (optional)
Data fields, extracted by the regular expressions into named subgroups, which will be additionally parsed as JSON encoded strings and written into `twister.json` as nested `recording` object properties. The corresponding `recording.csv` columns will contain JSON strings as-is.

Using this option, a test log can convey layered data structures passed from the test image for further analysis with summary results, traces, statistics, etc.

For example, this configuration:

```
record:
  regex: "RECORD:(?P<type>.*):DATA:(?P<metrics>.*)"
  as_json: [metrics]
```

when matched to a test log string:

```
RECORD:jitter_drift:DATA:{"rollovers":0, "mean_us":1000.0}
```

will be reported in `twister.json` as:

```
"recording":[
    {
          "type":"jitter_drift",
          "metrics":{
              "rollovers":0,
              "mean_us":1000.0
          }
    }
]
```

Robot

Zephyr supports Robot Framework as one of solutions for automated testing.

Robot files allow you to express interactive test scenarios in human-readable text format and execute them in simulation or against hardware. At this moment Zephyr integration supports running Robot tests in the Renode simulation framework, QEMU and Native Simulator.

To execute a Robot test suite with twister, run the following command:

```
$ west twister --platform hifive1 --test␣
↪samples/subsys/shell/shell_module/sample.shell.shell_module.robot
```

Writing Robot tests For the list of keywords provided by the Robot Framework itself, refer to the official Robot documentation.

Information on writing and running Robot Framework tests in Renode can be found in the testing section of Renode documentation. It provides a list of the most commonly used keywords together with links to the source code where those are defined.

It's possible to extend the framework by adding new keywords expressed directly in Robot test suite files, as an external Python library or, like Renode does it, dynamically via XML-RPC. For details see the extending Robot Framework section in the official Robot documentation.

Running a single testsuite To run a single testsuite instead of a whole group of test you can run:

```
$ west twister -p qemu_riscv32 -s arch.shared_interrupt
```

The `robot` harness is used to execute Robot Framework test suites in simulation target (Qemu, Native Simulator, Renode).

robot_testsuite: <robot file path> (default empty)
Specify one or more paths to a file containing a Robot Framework test suite to be run.

robot_option: <robot option> (default empty)
One or more options to be send to robotframework.

Power

The `power` harness is used to measure and validate the current consumption. It integrates with 'pytest' to perform automated data collection and analysis using a hardware power monitor.

The harness executes the following steps:

1. Initializes a power monitoring device (e.g., `stm_powershield`) via the `PowerMonitor` abstract interface.
2. Starts current measurement for a defined `measurement_duration`.
3. Collects raw current waveform data.
4. Uses a peak detection algorithm to segment data into defined execution phases based on power transitions.
5. Computes RMS current values for each phase using a utility function.
6. Compares the computed values with user-defined expected RMS values.

```yaml
harness: power
harness_config:
  fixture: pm_probe
  power_measurements:
    elements_to_trim: 100
    min_peak_distance: 40
    min_peak_height: 0.008
    peak_padding: 40
    measurement_duration: 6
    num_of_transitions: 4
    expected_rms_values: [56.0, 4.0, 1.2, 0.26, 140]
    tolerance_percentage: 20
```

- **elements_to_trim** – Number of samples to discard at the start of measurement to eliminate noise.
- **min_peak_distance** – Minimum distance between detected current peaks (helps detect distinct transitions).

- **min_peak_height** – Minimum current threshold to qualify as a peak (in amps).
- **peak_padding** – Number of samples to extend around each detected peak.
- **measurement_duration** – Total time (in seconds) to record current data.
- **num_of_transitions** – Expected number of power state transitions in the DUT during test execution.
- **expected_rms_values** – Target RMS values for each identified execution phase (in milliamps).
- **tolerance_percentage** – Allowed deviation percentage from the expected RMS values.

Display capture

The `display_capture` harness is used to verify display driver functionality by capturing and analyzing display output using a camera. It integrates with pytest to perform automated visual testing using video fingerprints.

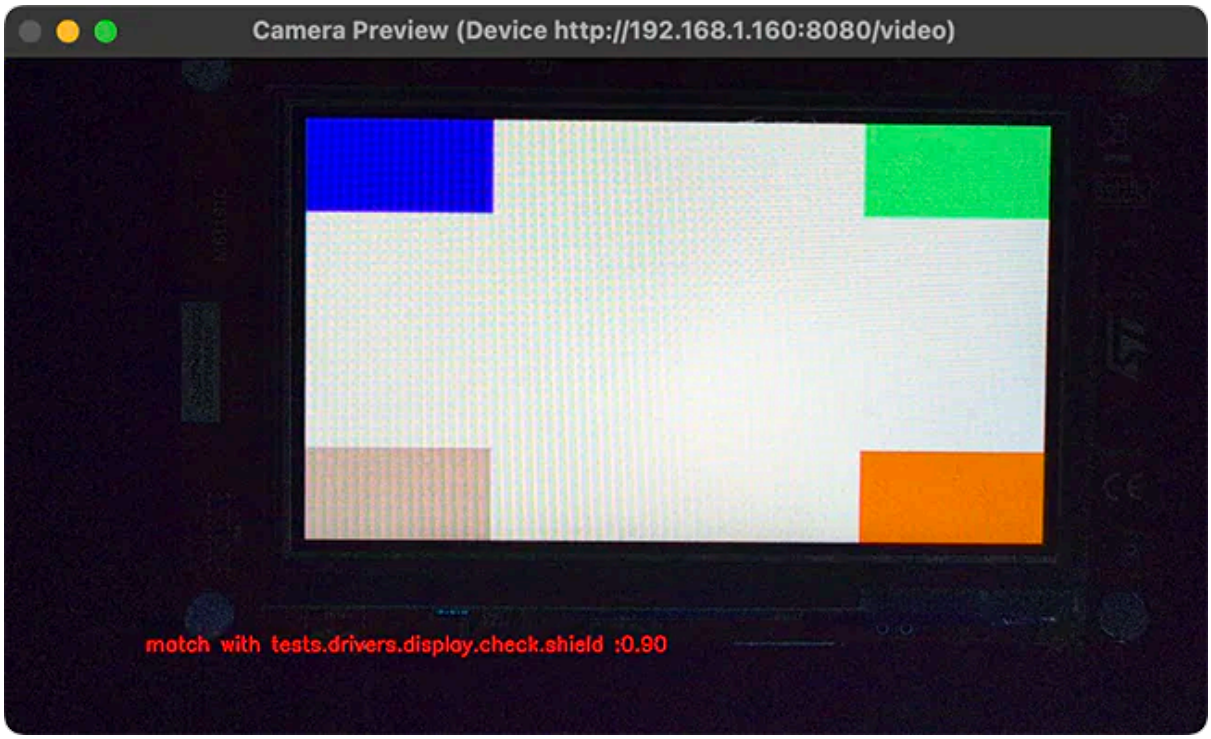


Fig. 5: Window being displayed for a "compare" run where fingerprint is a 90% match with the reference.

Hardware setup The display capture harness requires:

- UVC compatible camera with at least 2 megapixels (e.g., 1080p resolution)
- Light-blocking enclosure or black curtain to ensure consistent lighting
- PC host with camera connection for capturing display output
- DUT connected to the same PC for flashing and serial console access

Configuration The harness uses a YAML configuration file that defines camera settings, test parameters, and video signature analysis options. A typical configuration is shown below:

Listing 1: display_config.yaml

```yaml
case_config:
  device_id: 0
  res_x: 1280
  res_y: 720
  fps: 30
  run_time: 20
test:
  timeout: 30
  prompt: "screen starts"
  expect: ["tests.drivers.display.check.shield"]
plugins:
  - name: signature
    module: plugins.signature_plugin
    class: VideoSignaturePlugin
    status: enable
    config:
      operations: "compare"  # or "generate"
      metadata:
        name: "tests.drivers.display.check.shield"
        platform: "frdm_mcxn947"
      directory: "./fingerprints"
      duration: 100
      method: "combined"
      threshold: 0.65
      phash_weight: 0.35
      dhash_weight: 0.25
      histogram_weight: 0.2
      edge_ratio_weight: 0.1
      gradient_hist_weight: 0.1
```

- `case_config` - This section defines to the general camera settings and duration of the test.
  - `device_id` - The camera device ID (defaults to 0). Any valid OpenCV camera identifier, which can be:
    - An integer for local cameras (use 0 for the first camera, 1 for the second, etc).
    - A device path string such as `/dev/video0` on Linux.
    - An IP video stream URL such as `rtsp://192.168.1.100:8554/stream` for network cameras.
  - `res_x` - The horizontal resolution of the camera (integer, defaults to 1280).
  - `res_y` - The vertical resolution of the camera (integer, defaults to 720).
  - `fps` - The frames per second of the camera (integer, defaults to 30).
  - `run_time` - The duration of the test in seconds (integer, defaults to 20).
- `test` - This section contains the test configuration for device interaction.
  - `timeout` - Maximum time in seconds to wait for the prompt to appear on the device UART output (integer, defaults to 30).
  - `prompt` - The string pattern to wait for in the device UART output before starting the display capture. This can be a regular expression (string, defaults to `uart:~$`).
  - `expect` - A list of expected test result strings that must match the results returned by the application. The test passes if the captured results match this list (list of strings, defaults to `['PASS']`).

- `plugins` - This section contains the configuration for the plugins processing the camera frames. Only the `VideoSignaturePlugin` plugin is currently supported, and it takes the following configuration options:
  - `operations` - The operation to perform when running the test (string). Must be set to either `generate` to capture fingerprints or `compare` to compare the captured fingerprints with the reference fingerprints.
  - `metadata` - Metadata information for fingerprint identification (optional).
    - `name` - Test case name identifier (string).
    - `platform` - Target platform identifier (string).
  - `directory` - The directory where the fingerprints are stored (string, defaults to `./fingerprints`).
  - `duration` - The number of frames to analyze (integer). More frames takes longer but generate more accurate fingerprints).
  - `method` - The method used to generate display fingerprints (string, defaults to `combined`). Must be set to either of the following values: `phash`, `dhash`, `histogram`, or `combined`.

    `phash` (Perceptual Hash)
    Captures overall visual structure and layout. Best for detecting major rendering issues, e.g. UI elements being positioned incorrectly.

    `dhash` (Difference Hash)
    Detects brightness patterns and gradients. Sensitive to contrast changes, e.g. brightness or contrast problems.

    `histogram` (Color Histogram)
    Analyzes color distribution. Fast at detecting obvious color problems, e.g. color swap bugs.

    `combined` (recommended method)
    Weights all methods together (see *`method`*`_weight` option below) for robust comparison. Provides balanced detection of both major and subtle visual issues.
  - `threshold` - The similarity score above which it is considered that there is a match between the reference and the captured fingerprints (optional float, defaults to 0.65).
  - `phash_weight` - The weight for the phash method (optional float, defaults to 0.35)
  - `dhash_weight` - The weight for the dhash method (optional float, defaults to 0.25)
  - `histogram_weight` - The weight for the histogram method (optional float, defaults to 0.2)
  - `gradient_hist_weight` - The weight for the gradient histogram method (optional float, defaults to 0.1)
  - `edge_ratio_weight` - The weight for the edge ratio method (optional float, defaults to 0.1)

The configuration file path is specified in the test's `testcase.yaml` via the `display_capture_config` harness configuration option using the `DISPLAY_TEST_DIR` environment variable:

```yaml
harness: display_capture
harness_config:
  pytest_dut_scope: session
  fixture: fixture_display
  display_capture_config: "${DISPLAY_TEST_DIR}/display_config.yaml"
```

Workflow First, generate **reference fingerprints** for a known-good display output:

```
# Build and flash the display test
west build -b <board> tests/drivers/display/display_check
west flash

# Configure for fingerprint generation mode by setting the 'operations' field to
↪'generate'
# in the configuration file.

# Generate fingerprints
export DISPLAY_TEST_DIR=<path-to-config-directory>
west twister --device-testing --hardware-map map.yml \
    -T tests/drivers/display/display_check/
```

Fingerprints are stored in the directory specified in the `directory` field of the configuration file, and organized by test name and platform as defined in the `metadata` field of the configuration file.

Once the fingerprints have been generated, you can run the test(s) again, this time in **comparison mode**:

```
# Set the 'operations' field to 'compare' in the configuration file.

export DISPLAY_TEST_DIR=<path-to-fingerprints-parent-directory>
west twister --device-testing --hardware-map map.yml \
    -T tests/drivers/display/display_check/
```

The harness compares captured video against reference fingerprints using the configured signature methods and thresholds. If the similarity score between reference and captured fingerprints exceeds the configured `threshold`, the test passes.

Note

- The test name in the DUT's `testcase.yaml` must match the `name` field in the fingerprint's metadata configuration.
- Multiple fingerprints can be stored in one directory for comprehensive validation, though this increases comparison time.
- Fingerprints are specific to both the test scenario and platform.

Script

The `script` harness executes shell scripts as test cases. It resolves scripts from the `tests_scripts` harness configuration option, runs each script as a subprocess, and reports individual pass/fail results based on the script exit code.

The `script` harness also serves as a base class for the *bsim*, *pytest*, and *ctest* harnesses, providing shared subprocess execution, output streaming, and log handling.

tests_scripts: <list of script paths> (default tests_scripts)
Specify a list of shell script paths, relative to the test source directory, that need to be executed when a test scenario runs. Each entry can be a path to a single file, a directory, or a glob pattern. When a directory is specified, all `.sh` files in that directory, including its subdirectories, are collected (excluding files starting with _). The default is the `tests_scripts` directory.

```
harness: script
harness_config:
  tests_scripts:
    - tests_scripts/test_a.sh
    - ../../test/test_b.sh
    - $ENV_VAR/tests_scripts
```

Any extra command-line arguments passed to Twister after `--` are forwarded to every script as additional positional arguments.

Bsim

The `bsim` harness extends the *script harness* to support BabbleSim tests. During the build phase it copies the final executable (`zephyr.exe`) from the build directory to BabbleSim's `bin` directory (`${BSIM_OUT_PATH}/bin`). During the run phase it executes the test scripts listed in `tests_scripts`.

By default, the executable file name is (with dots and slashes replaced by underscores): `bs_<platform_name>_<test_path>_<test_scenario_name>`. This name can be overridden with the `bsim_exe_name` option in `harness_config` section.

Additional `bsim` harness keys extending the *script harness*:

bsim_exe_name: <string>
If provided, the executable filename when copying to BabbleSim's bin directory, will be `bs_<platform_name>_<bsim_exe_name>` instead of the default based on the test path and scenario name.

Example configuration with a multi-images BabbleSim test where the advertiser is build-only and the scanner references it via *required_applications*:

```
common:
  platform_allow:
    - nrf52_bsim/native
  harness: bsim
tests:
  bluetooth.host.adv.extended.advertiser:
    build_only: true
    harness_config:
      bsim_exe_name: tests_bsim_bluetooth_host_adv_extended_prj_advertiser_conf
    extra_args:
      CONF_FILE=prj_advertiser.conf
  bluetooth.host.adv.extended.scanner:
    harness_config:
      bsim_exe_name: tests_bsim_bluetooth_host_adv_extended_prj_scanner_conf
      tests_scripts:
        - tests_scripts/run_adv_extended.sh
    extra_args:
      CONF_FILE=prj_scanner.conf
    required_applications:
      - application: bluetooth.host.adv.extended.advertiser
```

Shell

The shell harness is used to execute shell commands and parse the output and utilizes the pytest framework and the pytest harness of twister.

The following options apply to the shell harness:

shell_commands: <list of pairs of commands and their expected output> (default empty)
Specify a list of shell commands to be executed and their expected output. For example:

```yaml
harness_config:
  shell_commands:
  - command: "kernel cycles"
    expected: "cycles: .* hw cycles"
  - command: "kernel version"
    expected: "Zephyr version .*"
  - command: "kernel sleep 100"
```

If expected output is not provided, the command will be executed and the output will be logged.

shell_commands_file: <string> (default empty)
Specify a file containing test parameters to be used in the test. The file should contain a list of commands and their expected output. For example:

```yaml
- command: "mpu mtest 1"
  expected: "The value is: 0x.*"
- command: "mpu mtest 2"
  expected: "The value is: 0x.*"
```

If no file is specified, the shell harness will use the default file `test_shell.yml` in the test directory. `shell_commands` will take precedence over `shell_commands_file`.

2.12.9 Sidecars

Some tests need a host-side resource to exist for the duration of a run: a daemon the emulated guest talks to, a shared memory region the host reads back afterwards, or a network interface the guest attaches to. A *sidecar* models this. It is selected with the `sidecar:` entry in a test scenario's `tests.yaml` and is orthogonal to the harness: the harness interprets the guest's output while the sidecar provisions the host side around the run. Any harness (`console` for a sample, `ztest` for a test, ...) can therefore be paired with any sidecar.

```yaml
tests:
  some.test:
    harness: ztest
    sidecar: <name>
```

A sidecar has a small lifecycle, driven by Twister for each test instance:

1. **configure** – the sidecar reads what it needs from the instance and its `sidecar_config` block before anything is provisioned.
2. **host check** – at test-plan time the sidecar reports whether the host provides what it needs (for example, that a required daemon binary is installed). When it does not, the test is *built only* and not executed, exactly like a platform whose simulator is not installed.
3. **setup** – called just before the handler runs the test image; it brings the host resource up (starts a daemon, creates an interface, ...). If the host side still turns out to be unavailable – for example bringing the resource up needs privileges that are not present – setup reports this and Twister *skips* execution instead of failing the test.
4. **teardown** – called after the handler returns, in a `finally` block, so it always runs even if the test failed or timed out. It releases the resource and may also collect data the guest left behind (for example reading a shared memory region back into the build directory).

Because provisioning is decoupled from output processing, Twister can also attach a sidecar to an instance itself, without the test opting in – for example to route coverage data off a guest that has no other host transport.

Each sidecar defines its own configuration keys under a block of `sidecar_config` named after the sidecar. Namespacing by sidecar name keeps each sidecar's keys separate, so only the block matching the scenario's `sidecar:` value is consumed. For example, the `virtiofs` sidecar shares a host directory seeded from a template with:

```
tests:
  some.test:
    harness: console
    sidecar: virtiofs
    sidecar_config:
      virtiofs:
        shared: shared
```

2.12.10 Selecting platform scope

One of the key features of Twister is its ability to decide on which platforms a given test scenario should run. This behavior has its roots in Twister being developed as a test runner for Zephyr's CI. With hundreds of available platforms and thousands of tests, the testing tools should be able to adapt the scope and select/filter out what is relevant and what is not.

Twister always prepares an initial list of platforms in scope for a given test, based on command line arguments and the *test's configuration*. Then, platforms that don't fulfill the conditions required in the configuration yaml (e.g. minimum ram) are filtered out from the scope. Using `--force-platform` allows to override filtering caused by `platform_allow`, `platform_exclude`, `arch_allow` and `arch_exclude` keys in test configuration files.

Command line arguments define the initial scope in the following way:

- `-p/--platform <platform_name>` (can be used multiple times): only platforms passed with this argument;
- `-l/--all`: all available platforms;
- `-G/--integration`: all platforms from an `integration_platforms` list in a given test configuration file. If a test has no `integration_platforms` *"scope presumption"* will happen;
- No scope argument: *"scope presumption"* will happen.

"Scope presumption": A list of Twister's *default platforms* is used as the initial list. If nothing is left after the filtration, the `platform_allow` list is used as the initial scope.

2.12.11 Running in Integration Mode

This mode is used in continuous integration (CI) and other automated environments used to give developers fast feedback on changes. The mode can be activated using the `--integration` option of twister and narrows down the scope of builds and tests if applicable to platforms defined under the integration keyword in the test configuration file (`tests.yaml`).

2.12.12 Running tests on custom emulator

Apart from the already supported QEMU and other simulated environments, Twister supports running any out-of-tree custom emulator defined in the board's `board.cmake`. To use this type of simulation, add the following properties to `custom_board/custom_board.yaml`:

```
simulation:
  - name: custom
    exec: <name_of_emu_binary>
```

This tells Twister that the board is using a custom emulator called `<name_of_emu_binary>`, make sure this binary exists in the PATH.

Then, in `custom_board/board.cmake`, set the supported emulation platforms to `custom`:

```
set(SUPPORTED_EMU_PLATFORMS custom)
```

Finally, implement the `run_custom` target in `custom_board/board.cmake`. It should look something like this:

```
add_custom_target(run_custom
  COMMAND
  <name_of_emu_binary to invoke during 'run'>
  <any args to be passed to the command, i.e. ${BOARD}, $
↪{APPLICATION_BINARY_DIR}/zephyr/zephyr.elf>
  WORKING_DIRECTORY ${APPLICATION_BINARY_DIR}
  DEPENDS ${logical_target_for_zephyr_elf}
  USES_TERMINAL
  )
```

2.12.13 Running Tests in Random Order

Enable ZTEST framework's `CONFIG_ZTEST_SHUFFLE` config option to run your tests in random order. This can be beneficial for identifying dependencies between test cases. For native_sim platforms, you can provide the seed to the random number generator by providing `--seed=value` as an argument to twister. See *Shuffling Test Sequence* for more details.

2.12.14 Running Tests on Hardware

Beside being able to run tests in QEMU and other simulated environments, twister supports running most of the tests on real devices and produces reports for each run with detailed FAIL/PASS results.

Executing tests on a single device

To use this feature on a single connected device, run twister with the following new options:

Linux

```
west twister --device-testing --device-serial /dev/ttyACM0 \
--device-serial-baud 115200 -p frdm_k64f  -T tests/kernel
```

Windows

```
west twister --device-testing --device-serial COM1 \
--device-serial-baud 115200 -p frdm_k64f  -T tests/kernel
```

The `--device-serial` option denotes the serial device the board is connected to. This needs to be accessible by the user running twister. You can run this on only one board at a time, specified using the `--platform` option. If the platform supports multiple serial ports, you can provide `--device-serial` multiple times, and it will be passed to the pytest harness. Alternatively you can use the hardware map, see *multi-core testing* for more details

The `--device-serial-baud` option is only needed if your device does not run at 115200 baud.

To support devices without a physical serial port, use the `--device-serial-pty` option. In this cases, log messages are captured for example using a script. In this case you can run twister with the following options:

Linux

```
west twister --device-testing --device-serial-pty "script.py" \
-p intel_adsp/cavs25 -T tests/kernel
```

Windows

> **Note**
>
> Not supported on Windows OS

The script is user-defined and handles delivering the messages which can be used by twister to determine the test execution status.

The `--device-flash-timeout` option allows to set explicit timeout on the device flash operation, for example when device flashing takes significantly large time.

The `--device-flash-with-test` option indicates that on the platform the flash operation also executes a test scenario, so the flash timeout is increased by a test scenario timeout.

Executing tests on multiple devices

To build and execute tests on multiple devices connected to the host PC, a hardware map needs to be created with all connected devices and their details such as the serial device, baud and their IDs if available. Run the following command to produce the hardware map:

```
$ west twister --generate-hardware-map map.yml
```

The generated hardware map file (map.yml) will have the list of connected devices, for example:

Linux

```
- connected: true
  id: "OSHW000032254e4500128002ab98002784d1000097969900"
  platform: unknown
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
- connected: true
  id: "000683759358"
  platform: unknown
  product: J-Link
  runner: unknown
  serial: /dev/cu.usbmodem0006837593581
```

Windows

```
- connected: true
  id: "OSHW000032254e4500128002ab98002784d1000097969900"
  platform: unknown
  product: unknown
  runner: unknown
  serial: COM1
- connected: true
  id: "000683759358"
  platform: unknown
  product: unknown
  runner: unknown
  serial: COM2
```

Any options marked as `unknown` need to be changed and set with the correct values, in the above example the platform names, the products and the runners need to be replaced with the correct values corresponding to the connected hardware. In this example we are using a reel_board and an nrf52840dk/nrf52840:

Linux

```yaml
- connected: true
  id: "OSHW000032254e4500128002ab98002784d1000097969900"
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/cu.usbmodem146114202
  baud: 9600
- connected: true
  id: "000683759358"
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: nrfjprog
  serial: /dev/cu.usbmodem0006837593581
  baud: 9600
```

Windows

```yaml
- connected: true
  id: "OSHW000032254e4500128002ab98002784d1000097969900"
  platform: reel_board
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: COM1
  baud: 9600
- connected: true
  id: "000683759358"
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: nrfjprog
  serial: COM2
  baud: 9600
```

The baud entry is only needed if not running at 115200.

If the map file already exists, then new entries are added and existing entries will be updated. This way you can use one single master hardware map and update it for every run to get the correct serial devices and status of the devices.

With the hardware map ready, you can run any tests by pointing to the map

Linux

```
west twister --device-testing --hardware-map map.yml -T samples/hello_world/
```

Windows

```
west twister --device-testing --hardware-map map.yml -T samples\hello_world
```

The above command will result in twister building tests for the platforms defined in the hardware map and subsequently flashing and running the tests on those platforms.

> **Note**
>
> Currently only boards with support for pyocd, nrfjprog, jlink, openocd, or dediprog are supported with the hardware map features. Boards that require other runners to flash the Zephyr binary are still work in progress.

Hardware map allows to set `--device-flash-timeout` and `--device-flash-with-test` command line options as `flash-timeout` and `flash-with-test` fields respectively. These hardware

map values override command line options for the particular platform.

Serial PTY support using `--device-serial-pty` can also be used in the hardware map:

```
- connected: true
  id: None
  platform: intel_adsp/cavs25
  product: None
  runner: intel_adsp
  serial_pty: path/to/script.py
  runner_params:
    - --remote-host=remote_host_ip_addr
    - --key=/path/to/key.pem
```

The runner_params field indicates the parameters you want to pass to the west runner. For some boards the west runner needs some extra parameters to work. It is equivalent to following west and twister commands.

Linux

```
west flash --remote-host remote_host_ip_addr --key /path/to/key.pem

west twister -p intel_adsp/cavs25 --device-testing --device-serial-pty script.py
--west-flash="--remote-host=remote_host_ip_addr,--key=/path/to/key.pem"
```

Windows

> **Note**
>
> Not supported on Windows OS

> **Note**
>
> For serial PTY, the "–generate-hardware-map" option cannot scan it out and generate a correct hardware map automatically. You have to edit it manually according to above example. This is because the serial port of the PTY is not fixed and being allocated in the system at runtime.

If west is not available or does not know how to flash your system, a custom flash command can be specified using the `flash-command` flag. The script is called with a `--build-dir` with the path of the current build, as well as a `--board-id` flag to identify the specific device when multiple are available in a hardware map.

Linux

```
west twister -p npcx9m6f_evb --device-testing --device-serial /dev/ttyACM0
--flash-command './custom_flash_script.py,--flag,"complex, argument"'
```

Windows

> **Note**
>
> west twister -p npcx9m6f_evb –device-testing –device-serial COM1 –flash-command 'custom_flash_script.py,–flag,"complex, argument"'

Would result in calling `./custom_flash_script.py --build-dir <build directory> --board-id <board identification> --flag "complex, argument"`.

Fixtures Some tests require additional setup or special wiring specific to the test. Running the tests without this setup or test fixture may fail. A test scenario can specify the fixture it needs which can then be matched with hardware capability of a board and the fixtures it supports via the command line or using the hardware map file.

Fixtures are defined in the hardware map file as a list:

```
- connected: true
  fixtures:
    - gpio_loopback
  id: "0240000026334e450015400f5e0e000b4eb1000097969900"
  platform: frdm_k64f
  product: DAPLink CMSIS-DAP
  runner: pyocd
  serial: /dev/ttyACM9
```

When running `twister` with `--device-testing`, the configured fixture in the hardware map file will be matched to test scenarios requesting the same fixtures and these tests will be executed on the boards that provide this fixture.

To reserve a board for fixture-dependent tests, set `run_with_fixture_only` to `true`. Twister will select that board only for test scenarios that request matching fixtures; it will not select the board for scenarios without fixture requirements.

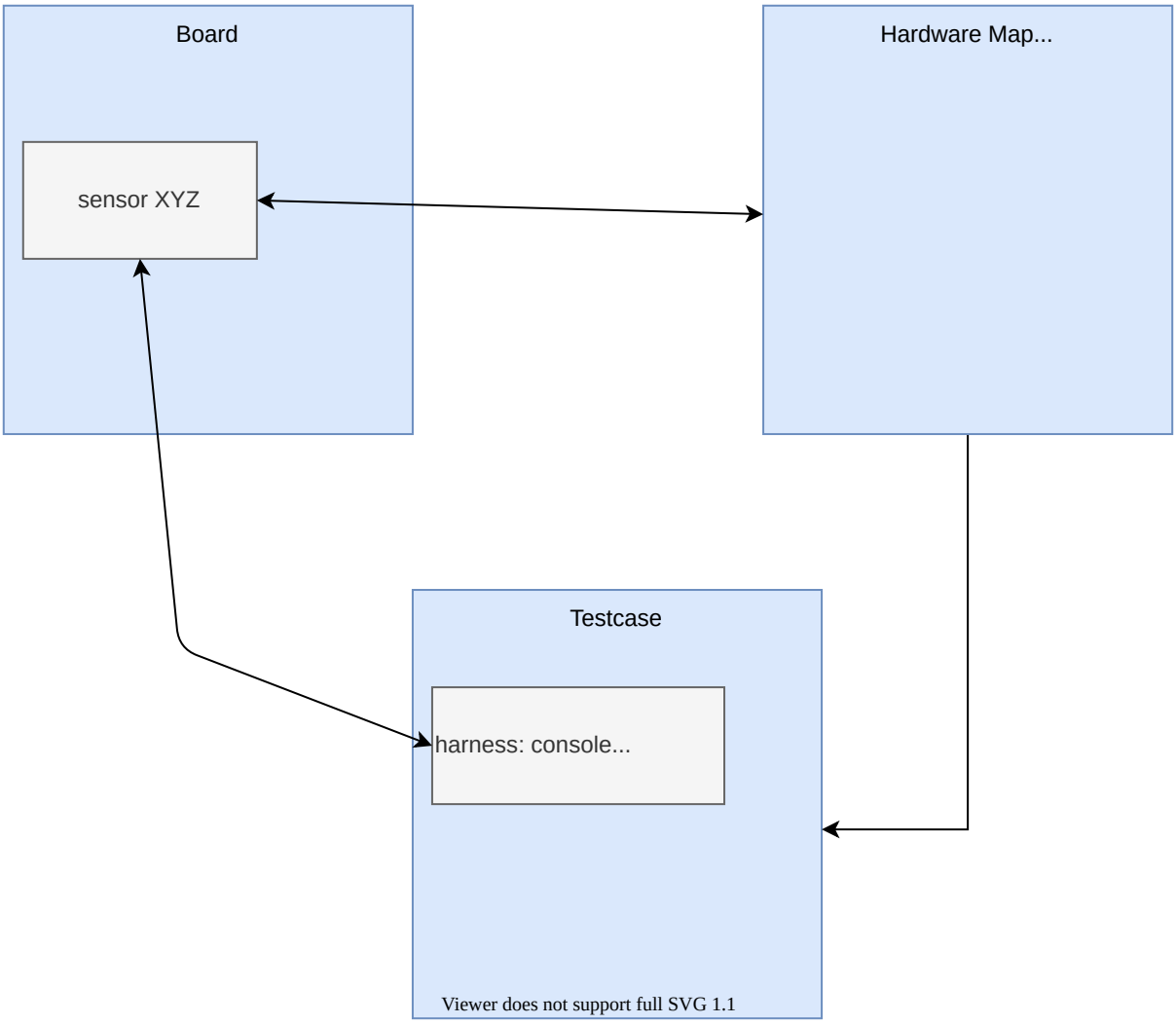
```
- connected: true
  fixtures:
    - gpio_loopback
  run_with_fixture_only: true
  id: 0240000026334e450015400f5e0e000b4eb1000097969900
  platform: frdm_k64f
```

Fixtures can also be provided via twister command option `--fixture`, this option can be used multiple times and all given fixtures will be appended as a list. And the given fixtures will be assigned to all boards, this means that all boards set by current twister command can run those test scenarios which request the same fixtures.

Some fixtures allow for configuration strings to be appended, separated from the fixture name by a `:`. Only the fixture name is matched against the fixtures requested by test scenarios.

Notes It may be useful to annotate board descriptions in the hardware map file with additional information. Use the `notes` keyword to do this. For example:

```
- connected: false
  fixtures:
    - gpio_loopback
  id: "000683290670"
  notes: An nrf5340dk/nrf5340 is detected as an nrf52840dk/nrf52840 with no serial
    port, and three serial ports with an unknown platform.  The board id of the␣
↪serial
    ports is not the same as the board id of the development kit.  If you␣
↪regenerate
    this file you will need to update serial to reference the third port, and␣
↪platform
    to nrf5340dk/nrf5340/cpuapp or another supported board target.
  platform: nrf52840dk/nrf52840
  product: J-Link
  runner: jlink
  serial: null
```


Board
sensor XYZ
Hardware Map...
Testcase
harness: console...
Viewer does not support full SVG 1.1

Overriding Board Identifier When (re-)generated the hardware map file will contain an `id` keyword that serves as the argument to `--board-id` when flashing. In some cases the detected ID is not the correct one to use, for example when using an external J-Link probe. The `probe_id` keyword overrides the `id` keyword for this purpose. For example:

```
- connected: false
  id: "0229000005d9ebc600000000000000000000000097969905"
  platform: mimxrt1060_evk
  probe_id: "000609301751"
  product: DAPLink CMSIS-DAP
  runner: jlink
  serial: null
```

Using Single Board For Multiple Variants

The `platform` attribute can be a list of names or a string with names separated by spaces. This allows to run tests for different platform variants on the same physical board, without re-configuring the hardware map file for each variant. For example:

```
- connected: true
  id: "001234567890"
  platform:
  - nrf5340dk/nrf5340/cpuapp
  - nrf5340dk/nrf5340/cpuapp/ns
  product: J-Link
  runner: nrfjprog
  serial: /dev/ttyACM1
```

Multi-Core testing support Twister supports testing multi-core applications where different cores use separate UART interfaces. This feature works only with the pytest harness (`harness: pytest`). Generated hardware map should contain multiple entries for the same physical device, each representing a different core connection. For example:

```
- connected: true
  id: "001234567890"
  serial: /dev/ttyACM0
- connected: true
  id: "001234567890"
  platform:
  - nrf54l15dk/nrf54l15/cpuapp
  product: J-Link
  runner: nrfutil
  serial: /dev/ttyACM1
```

Both instances share the same device ID but have different serial ports, allowing tests to interact with multiple cores simultaneously. Each connection is handled independently with separate log files.

Multi-DUTs testing support Twister supports test scenarios that require more than one device. This feature works only with the pytest harness (`harness: pytest`). Hardware and `native_sim` execution environments are supported.

To declare that a test needs an additional device, add `required_devices` under `harness_config` in the test's YAML file. Each entry in the list describes one extra DUT. An empty entry `{}` reserves a second device with the same platform and application as the main DUT. See *required_devices* for all available options.

Example test configuration:

```yaml
tests:
  multidut.basic:
    harness: pytest
    harness_config:
      required_devices:
        - {}
```

The hardware map must contain at least one entry per required device. Each entry needs a matching platform and a serial connection:

```yaml
- connected: true
  id: "01"
  platform: nrf52840dk/nrf52840
  serial: /dev/ttyACM0
- connected: true
  id: "02"
  platform: nrf52840dk/nrf52840
  serial: /dev/ttyACM1
```

Run the test on hardware with:

```
$ west twister -vv -ll debug -T tests/subsys/testsuite/multidut \
  --device-testing --hardware-map map.yaml
```

Run the test on `native_sim` (no hardware map required):

```
$ west twister -vv -ll debug -T tests/subsys/testsuite/multidut -p native_sim
```

Twister reserves all required devices (or creates placeholder entries if `native_sim` is used), then passes them to pytest with all necessary information (platform, serial connection, build artifacts to flash, etc.) so the test can interact with all devices.

An example multi-DUT test can be found at tests/subsys/testsuite/multidut.

Quarantine Twister allows user to provide configuration files defining a list of tests or platforms to be put under quarantine. Such tests will be skipped and marked accordingly in the output reports. This feature is especially useful when running larger test suits, where a failure of one test can affect the execution of other tests (e.g. putting the physical board in a corrupted state).

To use the quarantine feature one has to add the argument `--quarantine-list <PATH_TO_QUARANTINE_YAML>` to a twister call. Multiple quarantine files can be used. The current status of tests on the quarantine list can also be verified by adding `--quarantine-verify` to the above argument. This will make twister skip all tests which are not on the given list.

A quarantine yaml is a sequence of dictionaries. Each dictionary must have at least one of the following keys: `scenarios`, `platforms`, `architectures` or `simulations`. A combination of these entries is allowed. An optional `comment` entry can be used to provide more details (e.g., a link to a reported issue). These comments will also be added to the output reports.

When quarantining a class of tests or many scenarios in a single testsuite or when dealing with multiple issues within a subsystem, it is possible to use regular expressions, for example, **kernel.*** would quarantine all kernel tests.

An example of entries in a quarantine yaml:

```yaml
- scenarios:
    - sample.basic.helloworld
  comment: "Link to the issue:␣
↪https://github.com/zephyrproject-rtos/zephyr/pull/33287"
```

(continues on next page)

(continued from previous page)

```
- scenarios:
    - kernel.common
    - kernel.common.(misra|tls)
    - kernel.common.nano64
  platforms:
    - .*_cortex_.*
    - native_sim

- platforms:
    - qemu_x86
  comment: "filter out qemu_x86"

- architectures:
    - riscv

- simulations:
    - armfvp
```

2.12.15 Test Output and Reports

By default, Twister writes all of its output to a `twister-out` directory created in the current working directory. Use `-O`/`--outdir` to choose a different location. On each run this directory is cleaned, unless `--no-clean` is given; `--clobber-output` controls what cleaning does.

Top-level reports

The following files are written at the root of the output directory:

`twister.json`
: The primary machine-readable report. It contains an entry for every selected test suite and test case with its *status*, target platform, execution time, memory footprint, any *recorded data*, and the environment and options used for the run.

`testplan.json`
: The resolved test plan: every test instance (test scenario times platform) that Twister considered, including those that were filtered out together with the reason. This is the file to inspect to understand why a given scenario did or did not run. It can be reused with `--load-tests` to replay the same selection.

`twister.xml`
: JUnit XML summary suitable for CI systems.

`twister_report.xml`
: JUnit XML report including all test cases (not just the summary).

`twister_suite_report.xml`
: JUnit XML report grouped by test suite.

`twister.log`
: Human-readable log of the whole run.

`twister_footprint.json`
: ROM/RAM footprint report. Only generated when `--footprint-report` is used.

The report base name (`twister`) can be changed with `--report-name`, and `--report-suffix` appends a suffix (for example a version or commit ID) to all generated file names. Use `-o`/`--report-dir` to write the reports to a directory other than the output directory, and `--platform-reports` to additionally emit a per-platform `<platform>.json` and `<platform>.xml`. `--report-summary` prints a summary of the failures from the latest run without rebuilding.

Per-test artifacts

Each test instance has its own build directory under the output directory, named after the platform and the test: `twister-out/<platform>/<test path>/<scenario>/`. In addition to the normal Zephyr build artifacts (for example `zephyr/zephyr.elf`), it may contain:

`build.log`
: Output of the build for this instance.

`handler.log`
: Console output captured from the device or emulator while running the test.

`twister_harness.log`
: Log produced by pytest-based harnesses (for example `pytest` and `shell`).

`recording.csv`
: Data fields captured by the `record` option of the *console harness*, when configured.

2.12.16 Live Run Monitoring

For long runs it can be hard to tell from the scrolling console output what twister is actually doing: what is queued, what each job is building or running right now, and which tests have already failed and why. The `--console-monitor` option replaces the normal output with a live full-screen dashboard in the terminal for the duration of the run:

```
$ west twister -T tests/kernel --console-monitor
```

The dashboard shows overall progress with pass/fail/error/filtered breakdowns and an estimated time to completion, the set of test instances currently *in flight* with the pipeline stage each one is in (`cmake`, `build`, `run`, ...), and a scrollable table of every test instance in the plan, including statically filtered ones. Normal log output goes to `twister.log` in the meantime.

Navigation: `Tab` cycles the table filter (all/active/failures/passed/queued/filtered), `f` jumps straight to the failures view, and `/` starts an incremental text search over instance names and failure reasons (`Esc` clears it). Move the selection with the arrow keys or `j`/`k` and press `Enter` to open the detail view for an instance: its pipeline stage timeline, the list of failing test cases with their reasons, and the tail of its log files (`l` switches between the available logs, `j`/`k` or the arrow keys scroll, `g`/`G` jump to the top/end, and the view follows new output while pinned to the end) – particularly useful for inspecting failures while the rest of the run continues.

When the run finishes the dashboard stays up so failures can be inspected; pressing `q` leaves it, after which reports are written and twister exits as usual. Pressing `q` while the run is still going leaves the dashboard early and resumes the normal console output. The option requires an interactive terminal and is ignored otherwise (e.g. in CI).

The monitor observes the run without influencing it: monitoring events are delivered on a best-effort basis and are dropped rather than ever delaying the build/run pipeline.

2.12.17 Twister Configuration File

The Twister configuration file (`test_config.yaml`, passed with `--test-config`) can be used to customize various aspects of twister and the default enabled options and features. This allows tweaking the filtering capabilities depending on the environment and makes it possible to adapt and improve coverage when targeting different sets of platforms.

> **Note**
>
> This file (selected with `--test-config`) configures a whole Twister run. It is distinct from the per-application test configuration in `tests.yaml`, which describes individual *test scenarios*.

The Twister configuration file also adds support for test levels and the ability to assign a specific test to one or more levels. Using command line options of twister it is then possible to select a level and just execute the tests included in this level.

Additionally, the configuration file allows defining level dependencies and additional inclusion of tests into a specific level if the test itself does not have this information already.

In the configuration file you can include complete components using regular expressions and you can specify which test level to import from the same file, making management of levels easier.

To help with testing outside of upstream CI infrastructure, additional options are available in the configuration file, which can be hosted locally. As of now, those options are available:

- Ability to ignore default platforms as defined in board definitions (Those are mostly emulation platforms used to run tests in upstream CI)
- Option to specify your own list of default platforms overriding what upstream defines.
- Ability to override `build_on_all` options used in some test scenarios. This will treat tests or sample as any other just build for default platforms you specify in the configuration file or on the command line.
- Ignore some logic in twister to expand platform coverage in cases where default platforms are not in scope.

Platform Configuration

The following options control platform filtering in twister:

- `override_default_platforms`: override default key a platform sets in board configuration and instead use the list of platforms provided in the configuration file as the list of default platforms. This option is set to False by default.
- `increased_platform_scope`: This option is set to True by default, when disabled, twister will not increase platform coverage automatically and will only build and run tests on the specified platforms.
- `default_platforms`: A list of additional default platforms to add. This list can either be used to replace the existing default platforms or can extend it depending on the value of `override_default_platforms`.
- `build_toolchains`: A mapping of platform names to the list of toolchains every test assigned to that platform should be built with. Twister creates one test instance per toolchain, each in its own build directory. This sets, or overrides, the `build_toolchains` option of the board definition; an empty list disables multi-toolchain builds for a platform that requests them. Since this multiplies build time, it is typically enabled only in the configuration file used by CI (`tests/test_config_ci.yaml`) so that local runs keep building each test once. See *Toolchain Selection*.

And example platforms configuration:

```yaml
platforms:
  override_default_platforms: true
  increased_platform_scope: false
  default_platforms:
    - qemu_x86
  build_toolchains:
    native_sim:
      - host/gnu
      - host/llvm
```

Test Level Configuration

The test configuration allows defining test levels, level dependencies and additional inclusion of tests into a specific test level if the test itself does not have this information already.

In the configuration file you can include complete components using regular expressions and you can specify which test level to import from the same file, making management of levels simple.

And example test level configuration:

```
levels:
  - name: my-test-level
    description: >
      my custom test level
    adds:
      - kernel.threads.*
      - kernel.timer.behavior
      - arch.interrupt
      - boards.*
```

Combined configuration

To mix the Platform and level configuration, you can take an example as below:

An example platforms plus level configuration:

```
platforms:
  override_default_platforms: true
  default_platforms:
    - frdm_k64f
levels:
  - name: smoke
    description: >
        A plan to be used verifying basic zephyr features.
  - name: unit
    description: >
        A plan to be used verifying unit test.
  - name: integration
    description: >
        A plan to be used verifying integration.
  - name: acceptance
    description: >
        A plan to be used verifying acceptance.
  - name: system
    description: >
        A plan to be used verifying system.
  - name: regression
    description: >
        A plan to be used verifying regression.
```

To run with above test_config.yaml file, only default_platforms with given test level test scenarios will run.

```
$ west twister --test-config=<path to>/test_config.yaml -T tests --level="smoke"
```

2.13 Testing

2.13.1 Test Framework

The Zephyr Test Framework (Ztest) provides a simple testing framework intended to be used during development. It provides basic assertion macros and a generic test structure.

The framework can be used in two ways, either as a generic framework for integration testing, or for unit testing specific modules.

- *Quick start - Integration testing*
- *Quick start - Unit testing*
- *Creating a test suite*
- *Adding tests to a suite*
- *Advanced features*
- *Best practices for declaring the test suite*
- *Stress test framework*
- *API reference*
- *Customizing Test Output*
- *Shuffling Test Sequence*
- *Repeating Tests*
- *Test Selection*
- *FFF Extensions*

Quick start - Integration testing

A simple working base is located at samples/subsys/testsuite/integration. To make a test application for the **bar** component of **foo**, you should copy the sample folder to `tests/foo/bar` and edit files there adjusting for your test application's purposes.

To build and execute all applicable test scenarios defined in your test application use the *Twister* tool, for example:

```
west twister -T tests/foo/bar/
```

To select just one of the test scenarios, run Twister with `--scenario` command:

```
west twister --scenario tests/foo/bar/your.test.scenario.name
```

In the command line above `tests/foo/bar` is the path to your test application and `your.test.scenario.name` references a test scenario defined in `tests.yaml` file, which is like `sample.testing.ztest` in the boilerplate test suite sample.

See *Twister test project diagram* for more details on how Twister deals with Ztest application.

The sample contains the following files:

Listing 2: CMakeLists.txt

```
# SPDX-License-Identifier: Apache-2.0

cmake_minimum_required(VERSION 3.28.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(integration)
```

(continues on next page)

(continued from previous page)

```cmake

FILE(GLOB app_sources src/*.c)
target_sources(app PRIVATE ${app_sources})
```

Listing 3: tests.yaml

```yaml
tests:
  # section.subsection
  sample.testing.ztest:
    build_only: true
    platform_allow:
      - native_sim
    integration_platforms:
      - native_sim
    tags: test_framework
```

Listing 4: prj.conf

```
CONFIG_ZTEST=y
```

Listing 5: src/main.c

```c
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/ztest.h>


ZTEST_SUITE(framework_tests, NULL, NULL, NULL, NULL, NULL);

/**
 * @brief Test Asserts
 *
 * This test verifies various assert macros provided by ztest.
 *
 */
ZTEST(framework_tests, test_assert)
{
        zassert_true(1, "1 was false");
        zassert_false(0, "0 was true");
        zassert_is_null(NULL, "NULL was not NULL");
        zassert_not_null("foo", "\"foo\" was NULL");
        zassert_equal(1, 1, "1 was not equal to 1");
        zassert_equal_ptr(NULL, NULL, "NULL was not equal to NULL");
}
```

A test application may consist of multiple test suites that either can be testing functionality or APIs. Functions implementing a test case should follow the guidelines below:

- Test cases function names should be prefixed with **test_**
- Test cases should be documented using doxygen
- Test case function names should be unique within the section or component being tested

For example:

```
/**
 * @brief Test Asserts
 *
 * This test case verifies the zassert_true macro.
 */
ZTEST(my_suite, test_assert)
{
        zassert_true(1, "1 was false");
}
```

Listing Tests Tests (test applications) in the Zephyr tree consist of many test scenarios that run as part of a project and test similar functionality, for example an API or a feature. The `twister` script can parse the test scenarios, suites and cases in all test applications or a subset of them, and can generate reports on a granular level, i.e. if test cases have passed or failed or if they were blocked or skipped.

Twister parses the source files looking for test case names, so you can list all kernel test cases, for example, by running:

```
west twister --list-tests -T tests/kernel
```

Skipping Tests Special- or architecture-specific tests cannot run on all platforms and architectures, however we still want to count those and report them as being skipped. Because the test inventory and the list of tests is extracted from the code, adding conditionals inside the test suite is sub-optimal. Tests that need to be skipped for a certain platform or feature need to explicitly report a skip using `ztest_test_skip()` or `Z_TEST_SKIP_IFDEF`. If the test runs, it needs to report either a pass or fail. For example:

```
#ifdef CONFIG_TEST1
ZTEST(common, test_test1)
{
     zassert_true(1, "true");
}
#else
ZTEST(common, test_test1)
{
     ztest_test_skip();
}
#endif

ZTEST(common, test_test2)
{
     Z_TEST_SKIP_IFDEF(CONFIG_BUGxxxxx);
     zassert_equal(1, 0, NULL);
}

ZTEST_SUITE(common, NULL, NULL, NULL, NULL, NULL);
```

Quick start - Unit testing

Ztest can be used for unit testing. This means that rather than including the entire Zephyr OS for testing a single function, you can focus the testing efforts into the specific module in question. This will speed up testing since only the module will have to be compiled in, and the tested functions will be called directly.

To setup unit tests you have to add a CMakeLists.txt, a tests.yaml and a prj.conf to the directory containing the unit test source files. The resulting binary from this

directory is built using -DBOARD=unit_testing. When twister is invoked the script zephyr/scripts/pylib/twister/twisterlib/testplan.py filters out all tests.yaml in which type: unit is not set. Only unit tests are executed with a firmware build with BOARD=unit_testing.

> **Note**
>
> Unit tests are run as **native** applications on the host and are therefore subject to similar limitations as documented for the POSIX architecture. Running unit tests is therefore only supported for Linux. To run unit tests on Windows or macOS it is necessary to use containers or Virtual Machines running a Linux guest. Follow the same instructions as for the POSIX Arch dependencies.

The `unit_testing` board Unit tests are built for the special `unit_testing` board (subsys/testsuite/boards/unit_testing). It is not a real piece of hardware nor a simulated target: it is a pseudo-board with `arch: unit` that uses the host toolchain to produce a regular native executable. Selecting it with `-DBOARD=unit_testing` (which Twister does automatically for scenarios marked `type: unit`) builds and links only the source files you add to the `testbinary` target together with the Ztest unit-test harness.

Crucially, **the Zephyr kernel and operating system are not built at all**. There is no boot sequence, no scheduler, no devicetree-driven device initialization, and no driver model. The functions under test are compiled into the test binary and called directly. Any kernel API or other dependency that the module under test relies on must be supplied by the test itself, usually as a stub or a *mock*.

Difference from `native_sim` and other boards It is easy to confuse the `unit_testing` board with native_sim, since both run on the host. They are fundamentally different:

- native_sim builds the **complete Zephyr OS** – kernel, devicetree, Kconfig, drivers and subsystems – into a host binary that boots and runs exactly like a Zephyr image on real hardware, only compiled for the host instead of a target SoC. Use it to run full applications and integration tests on the host. Tests for these boards do **not** set `type: unit`.
- `unit_testing` builds **none** of that. It links only the code under test plus Ztest, with everything else stubbed or mocked, and calls the tested functions directly. This makes builds and runs fast and keeps the focus on a single module, at the cost of having to provide stubs for every dependency. These tests must set `type: unit` (see *below*).

In short, reach for `native_sim` to exercise code in the context of a running Zephyr system, and for `unit_testing` to test an isolated module without pulling in the kernel.

CMakeLists.txt In order to declare the unit tests present in a source folder, you need to add the relevant source files to the `testbinary` target from the CMake unittest component. See a minimal example below:

```
cmake_minimum_required(VERSION 3.28.0)

project(app)
find_package(Zephyr COMPONENTS unittest REQUIRED HINTS $ENV{ZEPHYR_BASE})
target_sources(testbinary PRIVATE main.c)
```

Since you won't be including basic kernel data structures that most code depends on, you have to provide function stubs in the test. Ztest provides some helpers for mocking functions, as demonstrated below.

In a unit test, mock objects can simulate the behavior of complex real objects and are used to decide whether a test failed or passed by verifying whether an interaction with an object occurred, and if required, to assert the order of that interaction.

tests.yaml You have to set the value for the key "type" to "unit" in the tests.yaml

```
tests:
   testscenario.testsuite:
      tags: your_tag
      type: unit
```

prj.conf For unit tests this contains usually only

```
CONFIG_ZTEST=y
```

If your unit tests require additional libraries (e.g. math-lib) you will have to add them either via the CMakeLists.txt or in the tests.yaml:

```
tests:
   testscenario.testsuite:
      tags: your_tag
      type: unit
      extra_args:
         - EXTRA_LDFLAGS="-lm"
```

Examples of unit tests can be found in the tests/unit/ folder.

Creating a test suite

Using Ztest to create a test suite is as easy as calling the `ZTEST_SUITE`. The macro accepts the following arguments:

- `suite_name` - The name of the suite. This name must be unique within a single binary.
- `ztest_suite_predicate_t` - An optional predicate function to allow choosing when the test will run. The predicate will get a pointer to the global state passed in through `ztest_run_all()` and should return a boolean to decide if the suite should run.
- `ztest_suite_setup_t` - An optional setup function which returns a test fixture. This will be called and run once per test suite run.
- `ztest_suite_before_t` - An optional before function which will run before every single test in this suite.
- `ztest_suite_after_t` - An optional after function which will run after every single test in this suite.
- `ztest_suite_teardown_t` - An optional teardown function which will run at the end of all the tests in the suite.

Below is an example of a test suite using a predicate:

```
#include <zephyr/ztest.h>
#include "test_state.h"

static bool predicate(const void *global_state)
{
     return ((const struct test_state*)global_state)->x == 5;
}

ZTEST_SUITE(alternating_suite, predicate, NULL, NULL, NULL, NULL);
```

Adding tests to a suite

There are 5 macros used to add a test to a suite, they are:

- ZTEST (suite_name, test_name) - Which can be used to add a test by test_name to a given suite by suite_name.
- ZTEST_P (suite_name, test_name) - Add a value-parameterized test to a given suite. The test body is executed once per registered parameter value. Inside the body, call ztest_get_current_param() or use the ZTEST_GET_PARAM typed helper to retrieve the current value. The suite fixture (data argument) is independent from the parameter and is never overwritten. See *Value-parameterized tests* for the full API.
- ZTEST_USER (suite_name, test_name) - Which behaves the same as ZTEST, only that when CONFIG_USERSPACE is enabled, then the test will be run in a userspace thread.
- ZTEST_F (suite_name, test_name) - Which behaves the same as ZTEST, only that the test function will already include a variable named fixture with the type <suite_name>_fixture.
- ZTEST_USER_F (suite_name, test_name) - Which combines the fixture feature of ZTEST_F with the userspace threading for the test.

Test fixtures Test fixtures can be used to help simplify repeated test setup operations. In many cases, tests in the same suite will require some initial setup followed by some form of reset between each test. This is achieved via fixtures in the following way:

```
#include <zephyr/ztest.h>

struct my_suite_fixture {
     size_t max_size;
     size_t size;
     uint8_t buff[1];
};

static void *my_suite_setup(void)
{
     /* Allocate the fixture with 256 byte buffer */
     struct my_suite_fixture *fixture = malloc(sizeof(struct my_suite_fixture) +␣
↪255);

     zassume_not_null(fixture, NULL);
     fixture->max_size = 256;

     return fixture;
}

static void my_suite_before(void *f)
{
     struct my_suite_fixture *fixture = (struct my_suite_fixture *)f;
     memset(fixture->buff, 0, fixture->max_size);
     fixture->size = 0;
}

static void my_suite_teardown(void *f)
{
     free(f);
}

ZTEST_SUITE(my_suite, NULL, my_suite_setup, my_suite_before, NULL,␣
↪my_suite_teardown);

ZTEST_F(my_suite, test_feature_x)
```

(continues on next page)

(continued from previous page)

```
{
     zassert_equal(0, fixture->size);
     zassert_equal(256, fixture->max_size);
}
```

Using memory allocated by a test fixture in a userspace thread, such as during execution of ZTEST_USER or ZTEST_USER_F, requires that memory to be declared userspace accessible. This is because the fixture memory is owned and initialized by kernel space. The Ztest framework provides the ZTEST_DMEM and ZTEST_BMEM macros for use of such user/kernel space shared memory.

Advanced features

Value-parameterized tests Value-parameterized tests allow a single test body to be executed once for each value in a supplied list, similar to GoogleTest's TEST_P / INSTANTIATE_TEST_SUITE_P pattern. The fixture and the parameter are completely independent: the suite's setup() return value is always passed as data and is never overwritten by a parameter value.

Declaring a parameterized test body Use ZTEST_P in the same way as ZTEST. Inside the body, the data pointer carries the suite fixture (identical to ZTEST_F). The current parameter value is retrieved through the run-time accessors:

```
#include <zephyr/ztest.h>

struct my_suite_fixture {
     int initial_value;
};

static void *my_suite_setup(void) {
     static struct my_suite_fixture f = { .initial_value = 42 };
     return &f;
}

ZTEST_SUITE(my_suite, NULL, my_suite_setup, NULL, NULL, NULL);

ZTEST_P(my_suite, test_multiply)
{
     struct my_suite_fixture *f = (struct my_suite_fixture *)data;
     int factor = ZTEST_GET_PARAM(int);

     /* fixture is always intact, regardless of parameter */
     zassert_equal(f->initial_value, 42, "fixture corrupted");
     zassert_true(f->initial_value * factor > 0, "product must be positive");
}
```

Declaring parameter values Use ZTEST_DEFINE_PARAM_VALUES to create a static value set from literal values:

```
ZTEST_DEFINE_PARAM_VALUES(small_factors, int, 1, 2, 3);
```

For values already stored in an array use ZTEST_DEFINE_PARAM_VALUES_ARRAY:

```
static const int big_factors[] = { 10, 100, 1000 };
ZTEST_DEFINE_PARAM_VALUES_ARRAY(big_factor_vals, big_factors);
```

For a numeric range use ZTEST_DEFINE_PARAM_RANGE, which mirrors GoogleTest's testing::Range(begin, end [, step]) semantics. Values are {begin, begin+step, ...

} up to but **not** including end. No backing array is allocated, so large ranges have zero RAM overhead:

```
/* {0, 2, 4, 6, 8} - 5 values, step is supplied explicitly */
ZTEST_DEFINE_PARAM_RANGE(even_vals, int, 0, 10, 2);

/* {1, 2, 3, 4, 5} - step=1 is the common case */
ZTEST_DEFINE_PARAM_RANGE(one_to_five, int, 1, 6, 1);
```

> **Note**
>
> `ZTEST_DEFINE_PARAM_RANGE` requires `end > begin` and `step > 0`, both enforced at compile time via `BUILD_ASSERT`.

For values that must be **computed at runtime** — for instance random numbers, hardware sensor readings, or values produced by a custom algorithm — use `ZTEST_DEFINE_PARAM_GENERATOR` or `ZTEST_DEFINE_PARAM_GENERATOR_WITH_SETUP`. Both accept a user-provided generator callback with the signature `void gen(size_t index, void *out)` that writes one value per invocation. Like ranges, no backing array is allocated.

The `_WITH_SETUP` variant additionally calls a `void setup(void)` hook **once** before the dispatch loop. This is the right place to seed a PRNG, reset a stateful counter, or open any resource needed by the generator:

```
#include <zephyr/random/random.h>

/* Seed the RNG before the first iteration so failures are reproducible. */
static void seed_rng(void)
{
	sys_rand_seed(MY_FUZZ_SEED);
}

static void rand_u32_gen(size_t idx, void *out)
{
	ARG_UNUSED(idx);
	*(uint32_t *)out = sys_rand32_get();
}

ZTEST_DEFINE_PARAM_GENERATOR_WITH_SETUP(fuzz_vals, uint32_t, MY_FUZZ_ITERATIONS,
                                        seed_rng, rand_u32_gen);
```

When no setup is needed, use the simpler form:

```
static void deterministic_gen(size_t idx, void *out)
{
	/* Deterministic but computed at runtime (e.g. based on hardware ID). */
	*(uint32_t *)out = get_device_seed() ^ (uint32_t)idx;
}

ZTEST_DEFINE_PARAM_GENERATOR(hw_vals, uint32_t, 16U, deterministic_gen);
```

> **Note**
>
> The `count_` argument to both generator macros must be a constant expression (a numeric literal, a `#define`, or a Kconfig symbol such as `MY_FUZZ_ITERATIONS`). Truly dynamic counts are not supported.

Struct-typed parameters work the same way:

```
struct point { int x; int y; };

static const struct point corners[] = { {0,0}, {1,0}, {0,1}, {1,1} };
ZTEST_DEFINE_PARAM_VALUES_ARRAY(corner_vals, corners);

ZTEST_P(my_suite, test_in_unit_square)
{
     const struct point *p = ZTEST_GET_PARAM_PTR(struct point);
     zassert_true(p->x >= 0 && p->x <= 1 && p->y >= 0 && p->y <= 1,
                  "point (%d, %d) outside unit square", p->x, p->y);
}
```

Instantiating a parameterized test `ZTEST_INSTANTIATE_TEST_SUITE_P` binds a value set to a test body. Each call creates a separate named instantiation; the same test body may be instantiated multiple times with different value sets:

```
ZTEST_INSTANTIATE_TEST_SUITE_P(small, my_suite, test_multiply, small_factors);
ZTEST_INSTANTIATE_TEST_SUITE_P(big,   my_suite, test_multiply, big_factor_vals);
```

The first argument (`small` / `big`) is an arbitrary unique identifier within the compilation unit; it is recorded in the test metadata but does not affect test naming as reported by Twister.

Retrieving the current parameter Inside a `ZTEST_P` body the following helpers are available:

Helper	Description
`ztest_has_current_param()`	Returns `true` when called inside a parameterized invocation.
`ztest_get_current_param()`	Returns a `const void *` pointer to the current value.
`ZTEST_GET_PARAM_PTR(type)`	Returns a `const type *` pointer to the current value.
`ZTEST_GET_PARAM(type)`	Dereferences and returns the current value as `type`.
`ztest_get_current_param_index()`	Returns the zero-based index of the current value within its set.
`ztest_get_current_param_size()`	Returns the size in bytes of one parameter element.

Non-parameterized tests (`ZTEST`, `ZTEST_F`) always see `ztest_has_current_param()` return `false` and `ztest_get_current_param()` return `NULL`.

Test result expectations Some tests were made to be broken. In cases where the test is expected to fail or skip due to the nature of the code, it's possible to annotate the test as such. For example:

```
#include <zephyr/ztest.h>

ZTEST_SUITE(my_suite, NULL, NULL, NULL, NULL, NULL);

ZTEST_EXPECT_FAIL(my_suite, test_fail);
ZTEST(my_suite, test_fail)
{
  /** This will fail the test */
  zassert_true(false, NULL);
}

ZTEST_EXPECT_SKIP(my_suite, test_skip);
```

(continues on next page)

(continued from previous page)

```
ZTEST(my_suite, test_skip)
{
  /** This will skip the test */
  zassume_true(false, NULL);
}
```

In this example, the above tests should be marked as failed and skipped respectively. Instead, Ztest will mark both as passed due to the expectation.

Test rules Test rules are a way to run the same logic for every test and every suite. There are a lot of cases where you might want to reset some state for every test in the binary (regardless of which suite is currently running). As an example, this could be to reset mocks, reset emulators, flush the UART, etc.:

```
#include <zephyr/fff.h>
#include <zephyr/ztest.h>

#include "test_mocks.h"

DEFINE_FFF_GLOBALS;

DEFINE_FAKE_VOID_FUN(my_weak_func);

static void fff_reset_rule_before(const struct ztest_unit_test *test, void␣
↪*fixture)
{
     ARG_UNUSED(test);
     ARG_UNUSED(fixture);

     RESET_FAKE(my_weak_func);
}

ZTEST_RULE(fff_reset_rule, fff_reset_rule_before, NULL);
```

A custom `test_main` While the Ztest framework provides a default `test_main()` function, it's possible that some applications will want to provide custom behavior. This is particularly true if there's some global state that the tests depend on and that state either cannot be replicated or is difficult to replicate without starting the process over. For example, one such state could be a power sequence. Assuming there's a board with several steps in the power-on sequence a test suite can be written using the `predicate` to control when it would run. In that case, the `test_main()` function can be written as follows:

```
#include <zephyr/ztest.h>

#include "my_test.h"

void test_main(void)
{
     struct power_sequence_state state;

     /* Only suites that use a predicate checking for phase == PWR_PHASE_0 will␣
↪run. */
     state.phase = PWR_PHASE_0;
     ztest_run_all(&state, false, 1, 1);
```

(continues on next page)

(continued from previous page)

```
    /* Only suites that use a predicate checking for phase == PWR_PHASE_1 will␣
↪run. */
    state.phase = PWR_PHASE_1;
    ztest_run_all(&state, false, 1, 1);

    /* Only suites that use a predicate checking for phase == PWR_PHASE_2 will␣
↪run. */
    state.phase = PWR_PHASE_2;
    ztest_run_all(&state, false, 1, 1);

    /* Check that all the suites in this binary ran at least once. */
    ztest_verify_all_test_suites_ran();
}
```

The signature of `ztest_run_all()` is `ztest_run_all(const void *state, bool shuffle, int suite_iter, int case_iter)`:

- `state` - Pointer to the global state passed to each suite's `predicate`.
- `shuffle` - When `true`, randomize the order in which suites and tests run (requires `CONFIG_ZTEST_SHUFFLE`); `false` keeps the default alphanumerical order.
- `suite_iter` - Number of times to repeat each test suite.
- `case_iter` - Number of times to repeat each test case.

In the example above each call runs the matching suites once, in order, without shuffling.

Best practices for declaring the test suite

twister and other validation tools need to obtain the list of test cases that a Zephyr *ztest* test image will expose.

Rationale

This all is for the purpose of traceability. It's not enough to have only a semaphore test application. We also need to show that we have testpoints for all APIs and functionality, and we trace back to documentation of the API, and functional requirements.

The idea is that test reports show results for every test case as passed, failed, blocked, or skipped. Reporting on only the high-level test application, particularly when tests do too many things, is too vague.

Other questions:

- Why not pre-scan with CPP and then parse? or post scan the ELF file?

  If C pre-processing or building fails because of any issue, then we won't be able to tell the subcases.

- Why not declare them in the YAML test configuration?

  A separate test case description file would be harder to maintain than just keeping the information in the test source files themselves – only one file to update when changes are made eliminates duplication.

Stress test framework

Zephyr stress test framework (Ztress) provides an environment for executing user functions in multiple priority contexts. It can be used to validate that code is resilient to preemptions. The framework tracks the number of executions and preemptions for each context. Execution can

have various completion conditions like timeout, number of executions or number of preemptions.

The framework is setting up the environment by creating the requested number of threads (each on different priority), optionally starting a timer. For each context, a user function (different for each context) is called and then the context sleeps for a randomized amount of system ticks. The framework is tracking CPU load and adjusts sleeping periods to achieve higher CPU load. In order to increase the probability of preemptions, the system clock frequency should be relatively high. The default 100 Hz on QEMU x86 is much too low and it is recommended to increase it to 100 kHz.

The stress test environment is setup and executed using `ZTRESS_EXECUTE` which accepts a variable number of arguments. Each argument is a context that is specified by `ZTRESS_TIMER` or `ZTRESS_THREAD` macros. Contexts are specified in priority descending order. Each context specifies completion conditions by providing the minimum number of executions and preemptions. When all conditions are met and the execution has completed, an execution report is printed and the macro returns. Note that while the test is executing, a progress report is periodically printed.

Execution can be prematurely completed by specifying a test timeout (`ztress_set_timeout()`) or an explicit abort (`ztress_abort()`).

User function parameters contains an execution counter and a flag indicating if it is the last execution.

The example below presents how to setup and run 3 contexts (one of which is k_timer interrupt handler context). Completion criteria is set to at least 10000 executions of each context and 1000 preemptions of the lowest priority context. Additionally, the timeout is configured to complete after 10 seconds if those conditions are not met. The last argument of each context is the initial sleep time which will be adjusted throughout the test to achieve the highest CPU load.

```
ztress_set_timeout(K_MSEC(10000));
ZTRESS_EXECUTE(ZTRESS_TIMER(foo_0, user_data_0, 10000, Z_TIMEOUT_TICKS(20)),
               ZTRESS_THREAD(foo_1, user_data_1, 10000, 0, Z_TIMEOUT_TICKS(20)),
               ZTRESS_THREAD(foo_2, user_data_2, 10000, 1000,␣
↪Z_TIMEOUT_TICKS(20)));
```

Configuration Static configuration of Ztress contains:

- `CONFIG_ZTRESS_MAX_THREADS` - number of supported threads.
- `CONFIG_ZTRESS_STACK_SIZE` - Stack size of created threads.
- `CONFIG_ZTRESS_REPORT_PROGRESS_MS` - Test progress report interval.

API reference

Running tests

Ztest testing macros

Assertions These macros will instantly fail the test if the related assertion fails. When an assertion fails, it will print the current file, line and function, alongside a reason for the failure and an optional message. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the assertions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from `zassert_equal(buf->ref, 2, "Invalid refcount")`:

```
Assertion failed at main.c:62: test_get_single_buffer: Invalid refcount (buf->ref␣
↪not equal to 2)
Aborted at unit test function
```

Ztest assertion macros

Expectations These macros will continue test execution if the related expectation fails and subsequently fail the test at the end of its execution. When an expectation fails, it will print the current file, line, and function, alongside a reason for the failure and an optional message but continue executing the test. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the expectations will only print the file and line numbers, reducing the binary size of the test.

For example, if the following expectations fail:

```
zexpect_equal(buf->ref, 2, "Invalid refcount");
zexpect_equal(buf->ref, 1337, "Invalid refcount");
```

The output will look something like:

```
START - test_get_single_buffer
    Expectation failed at main.c:62: test_get_single_buffer: Invalid refcount␣
↪(buf->ref not equal to 2)
    Expectation failed at main.c:63: test_get_single_buffer: Invalid refcount␣
↪(buf->ref not equal to 1337)
 FAIL - test_get_single_buffer in 0.0 seconds
```

Ztest expectation macros

Assumptions These macros will instantly skip the test or suite if the related assumption fails. When an assumption fails, it will print the current file, line, and function, alongside a reason for the failure and an optional message. If the config `CONFIG_ZTEST_ASSERT_VERBOSE` is 0, the assumptions will only print the file and line numbers, reducing the binary size of the test.

Example output for a failed macro from `zassume_equal(buf->ref, 2, "Invalid refcount")`:

```
START - test_get_single_buffer
    Assumption failed at main.c:62: test_get_single_buffer: Invalid refcount␣
↪(buf->ref not equal to 2)
 SKIP - test_get_single_buffer in 0.0 seconds
```

Ztest assumption macros

Ztress

Ztest ztress macros

Mocking via FFF Zephyr has integrated with FFF for mocking. See FFF for documentation. To use it, include the relevant header:

```
#include <zephyr/fff.h>
```

Zephyr provides several FFF-based fake drivers which can be used as either stubs or mocks. Fake driver instances are configured via *Devicetree* and *Configuration System (Kconfig)*. See the following devicetree bindings for more information:

- `zephyr,fake-can`
- `zephyr,fake-comp`
- `zephyr,fake-eeprom`
- `zephyr,fake-leds`
- `zephyr,fake-pwm`
- `zephyr,fake-regulator`

- `zephyr,fake-rtc`
- `zephyr,fake-stepper-ctrl`
- `zephyr,fake-stepper-driver`

Zephyr also has defined extensions to FFF for simplified declarations of fake functions. See *FFF Extensions*.

Customizing Test Output

Customization is enabled by setting `CONFIG_ZTEST_TC_UTIL_USER_OVERRIDE` to "y" and adding a file `tc_util_user_override.h` with your overrides.

Add the line `zephyr_include_directories(my_folder)` to your project's `CMakeLists.txt` to let Zephyr find your header file during builds.

See the file subsys/testsuite/include/zephyr/tc_util.h to see which macros and/or defines can be overridden. These will be surrounded by blocks such as:

```
#ifndef SOMETHING
#define SOMETHING <default implementation>
#endif /* SOMETHING */
```

Shuffling Test Sequence

By default the tests are sorted and ran in alphanumerical order. Test cases may be dependent on this sequence. Enable `CONFIG_ZTEST_SHUFFLE` to randomize the order. The output from the test will display the seed for failed tests. For native simulator builds you can provide the seed as an argument to twister with `--seed`.

Repeating Tests

By default the tests are executed once. The test cases and test suites may be executed multiple times. Enable `CONFIG_ZTEST_REPEAT` to execute the tests multiple times. By default the multiplication factors are 3, which means every test suite is executed 3 times and every test case is executed 3 times. This can be changed by the `CONFIG_ZTEST_SUITE_REPEAT_COUNT` and `CONFIG_ZTEST_TEST_REPEAT_COUNT` Kconfig options.

Test Selection

For tests built for native simulator, use command line arguments to list or select tests to run. The test argument expects a comma separated list of `suite::test` . You can substitute the test name with an `*` to run all tests within a suite.

For example

```
$ zephyr.exe -list
$ zephyr.exe -test=
↪"fixture_tests::test_fixture_pointer,framework_tests::test_assert_mem_equal"
$ zephyr.exe -test="framework_tests::*"
```

FFF Extensions

FFF extensions

2.13.2 Benchmarking Framework

The Zephyr benchmarking framework provides cycle-accurate performance measurements. It automates data collection and statistical calculation, offering a standardized way to evaluate execution metrics across the Zephyr ecosystem.

Overview

This framework helps identify regressions and optimize critical paths by providing:

- **Standardized API**: Macros that align with existing `ztest` conventions.
- **Statistical Analysis**: Calculations of Mean, Standard Deviation, Standard Error, and Min/Max values.
- **Overhead Compensation**: Inclusion of a control test to account for the benchmarking frameworks own execution time. The control is subtracted from every reported statistic identically, so the corrected values are real numbers that generally do not coincide with any single measurement.

Configuration

To use the benchmarking framework, you must enable the following Kconfig options:

```
CONFIG_ZTEST=y
CONFIG_ZTEST_BENCHMARK=y
```

Usage

A benchmark suite is defined similarly to the normal ztest testsuite by first defining the suite with `ZTEST_BENCHMARK_SUITE` and then adding individual benchmark tests to the suite using either `ZTEST_BENCHMARK` or `ZTEST_BENCHMARK_TIMED` macros.

```
#include <zephyr/ztest.h>

ZTEST_BENCHMARK_SUITE(<test suite name>, <setup_fn>, <teardown_fn>);
```

Standard Benchmarks Standard benchmarks are sample-based, meaning they execute the test a specified number of times and measure the total cycles taken. This is useful for benchmarking critical paths where you want to understand the raw CPU performance in terms of cycles. It provides insights into the efficiency of the code and helps identify bottlenecks in terms of CPU usage. This benchmarking method is suitable for code that has consistent execution times and is not heavily influenced by external factors such as I/O operations or context switches.

```
#include <zephyr/ztest.h>

ZTEST_BENCHMARK_SUITE(<suite name>, NULL, NULL);

ZTEST_BENCHMARK(<suite name>, <benchmark name>, <number of samples>, <setup_fn>,
↪<teardown_fn>)
{
    /* Code to benchmark */
}
```

A standard benchmark follows a flow where the setup function is called before each sample, the test function is executed for the specified number of samples, and the teardown function is called after each sample.

Timed Benchmarks Timed benchmarks in contrast to the standard benchmarks measures execution time of the code instead of cycles. This is useful for benchmarking code that may have variable execution times or when you want to measure the actual time taken rather than just CPU cycles of a critical path. It provides a broader view of performance characteristics, especially for code that involves I/O operations, context switches, or other factors that can influence execution time beyond raw CPU performance.

```
ZTEST_BENCHMARK_TIMED(<suite name>, <benchmark name>, <time in ms>, <setup_fn>,
↪<teardown_fn>)
{
      /* Code to benchmark */
}
```

Unlike standard benchmarks that favor isolation, Timed Benchmarks executes setup and teardown functions only once allowing the test function to run hot within a dedicated time window. This will give a more realistic measurements as it includes the overhead of the system as it would be in a real-world scenario, such as interrupts, context switches, and other background tasks.

Manual Benchmarks Standard and timed benchmarks are timed by the framework itself, which takes both timestamps in thread context around each invocation of the benchmark body. Some measurements cannot be expressed that way because their endpoints are captured in different execution contexts: for example the latency from raising an interrupt to the first instruction of its ISR, or from the end of an ISR to a woken thread running.

Manual benchmarks keep the loop, the setup and the teardown in the framework and hand the body only the choice of what is measured, by bracketing it with `ztest_benchmark_start()` and `ztest_benchmark_end()`. The framework computes and reports the same statistics as for standard benchmarks.

```
ZTEST_BENCHMARK_MANUAL(<suite name>, <benchmark name>, <samples>, <setup_fn>,
↪<teardown_fn>)
{
      prepare();

      ztest_benchmark_start();
      operation_under_test();
      ztest_benchmark_end();
}
```

Both hooks only take a timestamp, so either may be called from an ISR and the span need not begin and end in the same execution context. For the latency from an interrupt to the thread it wakes, the ISR marks the start and the thread ends the span:

```
static void my_isr(const void *arg)
{
      ztest_benchmark_start();
}

ZTEST_BENCHMARK_MANUAL(<suite name>, isr_exit_latency, 1000, NULL, NULL)
{
      trigger_the_interrupt();

      ztest_benchmark_end();
}
```

The sample itself is computed once the body has returned, which is what keeps the statistics out of interrupt context: updating them uses floating point, which is not allowed in an ISR on every architecture. An iteration whose body marks no span contributes no sample.

Manual benchmarks are corrected against a control of their own rather than the one the sampled benchmarks use. A manual span costs the two timestamps that bracket it and nothing else, where the sampled control also measures the indirect call to the benchmark body, which is not part of a manual span. A span whose ends are captured in two different contexts pays neither exactly, and no control can express that.

The warmup applies here exactly as it does to a sampled benchmark: the framework runs the body `CONFIG_ZTEST_BENCHMARK_WARMUP` extra times and discards those spans, keeping the first as the cold cost. The body never has to distinguish between the two phases, and every measured iteration is preceded by exactly the same work as the one before it.

Understanding Results

Standard Benchmarking Results

```
<suite name> ##############################################
<benchmark name> ==========================================
   Sample size:<number of samples>, total cycles: <total amount of cycles>
   Mean(u): <mean cycles per sample>
   Standard deviation(s): <cycles>
   Standard Error(SE): <cycles>
   Min: <cycles> (run #<sample number>)
   Max: <cycles> (run #<sample number>)
```

Note

`CONFIG_ZTEST_BENCHMARK_OUTLIERS` replaces the standard error line with the number of samples that exceed the median by more than `CONFIG_ZTEST_BENCHMARK_OUTLIER_MARGIN_DIV` allows, and that number as a percentage. The margin matters because the counter resolution spreads the baseline over a few counts; without it about half of any distribution would be reported. A latency distribution is usually a baseline plus a handful of disturbed runs rather than samples of one stochastic population, and a standard error over the two suggests an average that no single execution ever produced – ten thousand runs of 760 cycles and one of 1240 give a standard error of 0.048, which reads as precision and is not, where `1 / 10000 (0.010%)` says what happened. Only the report changes; the standard error keeps its CSV column.

Statistical Metrics

- **Mean (u)**: The average number of cycles taken per sample. It provides a central value representing the expected cost of execution.
- **Standard Deviation (s)**: Measures the amount of variation or fluctuation of the execution cost from the mean. A low standard deviation indicates that the behavior is deterministic and consistent.
- **Standard Error (SE)**: Estimates how far the sample mean is likely to be from the "true" mean of the system. It provides insight into the statistical reliability of the test. A lower SE indicates higher confidence in the result.
- **Min/Max**: The minimum and maximum cycle counts observed, along with which sample they occurred on.

Warmup and the cold cost A benchmark is defined as three phases: the setup, then `CONFIG_ZTEST_BENCHMARK_WARMUP` iterations that are executed but not recorded, then the measured samples. The warmup applies to every benchmark in the image.

The very first execution is always unrecorded, whatever the warmup is set to. It runs with cold caches, branch predictors and TLBs and can be an order of magnitude slower than the steady

state, and rather than discard that information the framework reports it on its own as the **cold cost**. The two answer different questions — what an operation costs the first time it is reached, and what it costs thereafter — and an application that runs a path once at startup cares about the first.

Keeping it out of the distribution is what makes the distribution mean anything at the tail. A benchmark that reported the cold start in both places would have it decide the far percentiles: at ten thousand samples the nearest-rank p99.99 is the second largest value, so one cold start would be read as the steady-state tail.

Raising the warmup beyond that costs accuracy on a micro benchmark whose whole cost is comparable to a cache miss, and with a large enough sample count the early iterations have no measurable effect on the mean, so the default of zero is the right starting point. Raise it when the benchmark is large enough for cache state to matter and the steady state is what you are after.

Latency percentiles The mean and the standard error describe how precisely the average was estimated. That is the right question for throughput, but not for latency: a real-time system is characterised by how bad an individual operation can be, and that lives in the tail of the distribution rather than near the mean.

Enable `CONFIG_ZTEST_BENCHMARK_PERCENTILES` to retain the individual samples and report `min`, `p50`, `p90`, `p99`, `p99.9`, `p99.99` and `max` alongside the usual statistics. In CSV mode each sampled and manual benchmark emits an additional `P` row; see `CONFIG_ZTEST_BENCHMARK_OUTPUT_CSV` for the columns.

The difference this makes is clearest on a distribution with a tail. Measuring interrupt entry latency on a loaded system gives a mean of 1264 cycles with a standard error of 34, which describes no interrupt that actually occurred: the percentiles show 1216 cycles all the way out to p99, and 25184 beyond it.

A percentile has to be resolvable by the number of samples taken. p99 needs at least 100 samples, p99.9 at least 1000 and p99.99 at least 10000; with fewer, the reported value degenerates to the maximum. Samples are retained in a buffer of `CONFIG_ZTEST_BENCHMARK_MAX_SAMPLES` entries of 8 bytes each, which has to be at least as large as the sample count of the largest benchmark. The retained samples are the first ones taken rather than a sample of the whole run, so percentiles over a truncated prefix would miss whatever happened after the buffer filled, which is where the tail tends to live. A benchmark that overruns the buffer therefore reports no percentiles at all, and says how many samples did not fit.

Plainly speaking, lower values are better for all metrics. Lower mean, min, and max values indicates better raw performance. Lower standard deviation and standard error values indicate more consistent and therefore reliable performance.

Timed Benchmarking Results

```
<benchmark name> ===============================================
   Samples: <Number of samples executed during the benchmark>
   Total Time: <Gross execution time>
   Work Time: <Net execution time> ns (Net)
   Ops/Sec: <average operations per second>
   Cycles/Op: <average cycles per operation>
```

Statistical Metrics

- **Total Time**: The total time taken for all samples of the benchmark, including overhead.
- **Work Time**: The total time taken for the code under test, excluding the overhead of the benchmarking framework itself. This provides a more accurate measure of the actual performance of the code being benchmarked.

- **Ops/Sec**: The number of operations (samples) that can be performed per second, calculated by dividing the number of samples by the net work time (in seconds). This metric is particularly useful for understanding the throughput of the code being benchmarked.
- **Cycles/Op**: The average number of CPU cycles taken per operation, calculated by dividing the net number of cycles by the number of samples. This metric provides insight into the efficiency of the code in terms of CPU usage.

In general higher values for Ops/Sec are better as it indicates higher throughput, just as lower values for Cycles/Op are better. As the Cycles/Op and Ops/Sec metrics are derived from the same underlying data, they are reflecting the same performance characteristics from different perspectives. A high Ops/Sec should correspond to a low Cycles/Op, and vice versa.

Benchmark Output Options The benchmarking framework provides several options for outputting results of benchmarking data. By default, results are printed in a verbose human-readable format that is easy to understand and interpret. Alternatively, you can enable the `CONFIG_ZTEST_BENCHMARK_OUTPUT_CSV` Kconfig option to output results in a CSV format that can be easily imported by scripts for further analysis. The CSV output includes all the same metrics as the verbose output, but in a format that is more conducive to automated analysis and reporting.

The standard benchmark csv output format is as follows:

```
S,<suite name>,<benchmark name>,<sample size>,<total␣
↪cycles>,<mean>,<stddev>,<stderr>,<min>,<min sample>,<max>,<max sample>
```

The timed benchmark csv output format is as follows:

```
T,<suite name>,<benchmark name>,<samples>,<total time>,<work␣
↪time>,<ops/sec>,<cycles/op>
```

Important Considerations

- **Noise**: Benchmarking is inherently sensitive to system noise. To obtain as accurate results as possible, disable unnecessary background tasks and interrupts that may interfere with timing.
- **Cache Warming**: The first sample of a benchmark may often be slower due to cache misses and this can skew results slightly on small sample sizes, choose a sufficiently large number of samples to mitigate this effect.
- **Use setup/teardown functions**: It is *highly* encouraged to utilize the setup and teardown functions to isolate the code under test as much as possible. Excessive setup/teardown code in the benchmark code can introduce noise and skew results *significantly* for small critical paths.

API Reference

Zephyr Benchmarking Framework

2.13.3 Generating coverage reports

With Zephyr, you can generate code coverage reports to analyze which parts of the code are covered by a given test or application.

You can do this in two ways:

- In a real embedded target or QEMU, using Zephyr's gcov integration
- Directly in your host computer, by compiling your application targeting the POSIX architecture

Test coverage reports in embedded devices or QEMU

Overview GCC GCOV is a test coverage program used together with the GCC compiler to analyze and create test coverage reports for your programs, helping you create more efficient, faster running code and discovering untested code paths

In Zephyr, gcov collects coverage profiling data in RAM (and not to a file system) while your application is running. Support for gcov collection and reporting is limited by available RAM size and so is currently enabled only for QEMU emulation of embedded targets.

Details There are 2 parts to enable this feature. The first is to enable the coverage for the device and the second to enable in the test application. As explained earlier the code coverage with gcov is a function of RAM available. Therefore ensure that the device has enough RAM when enabling the coverage for it. For example a small device like frdm_k64f can run a simple test application but the more complex test cases which consume more RAM will crash when coverage is enabled.

To enable the device for coverage, select `CONFIG_HAS_COVERAGE_SUPPORT` in the Kconfig.board file.

To report the coverage for the particular test application set `CONFIG_COVERAGE`.

Steps to generate code coverage reports These steps will produce an HTML coverage report for a single application.

1. Build the code with CONFIG_COVERAGE=y.

```
west build -b mps2/an385 -- -DCONFIG_COVERAGE=y -DCONFIG_COVERAGE_DUMP=y
```

2. Capture the emulator output into a log file. You may need to terminate the emulator with `Ctrl-A X` for this to complete after the coverage dump has been printed:

```
$ ninja -Cbuild run | tee log.log
```

3. Generate the gcov `.gcda` and `.gcno` files from the log file that was saved:

```
$ python3 scripts/gen_gcov_files.py -i log.log
```

4. Find the gcov binary placed in the SDK. You will need to pass the path to the gcov binary for the appropriate architecture when you later invoke `gcovr`:

```
$ find $ZEPHYR_SDK_INSTALL_DIR -iregex ".*gcov"
```

5. Create an output directory for the reports:

```
$ mkdir -p coverage-report
```

6. Run `gcovr` to get the reports:

```
$ gcovr -r $ZEPHYR_BASE . --html -o coverage-report/coverage.html --
↪html-details --gcov-executable <gcov_path_in_SDK>
```

Coverage reports using the POSIX architecture

When compiling for the POSIX architecture, you utilize your host native tooling to build a native executable which contains your application, the Zephyr OS, and some basic HW emulation.

That means you can use the same tools you would while developing any other desktop application.

To build your application with gcc's gcov, simply set `CONFIG_COVERAGE` before compiling it. When you run your application, `gcov` coverage data will be dumped into the respective `gcda` and `gcno` files. You may postprocess these with your preferred tools. For example:

```
west build -b native_sim samples/hello_world -- -DCONFIG_COVERAGE=y
```

```
$ ./build/zephyr/zephyr.exe
# Press Ctrl+C to exit
$ lcov --capture --directory ./ --output-file lcov.info -q --rc␣
↪lcov_branch_coverage=1
$ genhtml lcov.info --output-directory lcov_html -q --ignore-errors source --
↪branch-coverage --highlight --legend
```

Note

You need a recent version of lcov (at least 1.14) with support for intermediate text format. Such packages exist in recent Linux distributions.

Alternatively, you can use gcovr (at least version 4.2).

Coverage reports using Twister

Zephyr's *twister script* can automatically generate a coverage report from the tests which were executed. You just need to invoke it with the `--coverage` command line option.

For example, you may invoke:

```
$ west twister --coverage -p qemu_x86 -T tests/kernel
```

or:

```
$ west twister --coverage -p native_sim -T tests/bluetooth
```

which will produce `twister-out/coverage/index.html` report as well as the coverage data collected by `gcovr` tool in `twister-out/coverage.json`.

Other reports might be chosen with `--coverage-tool` and `--coverage-formats` command line options.

To generate code coverage report including Zephyr sources as well as your application code outside of Zephyr repository (see *Application Types*) call Twister from your project directory with `--coverage-basedir $ZEPHYR_BASE` command line option, for example:

```
$ west twister --coverage -p native_sim --coverage-basedir $ZEPHYR_BASE -T␣
↪your_project_dir
```

Note

By default, Twister calls `gcovr` tool which filters source files assuming real paths are everywhere with all symlinks resolved, so when your development environment has directories with symlinks then, to avoid incomplete `gcovr` reports, either your *ZEPHYR_BASE* should contain a real path, or `lcov` tool used instead of `gcovr` with additional Twister command line option `--coverage-tool lcov`.

The process differs for unit tests, which are built with the host toolchain and require a different board:

```
$ west twister --coverage -p unit_testing -T tests/unit
```

which produces a report in the same location as non-unit testing.

Per-test coverage matrix By default coverage is aggregated per test scenario. To attribute coverage down to individual *Ztest* test cases – for example to answer "which tests exercised line X of `foo.c`" – pass `--coverage-per-test`:

```
$ west twister -p mps2/an385 -T tests/kernel --coverage --coverage-tool lcov \
    --coverage-per-test
```

This enables `CONFIG_ZTEST_COVERAGE_PER_TEST`, which resets the gcov counters before each test case and dumps an isolated, test-tagged coverage artifact after it. On platforms that support semi-hosting (ARM, RISC-V and Xtensa targets under QEMU, such as `mps2/an385`) the per-test data is written straight to the host filesystem, avoiding the large amount of serial console traffic that per-test dumping would otherwise produce; other platforms fall back to the serial console transport.

In addition to the usual aggregated coverage report, this produces:

- one `<scenario>.<test>.info` tracefile per test under each build's `coverage/tests/` directory, and
- `twister-out/coverage/test_matrix.json`, a machine-readable matrix with a `by_line` view (`{file: {line: [tests]}}`) and a `by_test` view (`{test: {file: [lines]}}`).

Per-test attribution is carried by the matrix (and its dashboard), not by the aggregated lcov report: each instance's coverage is collapsed to a single tracefile before aggregation, so the end-of-run reporting stays proportional to the number of instances rather than the total number of test cases.

> **Note**
>
> `--coverage-per-test` requires the `lcov` coverage tool, because the matrix relies on lcov's per-test `TN` tracefile records; `gcovr` has no equivalent. It also implies `--coverage`.

Visualizing the matrix `test_matrix.json` can be turned into a self-contained, interactive HTML dashboard with the standalone scripts/gen_test_matrix_dashboard.py script (it has no dependency on Twister and can be run on any previously generated matrix):

```
$ scripts/gen_test_matrix_dashboard.py -i twister-out/coverage/test_matrix.json
```

This writes `twister-out/coverage/test_matrix.html`, which lists – per test – how many files and lines it covers and how many lines it covers *uniquely* (that no other test reaches, useful for spotting redundant or load-bearing tests), and lets you drill into the files a test covers or look up which tests cover a given file and line.

Using different toolchains Twister looks at the environment variable `ZEPHYR_TOOLCHAIN_VARIANT` to check which gcov tool to use by default. The following are used as the default for the Twister `--gcov-tool` argument default:

Toolchain	`--gcov-tool` value
host	`gcov`
llvm	`llvm-cov gcov`
zephyr	`gcov`

2.13.4 BabbleSim

BabbleSim and Zephyr

In the Zephyr project we use the Babblesim simulator to test some of the Zephyr radio protocols, including the Bluetooth LE stack, 802.15.4, and some of the networking stack.

BabbleSim is a physical layer simulator, which in combination with the Zephyr bsim boards can be used to simulate a network of Bluetooth LE and 15.4 devices. When we build Zephyr targeting a bsim board we produce a Linux executable, which includes the application, Zephyr OS, and models of the HW.

When there is radio activity, this Linux executable will connect to the BabbleSim Phy simulation to simulate the radio channel.

In the BabbleSim documentation you can find more information on how to get and build the simulator. In the nrf52_bsim, nrf5340bsim, and nrf54l15bsim boards documentation you can find more information about how to build Zephyr targeting these particular boards, and a few examples.

Types of tests

Tests without radio activity: bsim tests with twister The bsim boards can be used without radio activity, and in that case, it is not necessary to connect them to a physical layer simulation. Thanks to this, these target boards can be used just like native_sim with *twister*, to run all standard Zephyr twister tests, but with models of a real SOC HW, and their drivers.

Tests with radio activity When there is radio activity, BabbleSim tests require at the very least a physical layer simulation running, and most, more than 1 simulated device. Due to this, these tests are executed by running a dedicated script for each test which starts both the needed simulated devices, and the physical layer executable with their required parameters, and any other tool which may be required.

To be able to run them with twister, the *bsim harness* should be used.

These tests are kept in the tests/bsim/ folder.

Check the sub-sections below for more information about how to build and run them, as well as the conventions they follow.

There are two main sets of tests:

- Self checking embedded application/tests: In which some of the simulated devices applications are built with some checks which decide if the test is passing or failing. These embedded applications tests use the bs_tests system to report the pass or failure, and in many cases to build several tests into the same binary.
- Test using the EDTT tool, in which a EDTT (python) test controls the embedded applications over an RPC mechanism, and decides if the test passes or not. Today these tests include a very significant subset of the BT qualification test suite.

More information about how different tests types relate to BabbleSim and the bsim boards can be found in the bsim boards tests section.

Test coverage and BabbleSim

As the bsim boards are based on the POSIX architecture, you can easily collect test coverage information.

Check *the page on coverage generation* for more info, and note you can just pass the `--coverage` option to twister to automatically build with `CONFIG_COVERAGE` and generate the coverage report.

Building and running the tests

Check the nrf52_bsim page for instructions on how to set up the simulator.

You can build and run these tests with *twister*. To run multidevice tests, you will need to pass twister the option `--fixture bsim_multi_test`.

For example, from ${ZEPHYR_BASE}, you can build and run one of the BT tests with:

```
twister -p nrf52_bsim/native -T tests/bsim/bluetooth/host/adv/chain/ --fixture␣
↪bsim_multi_test
```

If the test binaries have already been built, you can also run a test directly using its individual test script. For example:

```
BOARD=nrf52_bsim/native␣
↪tests/bsim/bluetooth/host/adv/chain/tests_scripts/adv_chain.sh
```

Twister command line options like `-n`, `--no-clean`, `--aggressive-no-clean`, or `-b`, `--build-only` are likely to be useful when debugging or fixing issues.

Legacy batch scripts Until the twister bsim harness was added, building and running multidevice bsim tests relied on two scripts in the tests/bsim/ folder: `compile.sh` and `run_parallel.sh`. These scripts were used by the CI system to build the needed images and execute these tests in batch. They were also meant as utilities for users to build and execute their tests. These scripts can still be used, but users are recommended to transition to using *twister* and `tests.yaml` definitions.

These scripts expect a few environment variables to be set. For example, from Zephyr's root folder, you can run:

```
# Build all the tests
${ZEPHYR_BASE}/tests/bsim/compile.sh

# Run them (in parallel)
RESULTS_FILE=${ZEPHYR_BASE}/myresults.xml \
   SEARCH_PATH=${ZEPHYR_BASE}/tests/bsim \
      ${ZEPHYR_BASE}/tests/bsim/run_parallel.sh
```

Or to build and run only a specific subset, e.g. host advertising tests:

```
# Build the Bluetooth host advertising tests
${ZEPHYR_BASE}/tests/bsim/bluetooth/host/adv/compile.sh

# Run them (in parallel)
RESULTS_FILE=${ZEPHYR_BASE}/myresults.xml \
   SEARCH_PATH=${ZEPHYR_BASE}/tests/bsim/bluetooth/host/adv \
      ${ZEPHYR_BASE}/tests/bsim/run_parallel.sh
```

Check the `run_parallel.sh` help for more options and examples on how to use this script to run the tests in batch.

After building the tests' required binaries you can run a test directly using its individual test script.

For example you can build the required binaries for the networking tests with

```
WORK_DIR=${ZEPHYR_BASE}/bsim_out ${ZEPHYR_BASE}/tests/bsim/net/compile.sh
```

and then directly run one of the tests:

```
${ZEPHYR_BASE}/tests/bsim/net/sockets/echo_test/tests_scripts/echo_test_802154.sh
```

Conventions

Test code See the Bluetooth sample test for conventions that apply to test code.

Test scripts Please follow the existing conventions and do not design one-off bespoke runners (e.g. a python script, or another shell abstraction).

The rationale is that it is easier and faster for the maintainers to perform tree-wide updates for build system or compatibility changes if the tests are run in the same manner, with the same variables, etc..

If you have a good idea for improving your test script, please make a PR changing *all* the test scripts in order to benefit everyone and conserve homogeneity. You can of course discuss it first in an RFC issue or on the babblesim discord channel.

Scripts starting with an underscore (_) are not automatically discovered and run. They can serve as either helper functions for the main script, or can be used for local development utilities, e.g. building and running tests locally, debugging, etc..

Here are the conventions:

- Each test is defined by a shell script with the extension `.sh`, in a subfolder called `tests_scripts/`.
- It is recommended to run a single test per script file. It allows for better parallelization of the runs in CI.
- Scripts expect that the binaries they require are already built. They should not compile binaries.
- Scripts will spawn the processes for every simulated device and the physical layer simulation.
- Scripts must return 0 to the invoking shell if the test passes, and not 0 if the test fails.
- Each test must have a unique simulation id, to enable running different tests in parallel.
- Neither the scripts nor the images should modify the workstation filesystem content beyond the `${BSIM_OUT_PATH}/results/<simulation_id>/` or `/tmp/` folders. That is, they should not leave stray files behind.
- Tests that require several consecutive simulations (e.g, if simulating a device pairing, powering off, and powering up after as a new simulation) should use separate simulation ids for each simulation segment, ensuring that the radio activity of each segment can be inspected a posteriori.
- Avoid overly long tests. If the test takes over 20 seconds of runtime, consider if it is possible to split it in several separate tests.
- If the test takes over 5 seconds, set `EXECUTE_TIMEOUT` to a value that is at least 5 times bigger than the measured run-time.
- Do not set `EXECUTE_TIMEOUT` to a value lower than the default.
- Tests should not be overly verbose: less than a hundred lines are expected on the outputs. Do make use of `LOG_DBG()` extensively, but don't enable the `DBG` log level by default.
- Test scripts that use the Physical layer simulation are expected to pass any extra parameters passed to the test script directly to the Phy executable. This is used, for example, to run the Phy in check mode (`-c`), in which the Phy verifies that the last simulation and this one have produced the exact same radio traffic.

2.13.5 CI Test Plan Selector (test_plan_v2)

The CI test plan selector is a modular Python script located at `scripts/ci/test_plan_v2.py`. Its purpose is to analyse the set of files changed in a Pull Request and emit a targeted twister test plan — exercising only the tests that are plausibly affected by the change while avoiding a full tree-wide run on every commit.

The script writes two artefacts:

- `testplan.json` — passed to twister via `twister --load-tests`.
- `.testplan` — a plain `KEY=value` environment file consumed by CI orchestration scripts, containing `TWISTER_TESTS`, `TWISTER_NODES`, and `TWISTER_FULL`.

Architecture

The selector is built around a *strategy pipeline*. Each strategy is an independent analyser that:

1. Receives the list of changed files (or the subset not yet consumed by earlier strategies).
2. Inspects those files according to its own logic.
3. Returns a list of `TwisterCall` descriptors and the set of files it has *handled*.

The `Orchestrator` drives the pipeline, merges all results into a `PlanAccumulator`, deduplicates test suites, and writes the output files.

Strategy ordering

Strategies run in a fixed order that reflects two principles:

Specificity over generality. The most precise strategies run first. A file that lives inside `tests/kernel/sched/` is handled definitively by `DirectTestStrategy` (run exactly those tests) before the catch-all `MaintainerAreaStrategy` can add an entire Kernel area sweep.

Consume-before-additive. Consuming strategies run before additive ones. Once a consuming strategy claims a file, downstream strategies never see it. This prevents a change to a board file from also triggering a driver compat scan and a Kconfig sweep for the same path.

The current order is:

#	Strategy	Consumes	Rationale
0	`ComplexityStrategy`	No	Scores the patchset using pydriller and lizard. Must run first so the score is available to `RiskClassifierStrategy`. Emits no twister calls.
1	`IgnoreStrategy`	Yes	Silently drops files that can never affect tests (documentation, CI workflows, tooling). Runs early so later strategies never waste effort on ignored paths.
1b	`BoilerplateFilter`	Yes	Consumes files whose entire diff consists of whitespace adjustments, blank-line changes, SPDX licence identifiers, copyright notices, or bare comment delimiters. Such changes cannot affect runtime behaviour, so removing them early keeps the pool clean for substantive strategies. Requires a git commit range; no-op otherwise.
2	`DirectTestStrategy`	Yes	A changed test or sample file must trigger *only* that test, not an entire area-wide sweep. Consuming this file prevents `MaintainerAreaStrategy` from also adding every test in the Kernel area when `tests/kernel/sched/main.c` is modified.
3	`SnippetStrategy`	Yes	Snippet changes are self-contained: only tests that declare the snippet in `required_snippets` need to run. Consuming prevents downstream strategies from treating a snippet YAML as an unknown configuration file.
4	`BoardStrategy`	Yes	Board-specific changes require a targeted integration test on every board variant. Consuming prevents a board `.yaml` file from also matching Kconfig and Header strategies.
5	`SoCStrategy`	Yes	SoC-level changes affect every board built on that SoC family. Consuming prevents the same `soc/` path from reaching the Kconfig sweep.
6	`ManifestStrategy`	Yes	A changed `west.yml` requires module-tagged integration tests, not a generic maintainer-area run. Consuming the manifest file prevents the catch-all from adding unrelated tests.
7	`DriverCompatStrategy`	No	Additive: driver files may also be covered by area patterns, so the file is not consumed and `KconfigImpactStrategy` and `MaintainerAreaStrategy` can add further coverage.
8	`DtsBindingStrategy`	No	Additive: binding changes combine overlay-based test selection with board-targeted area calls.
9	`KconfigImpactStrategy`	No	Additive: Kconfig changes may affect many unrelated files; only consumes when a non-widespread symbol is found, leaving widespread-symbol files for the catch-all.
10	`HeaderImpactStrategy`	No	Additive: header changes trace include users back to maintainer areas. Widespread headers (included in more than the configured threshold) are skipped.
11	`MaintainerAreaStrategy`	No	Catch-all: matches any remaining file against `MAINTAINERS.yml` area patterns and emits `--test-pattern` calls for each matching area that has a non-empty `tests:` list.

Boilerplate filter

`BoilerplateFilter` runs immediately after `IgnoreStrategy` and before any test-selection strategy. It inspects the actual diff of each changed file (via `git diff`) and consumes those whose entire content change is non-substantive:

- **Whitespace and blank-line changes** — detected by `git diff -w --ignore-blank-lines`: if that command produces no output for a file, every changed line is whitespace or blank.
- **SPDX and copyright header edits** — lines that begin with `SPDX-License-Identifier:`,

`SPDX-FileCopyrightText:`, or the word `Copyright`.

- **Comment-delimiter-only lines** — lines whose non-whitespace content consists entirely of `/*`, `*/`, `//`, `#`, or `*` characters (e.g. reformatted block-comment borders).

A file is consumed only when **every** added or removed line in its diff falls into one of those categories. A single substantive line (a changed statement, macro, or declaration) causes the file to pass through to downstream strategies unchanged.

The filter is a no-op when `--commits` is not supplied (e.g. when using `--modified-files`), because the diff cannot be computed without a commit range. In that case all files remain in the pool.

Consume vs. additive behaviour

Every strategy subclass carries a class attribute `consumes: bool`.

When `consumes = True`:
Files returned in the *handled* set are removed from the `remaining` pool before the next strategy runs. Use this when the strategy is *authoritative* for its file type — i.e. when seeing the file in a later strategy would produce redundant or incorrect results.

When `consumes = False` (default):
Files remain in the pool regardless of what the strategy returns. All downstream strategies receive the same file list. Use this for additive strategies that contribute *additional* test coverage without claiming exclusive ownership.

> **Note**
>
> The `Orchestrator` skips a strategy entirely when the `remaining` pool is empty. Because non-consuming strategies see all files, "remaining" only shrinks through consuming strategies. As soon as the pool is empty the orchestrator stops calling strategies.

Full-run and fallback conditions

The orchestrator signals `TWISTER_FULL=True` in `.testplan` when either of the following is true:

1. **Unresolved files.** After all strategies have run, at least one changed file was not *handled* by any strategy. Rather than silently skipping coverage for an unknown path, the script falls back to requesting a full run so no regression slips through.
2. **Explicit full-run signal.** A strategy returns a `TwisterCall` with `full_run=True`. When the orchestrator encounters such a call, it immediately sets `TWISTER_FULL=True`, empties the remaining pool, and stops executing further calls. This mechanism allows a strategy to opt out of targeted selection (e.g. when a core subsystem header used by the entire tree is modified).

When `TWISTER_FULL=True` the CI script is expected to discard `testplan.json` and run twister without `--load-tests`.

Node count calculation

`TWISTER_NODES` in `.testplan` is computed as follows:

- `0` — no tests were selected.
- `1` — fewer than `--tests-per-builder` tests were selected (fits in one builder).
- `ceil(total / tests_per_builder)` — otherwise. The ceiling ensures that no builder is over-capacity when the division is not exact.

The default value of `--tests-per-builder` is `900`.

Adding a new strategy

1. **Subclass** `SelectionStrategy` and implement the two abstract members:

```
class MyStrategy(SelectionStrategy):

    consumes: bool = False  # or True if authoritative

    @property
    def name(self):
        return "MyStrategy"

    def analyze(self, changed_files):
        # inspect changed_files
        calls = [...]       # list of TwisterCall
        handled = {...}     # subset of changed_files this strategy owns
        return calls, handled
```

2. **Decide consume vs. additive.** Ask: "If a downstream strategy also sees this file, will it produce useful additional coverage, or redundant/wrong results?" If redundant/wrong: `consumes = True`.
3. **Insert in the correct position** in `build_strategies()`. As a rule of thumb:
   - Consuming strategies belong before all additive strategies.
   - More-specific strategies belong before less-specific ones.
   - Strategies with no side-effects on the file pool (additive) can be ordered by cost — cheapest first.
4. **Guard optional dependencies** with a late import inside `analyze` and return `[], set()` gracefully when the dependency is missing. Use the `# noqa: PLC0415` comment to silence the late-import warning.
5. **Write unit tests** in `scripts/tests/ci/test_test_plan_v2.py` covering:
   - The happy path (files correctly identified and routed).
   - The no-op path (irrelevant files produce no calls and no handled set).
   - Edge cases (missing files on disk, malformed YAML, empty inputs).

Shared pipeline context

`PipelineContext` is a dataclass threaded through all strategies via `build_strategies()`. It currently carries:

- `complexity_score` — aggregate patchset score written by `ComplexityStrategy` and read by `RiskClassifierStrategy`.
- `file_metrics` — per-file `ComplexityMetrics` mappings for detailed logging and downstream decision-making.

New cross-strategy state should be added as typed fields on `PipelineContext` rather than as strategy-level instance variables.

Test strategy for validating selection behaviour

The test suite lives in `scripts/tests/ci/test_test_plan_v2.py` and is run with pytest:

```
pytest scripts/tests/ci/test_test_plan_v2.py -v
```

Tests are organised into one class per strategy or shared component. The following categories of tests are expected for each new strategy:

Unit tests (no Zephyr tree required)
Use a `tmp_path` fixture to create the minimum filesystem structures (`board.yml`, `snippet.yml`, `testcase.yaml`, etc.) needed to exercise each code path. These tests should be fast and hermetic.

Helper method tests
Test internal methods (path-walking, YAML parsing, regex extraction) in isolation. Pass synthetic content rather than relying on real files in the repository.

Orchestrator integration tests
Use mock strategies and a mock `TwisterExecutor` to test the orchestrator's consume-vs-additive logic, full-run signalling, and `.testplan` output without invoking twister.

Real-tree integration tests (optional, `@pytest.mark.integration`)
May be added and marked with `@pytest.mark.integration` to be excluded from the default run. These are only meaningful on a complete Zephyr checkout.

Minimal test checklist for a new strategy

- The strategy returns no calls and no handled files when no relevant files are in the input list.
- The strategy correctly identifies and returns the expected `TwisterCall` for a known-good input.
- The strategy does not crash when required files are absent from the filesystem.
- For consuming strategies: the handled set equals the matched files and nothing else.
- For additive strategies: the handled set is empty or equals only the files the strategy is authoritative for.

CLI reference

```
usage: test_plan_v2.py [-c A..B] [-m FILE] [-f PATH]
                       [-o FILE] [-p PLATFORM] [--maintainers-file FILE]
                       [-T DIR] [--quarantine-list FILE]
                       [--tests-per-builder N] [--disable-strategy NAME]
                       [--detailed-test-id]

-c A..B              Git commit range (e.g. ``main..HEAD``).  Changed files
                     are derived from ``git diff --name-only A..B``.
-m FILE              JSON file containing a list of changed file paths.
-f PATH              Treat PATH as a changed file (repeatable).
-o FILE              Output JSON file (default: ``testplan.json``).
-p PLATFORM          Restrict all selections to this platform (repeatable).
--maintainers-file   Path to ``MAINTAINERS.yml``.
-T DIR               Extra testsuite root forwarded to every twister call.
--quarantine-list    Quarantine YAML forwarded to twister.
--tests-per-builder  Tests per CI builder node (default: 900).
--disable-strategy   Skip a strategy by name (repeatable).
--detailed-test-id   Pass ``--detailed-test-id`` to twister.
```

2.14 Static Code Analysis (SCA)

Support for static code analysis tools in Zephyr is possible through CMake.

The build setting **`ZEPHYR_SCA_VARIANT`** can be used to specify the SCA tool to use. `ZEPHYR_SCA_VARIANT` is also supported as *environment variable*.

Use `-DZEPHYR_SCA_VARIANT=<tool>`, for example `-DZEPHYR_SCA_VARIANT=sparse` to enable the static analysis tool `sparse`.

2.14.1 SCA Tool infrastructure

Support for an SCA tool is implemented in a `sca.cmake` file. The `sca.cmake` must be placed under *SCA_ROOT*`/cmake/sca/`*tool*`/sca.cmake`. Zephyr itself is always added as an **SCA_ROOT** but the build system offers the possibility to add additional folders to the **SCA_ROOT** setting.

You can provide support for out of tree SCA tools by creating the following structure:

```
<sca_root>/                 # Custom SCA root
└── cmake/
    └── sca/
        └── <tool>/         # Name of SCA tool, this is the value given to␣
↪ZEPHYR_SCA_VARIANT
            └── sca.cmake   # CMake code that configures the tool to be used with␣
↪Zephyr
```

To add `foo` under `/path/to/my_tools/cmake/sca` create the following structure:

```
/path/to/my_tools
         └── cmake/
             └── sca/
                 └── foo/
                     └── sca.cmake
```

To use `foo` as SCA tool you must then specify `-DZEPHYR_SCA_VARIANT=foo`.

Remember to add `/path/to/my_tools` to **SCA_ROOT**.

SCA_TOOL can be set as a regular CMake setting using `-DSCA_ROOT=<sca_root>`, or added by a Zephyr module in its `module.yml` file, see *Zephyr Modules - Build settings*

Compiler and linker launchers

An SCA tool that needs to observe or wrap the compilation and link commands does so by setting the `CMAKE_<LANG>_COMPILER_LAUNCHER` and `CMAKE_<LANG>_LINKER_LAUNCHER` variables from its `sca.cmake`. They must be set as normal variables, not as cache entries.

A launcher set this way replaces any launcher that was already configured, `ccache` included. A tool that is a transparent wrapper, meaning it runs the command it is given unmodified, may instead keep the previous launcher by appending it:

```
set(CMAKE_C_COMPILER_LAUNCHER ${my_wrapper} ${CMAKE_C_COMPILER_LAUNCHER})
```

2.14.2 Native SCA Tool support

The following is a list of SCA tools natively supported by Zephyr build system.

Clang static analyzer support

Clang Static Analyzer is built on top of Clang and LLVM. Strictly speaking, the analyzer is part of Clang, as Clang consists of a set of reusable C++ libraries for building powerful source-level tools. The static analysis engine used by the Clang Static Analyzer is a Clang library, and has the capability to be reused in different contexts and by different clients.

LLVM provides various methods to run the analyzer on a codebase, through either a dedicated set of tools (scan-build and analyze-build), or via command line arguments when running clang ('–analyze').

- 'scan-build' utility comes as the most convenient way for projects using a simple $CC makefile variables, as it will wraps and replace the compiler calls to perform it's analysis.
- 'analyze-build' utility is a sub-tool from 'scan-build', it only relies on a 'compile_commands.json' database to perform the analysis.
- clang option '–analyze' will run the analyzer alongside the build, but objects files are not generated, making any link stage impossible. In our case the first link stage will fail and stop the analysis.

Because of it's complexe build infrastructure, invoking clang analyzer with 'analyze-build' is the most simple way to analyze a Zephyr project.

Clang static analyzer documentation

Installing clang analyzer 'scan-build' and its sub-tool 'analyze-build' come natively with llvm as part of the binaries. Make sure to have the binary directory accessible into your PATH.

'scan-build' is also available as a standalone python package available on pypi.

```
pip install scan-build
```

> **Note**
>
> The analyser requires that the project builds with a LLVM toolchain, and produces a 'compile_commands.json' database.

Run clang static analyzer To run clang static analyzer, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=clang` parameter, alongside the llvm toolchain parameters, e.g.

```
west build -b qemu_x86 samples/userspace/hello_world_user -- -
↪DZEPHYR_TOOLCHAIN_VARIANT=host/llvm -DLLVM_TOOLCHAIN_PATH=... -
↪DZEPHYR_SCA_VARIANT=clang
```

> **Note**
>
> By default, clang static analyzer produces a html report, but various other outputs can be selected with options (sarif, plist, html)

Configuring clang static analyzer Clang static analyzer can be controlled using specific options. To get an exhaustive list of available options, report to the 'analyze-build' helper and 'scan-build' helper.

```
analyze-build --help
```

Options already activated by default:

- –analyze-headers : Also analyze functions in #included files.

Parameter	Description
`CLANG_SCA_OPTS`	A semicolon separated list of 'analyze-build' options.

These parameters can be passed on the command line, or be set as environment variables.

```
west build -b stm32h573i_dk samples/hello_world -- -
↪DZEPHYR_TOOLCHAIN_VARIANT=host/llvm -DLLVM_TOOLCHAIN_PATH=... -
↪DZEPHYR_SCA_VARIANT=clang -DCLANG_SCA_OPTS="--sarif;--verbose"
```

CodeChecker support

CodeChecker is a static analysis infrastructure. It executes analysis tools available on the build system, such as Clang-Tidy, Clang Static Analyzer and Cppcheck. Refer to the analyzer's websites for installation instructions.

Installing CodeChecker CodeChecker itself is a python package available on pypi.

```
pip install codechecker
```

Building with CodeChecker To run CodeChecker, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=codechecker` parameter, e.g.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -
↪DZEPHYR_SCA_VARIANT=codechecker
```

Configuring CodeChecker CodeChecker uses different command steps, each with their respective configuration parameters. The following table lists all these options.

Parameter	Description
CODECHECKER_ANALY	The number of threads to use in analysis. (default: <CPU count>)
CODECHECKER_ANALY	Arguments passed to the `analyze` command directly. (e.g. `--timeout;360`)
CODECHECKER_CLEAN	Perform a cleanup after parsing/storing. This will remove all `plist` files.
CODECHECKER_CONFI	A JSON or YAML file with configuration options passed to all commands.
CODECHECKER_EXPOR	A comma separated list of report types. Allowed types are: `html,json,codeclimate,gerrit,baseline`.
CODECHECKER_NAME	The CodeChecker run metadata name. (default: `zephyr`)
CODECHECKER_PARSE	By default, CodeChecker identified issues will not fail the build, set this option to fail during analysis.
CODECHECKER_PARSE	Arguments passed to the `parse` command directly. (e.g. `--verbose;debug`)
CODECHECKER_PARSE	Skip parsing the analysis, useful if you simply want to store the results.
CODECHECKER_STORE	Run the `store` command after analysis.
CODECHECKER_STORE	Arguments passed to the `store` command directly. Implies `CODECHECKER_STORE`. (e.g. `--url;localhost:8001/Default`)
CODECHECKER_STORE	Pass an identifier `--tag` to the `store` command.
CODECHECKER_TRIM_	Trim the defined path from the analysis results, e.g. `/home/user/zephyrproject`. The value from `west topdir` is added by default.

These parameters can be passed on the command line, or be set as environment variables.

Running twister with CodeChecker When running CodeChecker as part of `twister` some default options are set as following:

Parameter	Value
CODECHECKER_ANALYZE_JOBS	1
CODECHECKER_NAME	`<board target>:<testsuite name>`
CODECHECKER_STORE_TAG	The value from `git describe` in the application source directory.

To override these values, set an environment variable or pass them as extra arguments.

Coverity

Coverity Scan is a service by which Black Duck provides the results of analysis on open source coding projects to open source code developers that have registered their products with Coverity Scan.

This integration was only tested with scan.coverity.com and the tool distribution available through this service.

Generating Build Data Files To use this integration, coverity tool distribution must be found in your *PATH* environment and *west build* should be called with a `-DZEPHYR_SCA_VARIANT=coverity` parameter, e.g.

```
west build -b qemu_cortex_m3 samples/hello_world -- -DZEPHYR_SCA_VARIANT=coverity
```

Results of the scan will be generated as `build/sca/coverity`.

You can also set `COVERITY_OUTPUT_DIR` as the destination for multiple and incremental scan results.

Result Analysis Follow the instructions on https://scan.coverity.com for uploading results.

Parasoft C/C++test support

Parasoft C/C++test is a software testing and static analysis tool for C and C++. It is a commercial software and you must acquire a commercial license to use it.

Documentation of C/C++test can be found at https://docs.parasoft.com/. Please refer to the documentation for how to use it.

Generating Build Data Files To use C/C++test, `cpptestscan` must be found in your *PATH* environment variable. And *west build* should be called with a `-DZEPHYR_SCA_VARIANT=cpptest` parameter, e.g.

```
west build -b qemu_cortex_m3 zephyr/samples/hello_world -- -
↪DZEPHYR_SCA_VARIANT=cpptest
```

A `.bdf` file will be generated as `build/sca/cpptest/cpptestscan.bdf`.

Generating a report file Please refer to Parasoft C/C++test documentation for more details.

To import and generate a report file, something like the following should work.

```
cpptestcli -data out -localsettings local.conf -bdf␣
↪build/sca/cpptest/cpptestscan.bdf -config "builtin://Recommended Rules" -report␣
↪out/report
```

You might need to set `bdf.import.c.compiler.exec`, `bdf.import.cpp.compiler.exec`, and `bdf.import.linker.exec` to the toolchain *west build* used.

Devicetree diagnostics (`dtdoctor`)

`dtdoctor` is a static analysis tool that helps diagnose Devicetree-related build errors.

It intercepts error messages from the compiler and linker and, when they refer to unresolved Devicetree device symbols (e.g. `__device_dts_ord_*`), provides detailed information about what might be causing the error and how to fix it.

Using dtdoctor To enable `dtdoctor`, build with `-DZEPHYR_SCA_VARIANT=dtdoctor`.

For example:

```
west build -b reel_board samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=dtdoctor
```

ECLAIR support

Bugseng ECLAIR is a certified static analysis tool and platform for software verification. Applications range from coding rule validation, with a particular emphasis on the MISRA and BARR-C coding standards, to the computation of software metrics, to the checking of independence and freedom from interference among software components, to the automatic detection of important classes of software errors.

Prerequisites The ECLAIR tool must be installed and made available in the operating system's PATH variable.

To verify the installation, you can run:

```
eclair -version
```

A valid license or trial license is required to use ECLAIR. To request a trial license, visit this page.

Running ECLAIR To run ECLAIR, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=eclair` parameter.

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=eclair
```

> **Note**
>
> This will only invoke the ECLAIR analysis with the predefined ruleset `first_analysis`. If you want to use a different ruleset, you need to provide a configuration file. See the next section for more information.

Configurations The configuration of the ECLAIR SCA environment can either be done via a CMake options file or with adapted options as command line arguments.

To invoke a CMake options file into the ECLAIR call, you can define the `ECLAIR_OPTIONS_FILE` variable, for example:

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=eclair -
↪DECLAIR_OPTIONS_FILE=my_options.cmake
```

The default (if no config file is given) configuration is always `first_analysis`, which is a tiny selection of rules to verify that everything is correctly working.

If the default configuration wants to be overwritten via the command line and not via an options file, that can be achieved by giving the argument `-DOption=ON|OFF`.

For example:

```
west build -b mimxrt1064_evk samples/basic/blinky -- -DZEPHYR_SCA_VARIANT=eclair -
↪DECLAIR_REPORTS_SARIF=ON
```

Zephyr is a large and complex project, so the configuration sets are split into the Zephyr's guidelines selection (taken from https://docs.zephyrproject.org/latest/contribute/coding_guidelines/index.html) in five sets to make it more digestible to use on a private machine:

- first_analysis (default): a tiny selection of the project's coding guidelines to verify that everything is correctly working.
- STU: Selection of the project's coding guidelines, which can be verified by analyzing the single translation units independently.
- STU_heavy: Selection of complex STU project coding guidelines that require a significant amount of time.
- WP: All whole program project coding guidelines ("system" in MISRA's parlance).
- std_lib: Project coding guidelines about the C Standard Library.

In addition, the zephyr_guidelines ruleset contains all the main rules listed in the Coding Guidelines.

Related CMake options:

- `ECLAIR_RULESET_FIRST_ANALYSIS`
- `ECLAIR_RULESET_STU`
- `ECLAIR_RULESET_STU_HEAVY`
- `ECLAIR_RULESET_WP`
- `ECLAIR_RULESET_STD_LIB`
- `ECLAIR_RULESET_ZEPHYR_GUIDELINES`

User-defined ruleset If you want to use your own defined ruleset instead of the predefined Zephyr coding guidelines rulesets, you can do so by setting `ECLAIR_RULESET_USER=ON`. Create your own ruleset file for ECLAIR with the following naming format: `analysis_<RULESET>.ecl`. After creating the file, define the name of the ruleset for ECLAIR with the CMake variable `ECLAIR_USER_RULESET_NAME`. If the ruleset file is not in the application source directory, you can define the path to the ruleset file with the CMake variable `ECLAIR_USER_RULESET_PATH`. This configuration takes relative paths and absolute paths.

Related CMake options and variables:

- `ECLAIR_RULESET_USER`
- `ECLAIR_USER_RULESET_NAME`
- `ECLAIR_USER_RULESET_PATH`

Generate additional report formats ECLAIR can generate additional report formats (e.g., DOC, ODT, XLSX) and different variants of reports in addition to the default ecd file. Following additional reports and report formats can be generated:

- Metrics in spreadsheet format.
- Findings in spreadsheet format.
- Findings in SARIF format.
- Summary report in plain textual format.
- Summary report in DOC format.
- Summary report in ODT format.
- Summary report in HTML format.
- Detailed reports in txt format.
- Detailed report in DOC format.
- Detailed report in ODT format.
- Detailed report in HTML format.

Related CMake options:

- ECLAIR_METRICS_TAB
- ECLAIR_REPORTS_TAB
- ECLAIR_REPORTS_SARIF
- ECLAIR_SUMMARY_TXT
- ECLAIR_SUMMARY_DOC
- ECLAIR_SUMMARY_ODT
- ECLAIR_SUMMARY_HTML
- ECLAIR_FULL_TXT
- ECLAIR_FULL_DOC
- ECLAIR_FULL_ODT
- ECLAIR_FULL_HTML

Detail level of full reports The detail level of the txt and doc full reports can also be adapted by a configuration. In this case, the following configurations are available:

- Show all areas
- Show only the first area

Related CMake options:

- ECLAIR_FULL_DOC_ALL_AREAS
- ECLAIR_FULL_DOC_FIRST_AREA
- ECLAIR_FULL_TXT_ALL_AREAS
- ECLAIR_FULL_TXT_FIRST_AREA

GCC static analysis support

Static analysis was introduced in GCC 10 and it is enabled with the option `-fanalyzer`. This option performs a much more expensive and thorough analysis of the code than traditional warnings.

Run GCC static analysis To run GCC static analysis, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=gcc` parameter, e.g.

```
west build -b qemu_x86 samples/userspace/hello_world_user -- -
↪DZEPHYR_SCA_VARIANT=gcc
```

Configuring GCC static analyzer GCC static analyzer can be controlled using specific options.

- Options controlling the analyzer
- Options controlling the diagnostic message formatting

Parameter	Description
GCC_SCA_OPTS	A semicolon separated list of GCC analyzer options.

These parameters can be passed on the command line, or be set as environment variables.

```
west build -b stm32h573i_dk samples/hello_world -- -DZEPHYR_SCA_VARIANT=gcc -
↪DGCC_SCA_OPTS="-fdiagnostics-format=json;-fanalyzer-verbosity=3"
```

> **Note**
>
> GCC static analyzer is under active development, and each new release comes with new options. This page gives an overview of options and fix introduced with each new release of the analyzer.

Latest version of the analyzer Since the Zephyr toolchain may not include the most recent version of the GCC static analyzer, the GCC static analysis can be run with a more recent GNU Arm embedded toolchain to take advantage of the latest analyzer version.

```
west build -b stm32h573i_dk samples/hello_world -- -DZEPHYR_SCA_VARIANT=gcc -
↪DZEPHYR_TOOLCHAIN_VARIANT=gnuarmemb -DGNUARMEMB_TOOLCHAIN_PATH=...
```

IAR C-STAT support

IAR C-STAT is a comprehensive static analysis tool for C/C++ source code. It can find errors and vulnerabilities supporting a number of coding standards such as MISRA C, MISRA C++, CERT C/C++ and CWE.

Installing IAR C-STAT IAR C-STAT comes pre-installed with the IAR Build Tools and with the IAR Embedded Workbench. Refer to your respective product's documentation for details.

Building with IAR C-STAT To run IAR C-STAT, you will need CMake 4.1.0 or later. When building with *west build* append the additional parameter to select IAR C-STAT `-DZEPHYR_SCA_VARIANT=iar_c_stat`, e.g.

```
west build -b stm32f429ii_aca samples/basic/blinky -- -
↪DZEPHYR_SCA_VARIANT=iar_c_stat
```

Configuring IAR C-STAT The IAR C-STAT accepts parameters for customizing the analysis. The following table lists the supported options.

Parameter	Description
CSTAT_RULESE	The pre-defined ruleset to be used. (default: `stdchecks`, accepted values: `all`, `cert`,`misrac2004`,`misrac2012`,`misrac++2008`,`stdchecks`,`security`)
CSTAT_ANALYZI	The number of threads to use in analysis. (default: <CPU count>)
CSTAT_ANALYZI	Arguments passed to the `analyze` command directly. (e.g. `--timeout=900`; `--deterministic`;`--fpe`)
CSTAT_DB	Override the default location of the C-STAT SQLite database. (e.g. `/home/user/cstat.db`)
CSTAT_CLEANUI	Perform a cleanup of the C-STAT SQLite database. (e.g. `true`)

These parameters can be passed on the command line, or be set as environment variables. Below you will find an example of how to enable and combine non-standard rulesets at will:

```
# From the root of the zephyr repository
west build -b stm32f429ii_aca samples/basic/blinky -- -
↪DZEPHYR_SCA_VARIANT=iar_c_stat -DCSTAT_RULESET=misrac2012,cert
```

include-what-you-use (IWYU) support

include-what-you-use (IWYU) is a tool built on top of Clang and LLVM that analyzes `#include` directives in C and C++ source files. For every translation unit it reports headers that are included

but not used, and symbols that are used but whose defining header is only included transitively. Acting on its suggestions keeps the set of includes minimal and explicit.

Installing include-what-you-use `include-what-you-use` is distributed by most Linux distributions, on ubuntu:

```
sudo apt-get install iwyu
```

Make sure the `include-what-you-use` binary is available in your *PATH*.

> **Note**
>
> IWYU is built on Clang, so building with the LLVM toolchain produces the most accurate results.

Run include-what-you-use To run include-what-you-use, *west build* should be called with a `-DZEPHYR_SCA_VARIANT=iwyu` parameter, e.g.

```
west build -b native_sim samples/hello_world -- -DZEPHYR_SCA_VARIANT=iwyu
```

The analysis runs alongside the compilation of each source file, and the suggested include changes are printed to the build output (stderr).

Configuring include-what-you-use include-what-you-use can be controlled using specific options. Refer to the IWYU documentation for the full list of options.

Parameter	Description
`IWYU_OP`	A semicolon separated list of include-what-you-use options. Each option is forwarded to the tool with the required `-Xiwyu` prefix automatically.

These parameters can be passed on the command line, or be set as environment variables.

```
west build -b native_sim samples/hello_world -- -DZEPHYR_SCA_VARIANT=iwyu -
↪DIWYU_OPTS="--no_comments;--verbose=3"
```

Polyspace support

Polyspace® is a commercial static code analysis tool from MathWorks, which is certified for use in highest safety contexts. It can check compliance to coding guidelines like MISRA C and CERT C, find CWEs, detect bugs and calculate code complexity metrics. Optionally, it can run a formal proof to verify the absence of run-time errors like array out of bounds access, overflows, race conditions and more, and thus help achieving memory safety.

Installing The Polyspace tools must be installed and made available in the operating system's or container's PATH variable. Specifically, the path `<polyspace_root>/polyspace/bin` must be on the list.

For installation instructions, see here. To use formal verification (proving the *absence* of defects), you additionally need to install this.

A license file must be available in the installation folder. To request a trial license, visit this page.

Running The code analysis can be triggered through the `west` command by appending the option `-DZEPHYR_SCA_VARIANT=polyspace` to the build, for example:

```
west build -b qemu_x86 samples/hello_world -- -DZEPHYR_SCA_VARIANT=polyspace
```

Reviewing results The identified issues are summarized at the end of the build and printed in the console window, along with the path of the folder containing detailed results.

For an efficient review, the folder should be opened in the Polyspace user interface or uploaded to the web interface and reviewed there.

For programmatic access of the results, e.g., in the CI pipeline, the individual issues are also described in a CSV file in the results folder.

Configuration By default, Polyspace scans for typical programming defects on all C and C++ sources. The following options are available to customize the default behavior:

Option	Effect	Example
`POLYSPACE_ONLY_APP`	If set, only user code is analyzed and Zephyr sources are ignored.	`-DPOLYSPACE_ONLY_APP=1`
`POLYSPACE_OPTIONS`	Provide additional command line flags, e.g., for selection of coding rules. Separate the options and their values by semicolon. For a list of options, see here.	`-DPOLYSPACE_OPTIONS="-misra3;mandatory-required;-checkers;all"`
`POLYSPACE_OPTIONS_`	Command line flags can also be provided in a text file, line by line. Provide the absolute path to the file.	`-DPOLYSPACE_OPTIONS_FILE=/workdir/zephyr/myoptions.txt`
`POLYSPACE_MODE`	Switch between bugfinding and proving mode. Default is bugfinding. See here for more details.	`-DPOLYSPACE_MODE=prove`
`POLYSPACE_PROG_NAM`	Override the name of the analyzed application. Default is board and application name.	`-DPOLYSPACE_PROG_NAME=myapp`
`POLYSPACE_PROG_VER`	Override the version of the analyzed application. Default is taken from git-describe.	`-DPOLYSPACE_PROG_VERSION=v1.0b-28f023`

Sparse support

Sparse is a static code analysis tool. Apart from performing common code analysis tasks it also supports an `address_space` attribute, which allows introduction of distinct address spaces in C code with subsequent verification that pointers to different address spaces do not get confused. Additionally it supports a `force` attribute which should be used to cast pointers between different address spaces. At the moment Zephyr introduces a single custom address space `__cache` used to identify pointers from the cached address range on the Xtensa architecture. This helps identify cases where cached and uncached addresses are confused.

Running with sparse To run a sparse verification build *west build* should be called with a `-DZEPHYR_SCA_VARIANT=sparse` parameter, e.g.

```
west build -d hello -b intel_adsp/cavs25 zephyr/samples/hello_world -- -
↪DZEPHYR_SCA_VARIANT=sparse
```

2.15 Toolchains

Guides on how to set up toolchains for Zephyr development.

2.15.1 Zephyr SDK

The Zephyr Software Development Kit (SDK) contains GNU and LLVM toolchains for each of Zephyr's supported architectures. It also includes additional host tools, such as custom QEMU and OpenOCD.

Use of the Zephyr SDK is highly recommended and may even be required under certain conditions (for example, running tests in QEMU for some architectures).

Supported architectures

The Zephyr SDK supports the following target architectures:

- ARC (32-bit and 64-bit; ARCv1, ARCv2, ARCv3)
- ARM (32-bit and 64-bit; ARMv6, ARMv7, ARMv8; A/R/M Profiles)
- Microblaze (32-bit)
- MIPS (32-bit and 64-bit)
- RISC-V (32-bit and 64-bit; RV32I, RV32E, RV64I)
- RX
- SPARC (32-bit and 64-bit; SPARC V8, SPARC V9)
- x86 (32-bit and 64-bit)
- Xtensa

Installation bundle and variables

The Zephyr SDK bundle supports all major operating systems (Linux, macOS and Windows) and is delivered as a compressed file.

To ease distribution, the SDK is pre-packed in three different variants you can download:

Table 8: SDK Bundle Variants

Variant	Host Tools	Toolchains Included
`gnu`	Yes	GNU (Binutils, GCC, and GDB) for all supported architectures
`llvm`	Yes	LLVM/Clang
`minimal`	Yes	None

The installation process consists of extracting the downloaded bundle file and running the included setup script.

Independently of which bundle you download, during installation you will be prompted for which of the toolchains you want to install, and if it is not found in the local directory where the setup script is being executed it will be downloaded during installation. It is also possible to download and uncompress both the GNU and LLVM SDK bundles into the same directory tree and in that way install both without the need to download either during the setup script execution.

Additional OS-specific instructions are described in the sections below.

If no toolchain is selected, the build system looks for Zephyr SDK and uses the toolchain from there. You can enforce this by setting the environment variable *ZEPHYR_TOOLCHAIN_VARIANT* to `zephyr`.

If you install the Zephyr SDK outside any of the default locations (listed in the operating system specific instructions below) and you want automatic discovery of the Zephyr SDK, then you must register the Zephyr SDK in the CMake package registry by running the setup script. If you decide not to register the Zephyr SDK in the CMake registry, then the *ZEPHYR_SDK_INSTALL_DIR* can be used to point to the Zephyr SDK installation directory.

You can also set *ZEPHYR_SDK_INSTALL_DIR* to point to a directory containing multiple Zephyr SDKs, allowing for automatic toolchain selection. For example, you can set `ZEPHYR_SDK_INSTALL_DIR` to `/company/tools`, where the `company/tools` folder contains the following subfolders:

- `/company/tools/zephyr-sdk-0.13.2`
- `/company/tools/zephyr-sdk-a.b.c`
- `/company/tools/zephyr-sdk-x.y.z`

This allows the Zephyr build system to choose the correct version of the SDK, while allowing multiple Zephyr SDKs to be grouped together at a specific path.

Zephyr SDK version compatibility

In general, the Zephyr SDK version referenced in this page should be considered the recommended version for the corresponding Zephyr version.

For the full list of compatible Zephyr and Zephyr SDK versions, refer to the Zephyr SDK Version Compatibility Matrix.

Zephyr SDK installation

> **Note**
>
> You can change `1.0.1` to another version in the instructions below if needed; the Zephyr SDK Releases page contains all available SDK releases.

> **Note**
>
> The instructions below are for installing using the Zephyr GNU SDK bundle, which includes the GNU toolchains for all supported architectures and the host tools. To install using the Zephyr LLVM SDK bundle, replace the `_gnu` suffix with `_llvm` in the SDK bundle file name.

> **Note**
>
> If you want to uninstall the SDK, you may simply remove the directory where you installed it.

Linux

1. Download and verify the Zephyr SDK bundle:

```
cd ~
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v1.0.1/
↪zephyr-sdk-1.0.1_linux-x86_64_gnu.tar.xz
```

```
wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v1.0.
↪1/sha256.sum | shasum --check --ignore-missing
```

If your host architecture is 64-bit ARM (for example, Raspberry Pi), replace `x86_64` with `aarch64` in order to download the 64-bit ARM Linux SDK.

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-1.0.1_linux-x86_64_gnu.tar.xz
```

> **Note**
>
> It is recommended to extract the Zephyr SDK bundle at one of the following locations:
>
> - `$HOME`
> - `$HOME/.local`
> - `$HOME/.local/opt`
> - `$HOME/bin`
> - `/opt`
> - `/usr/local`
>
> The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `$HOME`, the resulting installation path will be `$HOME/zephyr-sdk-<version>`.

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-1.0.1
./setup.sh
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

4. Install udev rules, which allow you to flash most Zephyr boards as a regular user:

```
sudo cp ~/zephyr-sdk-1.0.1/hosttools/sysroots/x86_64-pokysdk-linux/usr/share/
↪openocd/contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

macOS

1. Download and verify the Zephyr SDK bundle:

```
cd ~
curl -L -O https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v1.
↪0.1/zephyr-sdk-1.0.1_macos-aarch64_gnu.tar.xz
curl -L https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v1.0.1/
↪sha256.sum | shasum --check --ignore-missing
```

2. Extract the Zephyr SDK bundle archive:

```
tar xvf zephyr-sdk-1.0.1_macos-aarch64_gnu.tar.xz
```

> **Note**
>
> It is recommended to extract the Zephyr SDK bundle at one of the following locations:
>
> - `$HOME`
> - `$HOME/.local`
> - `$HOME/.local/opt`
> - `$HOME/bin`
> - `/opt`
> - `/usr/local`
>
> The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `$HOME`, the resulting installation path will be `$HOME/zephyr-sdk-<version>`.

3. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-1.0.1
./setup.sh
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

Windows

1. Open a `cmd.exe` terminal window **as a regular user**
2. Download the Zephyr SDK bundle:

```
cd %HOMEPATH%
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v1.0.1/
↪zephyr-sdk-1.0.1_windows-x86_64_gnu.7z
```

3. Extract the Zephyr SDK bundle archive:

```
7z x zephyr-sdk-1.0.1_windows-x86_64_gnu.7z
```

> **Note**
>
> It is recommended to extract the Zephyr SDK bundle at one of the following locations:
>
> - `%HOMEPATH%`
> - `%PROGRAMFILES%`
>
> The Zephyr SDK bundle archive contains the `zephyr-sdk-<version>` directory and, when extracted under `%HOMEPATH%`, the resulting installation path will be `%HOMEPATH%\zephyr-sdk-<version>`.

4. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-1.0.1
setup.cmd
```

> **Note**
>
> You only need to run the setup script once after extracting the Zephyr SDK bundle.
>
> You must rerun the setup script if you relocate the Zephyr SDK bundle directory after the initial setup.

2.15.2 Arm Compiler 6

1. Download and install a development suite containing the Arm Compiler 6 for your operating system.
2. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `armclang`.
   - Set `ARMCLANG_TOOLCHAIN_PATH` to the toolchain installation directory.
3. The Arm Compiler 6 needs the `ARMLMD_LICENSE_FILE` environment variable to point to your license file or server.

For example:

```
# Linux, macOS, license file:
export ARMLMD_LICENSE_FILE=/<path>/license_armds.dat
# Linux, macOS, license server:
export ARMLMD_LICENSE_FILE=8224@myserver
```

```
# Windows, license file:
set ARMLMD_LICENSE_FILE=c:\<path>\license_armds.dat
# Windows, license server:
set ARMLMD_LICENSE_FILE=8224@myserver
```

1. If the Arm Compiler 6 was installed as part of an Arm Development Studio, then you must set the `ARM_PRODUCT_DEF` to point to the product definition file: See also: Product and toolkit configuration. For example if the Arm Development Studio is installed in: `/opt/armds-2020-1` with a Gold license, then set `ARM_PRODUCT_DEF` to point to `/opt/armds-2020-1/gold.elmap`.

> **Note**
>
> The Arm Compiler 6 uses `armlink` for linking. This is incompatible with Zephyr's linker script template, which works with GNU ld. Zephyr's Arm Compiler 6 support Zephyr's CMake linker script generator, which supports generating scatter files. Basic scatter file support is in place, but there are still areas covered in ld templates which are not fully supported by the CMake linker script generator.
>
> Some Zephyr subsystems or modules may also contain C or assembly code that relies on GNU intrinsics and have not yet been updated to work fully with `armclang`.

2.15.3 Arm Toolchain for Embedded (ATfE)

Arm Toolchain for Embedded (ATfE) is a C and C++ toolchain from Arm based
on the free and open-source LLVM Compiler Infrastructure and the Picolib C library for baremetal targets.

ATfE is fined-tuned with a particular focus on performance for newer
ARM products (post 2024) like 64-bit Arm Architectures (AArch64), or the M-Profile Vector Extension (MVE, a 32-bit Armv8.1-M extension).

Installation

1. Download and install a Arm toolchain for embedded build for your operating system and extract it on your file system.
2. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `host/llvm`.
   - Set `LLVM_TOOLCHAIN_PATH` to the toolchain installation directory.
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `LLVM_TOOLCHAIN_PATH` values may be different on your system):

   Ubuntu

   ```
   echo $ZEPHYR_TOOLCHAIN_VARIANT
   host/llvm
   echo $LLVM_TOOLCHAIN_PATH
   /home/you/Downloads/ATfE
   ```

   macOS

   ```
   echo $ZEPHYR_TOOLCHAIN_VARIANT
   host/llvm
   echo $LLVM_TOOLCHAIN_PATH
   /home/you/Downloads/ATfE
   ```

   Windows

   ```
   > echo %ZEPHYR_TOOLCHAIN_VARIANT%
   host/llvm
   > echo %LLVM_TOOLCHAIN_PATH%
   C:\ATfE
   ```

4. You can also set `ZEPHYR_TOOLCHAIN_VARIANT` and `LLVM_TOOLCHAIN_PATH` as CMake variables when generating a build system for a Zephyr application, like so:

   ```
   west build ... -- -DZEPHYR_TOOLCHAIN_VARIANT=host/llvm -
   ↪DLLVM_TOOLCHAIN_PATH=...
   ```

Toolchain settings

Because LLVM is widely compatible with GNU tools, When builiding with any LLVM toolchain, you have to specify some settings to let the compiler know what tools to use:

Linker

- Set `CONFIG_LLVM_USE_LLD=y` to use LLVM linker.
- Set `CONFIG_LLVM_USE_LD=y` to use the GNU LD linker.

Runtime library

- Set `CONFIG_COMPILER_RT_RTLIB=y` to use LLVM runtime library.
- Set `CONFIG_LIBGCC_RTLIB=y` to use LibGCC runtime library.

```
west build ... -- -DZEPHYR_TOOLCHAIN_VARIANT=host/llvm -DLLVM_TOOLCHAIN_PATH=... -
↪DCONFIG_LLVM_USE_LLD=y -DCONFIG_COMPILER_RT_RTLIB=y
```

2.15.4 Cadence Tensilica Xtensa C/C++ Compiler (XCC)

1. Obtain Tensilica Software Development Toolkit targeting the specific SoC on hand. This usually contains two parts:
   - The Xtensa Xplorer which contains the necessary executables and libraries.
   - A SoC-specific add-on to be installed on top of Xtensa Xplorer.
     - This add-on allows the compiler to generate code for the SoC on hand.
2. Install Xtensa Xplorer and then the SoC add-on.
   - Follow the instruction from Cadence on how to install the SDK.
   - Depending on the SDK, there are two set of compilers:
     - GCC-based compiler: `xt-xcc` and its friends.
     - Clang-based compiler: `xt-clang` and its friends.
3. Make sure you have obtained a license to use the SDK, or has access to a remote licensing server.
4. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `xcc` or `xt-clang`.
   - Set `XTENSA_TOOLCHAIN_PATH` to the toolchain installation directory.
   - There are two ways to specify the SoC ID and the SDK version to use. They are mutually exclusive, and cannot be used together.
     1. When building for a single SoC:
        - Set `XTENSA_CORE` to the SoC ID where application is being targeted.
        - Set `TOOLCHAIN_VER` to the Xtensa SDK version.
     2. When building for multiple SoCs, for each SoC and board combination:
        - Set `XTENSA_CORE_{normalized_board_target}` to the SoC ID where application is being targeted.
        - Set `TOOLCHAIN_VAR_{normalized_board_target}` to the Xtensa SDK version.
5. For example, assuming the SDK is installed in `/opt/xtensa`, and using the SDK for application development on `intel_adsp/ace15_mtpm`, setup the environment using the two above mentioned ways:
   1. Single SoC:

      ```
      # Linux
      export ZEPHYR_TOOLCHAIN_VARIANT=xt-clang
      export XTENSA_TOOLCHAIN_PATH=/opt/xtensa/XtDevTools/install/tools/
      export XTENSA_CORE=ace10_LX7HiFi4_2022_10
      export TOOLCHAIN_VER=RI-2022.10-linux
      ```

   2. Multiple SoCs:

      ```
      # Linux
      export ZEPHYR_TOOLCHAIN_VARIANT=xt-clang
      export XTENSA_TOOLCHAIN_PATH=/opt/xtensa/XtDevTools/install/tools/
      export TOOLCHAIN_VER_intel_adsp_ace15_mtpm=RI-2022.10-linux
      export XTENSA_CORE_intel_adsp_ace15_mtpm=ace10_LX7HiFi4_2022_10
      ```

6. To use Clang-based compiler:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `xt-clang`.

- Note that the Clang-based compiler may contain an old LLVM bug which results in the following error:

```
/tmp/file.s: Assembler messages:
/tmp/file.s:20: Error: file number 1 already allocated
clang-3.9: error: Xtensa-as command failed with exit code 1
```

  If this happens, set `XCC_NO_G_FLAG` to `1`.

  – For example:

```
# Linux
export XCC_NO_G_FLAG=1
```

2.15.5 DesignWare ARC MetaWare Development Toolkit (MWDT)

1. You need to have ARC MWDT installed on your host.
2. You need to have *Zephyr SDK* installed on your host.

> **Note**
>
> A Zephyr SDK is used as a source of tools like device tree compiler (DTC), QEMU, etc... Even though ARC MWDT toolchain is used for Zephyr RTOS build, still the GNU preprocessor & GNU objcopy might be used for some steps like device tree preprocessing and `.bin` file generation. We used Zephyr SDK as a source of these ARC GNU tools as well. To setup ARC GNU toolchain please use SDK Bundle (Full or Minimal) instead of manual installation of separate tarballs. It installs and registers toolchain and host tools in the system, that allows you to avoid toolchain related issues while building Zephyr.

3. *Set these environment variables*:
   - Set *`ZEPHYR_TOOLCHAIN_VARIANT`* to `arcmwdt`.
   - Set `ARCMWDT_TOOLCHAIN_PATH` to the toolchain installation directory. MWDT installation provides `METAWARE_ROOT` so simply set `ARCMWDT_TOOLCHAIN_PATH` to `$METAWARE_ROOT/../` (Linux) or `%METAWARE_ROOT%\..\` (Windows).

> **Tip**
>
> If you have only one ARC MWDT toolchain version installed on your machine you may skip setting `ARCMWDT_TOOLCHAIN_PATH` - it would be detected automatically.

4. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `ARCMWDT_TOOLCHAIN_PATH` values may be different on your system):

```
# Linux:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
arcmwdt
$ echo $ARCMWDT_TOOLCHAIN_PATH
/home/you/ARC/MWDT_2023.03/

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
arcmwdt
> echo %ARCMWDT_TOOLCHAIN_PATH%
C:\ARC\MWDT_2023.03\
```

2.15.6 GNU Arm Embedded

1. Download and install a GNU Arm Embedded build for your operating system and extract it on your file system.

> **Note**
>
> On Windows, we'll assume for this guide that you install into the directory `C:\gnu_arm_embedded`. You can also choose the default installation path used by the ARM GCC installer, in which case you will need to adjust the path accordingly in the guide below.

> **Warning**
>
> On macOS Catalina or later you might need to *change a security policy* for the toolchain to be able to run from the terminal.

2. *Set these environment variables*:
   - Set *`ZEPHYR_TOOLCHAIN_VARIANT`* to `gnuarmemb`.
   - Set `GNUARMEMB_TOOLCHAIN_PATH` to the toolchain installation directory.
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `GNUARMEMB_TOOLCHAIN_PATH` values may be different on your system):

```
# Linux, macOS:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
gnuarmemb
$ echo $GNUARMEMB_TOOLCHAIN_PATH
/home/you/Downloads/gnu_arm_embedded

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
gnuarmemb
> echo %GNUARMEMB_TOOLCHAIN_PATH%
C:\gnu_arm_embedded
```

> **Warning**
>
> On macOS, if you are having trouble with the suggested procedure, there is an unofficial package on brew that might help you. Run `brew install gcc-arm-embedded` and configure the variables
>
> - Set *`ZEPHYR_TOOLCHAIN_VARIANT`* to `gnuarmemb`.
> - Set `GNUARMEMB_TOOLCHAIN_PATH` to the brew installation directory (something like `/usr/local`)

2.15.7 Qualcomm Hexagon LLVM Toolchain

1. Download a pre-built Hexagon LLVM cross-toolchain from the toolchain_for_hexagon releases page and extract it, for example to `/opt/hexagon-toolchain`. The install directory is the one holding `bin/clang`.

   Use a release built from LLVM 23 or newer. Earlier releases miscompile the `~BIT(n)`. The fix landed in #205489.

2. Set *ZEPHYR_TOOLCHAIN_VARIANT* to hexagon and *HEXAGON_TOOLCHAIN_PATH* to that directory:

```
export ZEPHYR_TOOLCHAIN_VARIANT=hexagon
export HEXAGON_TOOLCHAIN_PATH=/opt/hexagon-toolchain
```

HEXAGON_TOOLCHAIN_PATH

Install directory of the Hexagon LLVM cross-toolchain.

The toolchain is an ordinary LLVM installation, so this variant is the *Host Toolchains* `llvm` variant driving a different compiler. It has a path variable of its own because the Hexagon clang registers only the Hexagon target and so cannot build host-compiled targets such as native_sim; an environment that covers both has to name the two LLVM installations independently, as Zephyr's CI does when a single Twister run spans both kinds of platform.

Building a Hexagon target with `ZEPHYR_TOOLCHAIN_VARIANT=host/llvm` and `LLVM_TOOLCHAIN_PATH` is equivalent and remains supported, as long as the installation it names is the Hexagon one.

2.15.8 IAR Arm Toolchain

1. Download and install a release v9.70 or newer of IAR Arm Toolchain on your host (IAR Embedded Workbench or IAR Build Tools, perpetual or subscription licensing)
2. Make sure you have *Zephyr SDK* installed on your host.
3. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `iar`.
   - Set `IAR_TOOLCHAIN_PATH` to the toolchain installation directory.
4. The cloud licensed variant of the IAR Toolchain needs the `IAR_LMS_BEARER_TOKEN` environment variable to be set to a valid `license bearer token` (subscription licensing).

For example:

```
# Linux (default installation path):
export IAR_TOOLCHAIN_PATH=/opt/iar/cxarm-<version>/arm
export ZEPHYR_TOOLCHAIN_VARIANT=iar
export IAR_LMS_BEARER_TOKEN="<BEARER-TOKEN>"
```

```
# Windows:
set IAR_TOOLCHAIN_PATH=c:\<path>\cxarm-<version>\arm
set ZEPHYR_TOOLCHAIN_VARIANT=iar
set IAR_LMS_BEARER_TOKEN="<BEARER-TOKEN>"
```

Note

Known limitations:

- The IAR Toolchain uses `ilink` for linking and depends on Zephyr's CMAKE_LINKER_GENERATOR. `ilink` is incompatible with Zephyr's linker script template, which works with GNU ld.
- The GNU Assembler distributed with the Zephyr SDK is used for `.S-files`.
- C library support for `Minimal libc` only. C++ is not supported.
- Some Zephyr subsystems or modules may contain C or assembly code that relies on GNU intrinsics and have not yet been updated to work fully with `iar`.
- TrustedFirmware is not supported

2.15.9 Intel oneAPI Toolkit

1. Download Intel oneAPI Base Toolkit
2. Assuming the toolkit is installed in `/opt/intel/oneApi`, set environment using:

```
# Linux, macOS:
export ONEAPI_TOOLCHAIN_PATH=/opt/intel/oneapi
source $ONEAPI_TOOLCHAIN_PATH/compiler/latest/env/vars.sh

# Windows:
> set ONEAPI_TOOLCHAIN_PATH=C:\Users\Intel\oneapi
```

   To setup the complete oneApi environment, use:

```
source  /opt/intel/oneapi/setvars.sh
```

   The above will also change the python environment to the one used by the toolchain and might conflict with what Zephyr uses.

3. Set *ZEPHYR_TOOLCHAIN_VARIANT* to `oneApi`.

2.15.10 MPLAB XC32

1. Download and install MPLAB XC32 Compiler for your operating system.
2. *Set these environment variables*:
   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `xc32`.
   - Set `XC32_TOOLCHAIN_PATH` to the toolchain installation directory.
3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `XC32_TOOLCHAIN_PATH` values may be different on your system):

```
# Linux, macOS:
$ echo $ZEPHYR_TOOLCHAIN_VARIANT
xc32
$ echo $XC32_TOOLCHAIN_PATH
/opt/microchip/xc32/v5.00

# Windows:
> echo %ZEPHYR_TOOLCHAIN_VARIANT%
xc32
> echo %XC32_TOOLCHAIN_PATH%
C:\Microchip\xc32\v5.00
```

4. For Microchip SoCs, set `XC_PACK_DIR` to the root directory containing installed Microchip Device Family Packs (DFPs).

   Note

   Zephyr uses `XC_PACK_DIR` to find the DFP matching the selected SoC and to pass the required `-mdfp` and `-mprocessor` flag to the XC32 toolchain.

   For example:

```
# Linux, macOS:
$ export XC_PACK_DIR=/home/path/.packs/microchip
```

(continues on next page)

(continued from previous page)

```
# Windows:
> set XC_PACK_DIR=C:\Users\path\.mchp_packs\Microchip
```

2.15.11 Host Toolchains

In some specific configurations, like when building for non-MCU x86 targets on a Linux host, you may be able to reuse the native development tools provided by your operating system.

To use your host gcc, set the *ZEPHYR_TOOLCHAIN_VARIANT environment variable* to `host/gnu`. To use clang, set *ZEPHYR_TOOLCHAIN_VARIANT* to `host/llvm`.

2.15.12 Other Cross Compilers

This toolchain variant is borrowed from the Linux kernel build system's mechanism of using a `CROSS_COMPILE` environment variable to set up a GNU-based cross toolchain.

Examples of such "other cross compilers" are cross toolchains that your Linux distribution packaged, that you compiled on your own, or that you downloaded from the net. Unlike toolchains specifically listed in *Toolchains*, the Zephyr build system may not have been tested with them, and doesn't officially support them. (Nonetheless, the toolchain set-up mechanism itself is supported.)

Follow these steps to use one of these toolchains.

1. Install a cross compiler suitable for your host and target systems.

   For example, you might install the `gcc-arm-none-eabi` package on Debian-based Linux systems, or `arm-none-eabi-newlib` on Fedora or Red Hat:

   ```
   # On Debian or Ubuntu
   sudo apt-get install gcc-arm-none-eabi
   # On Fedora or Red Hat
   sudo dnf install arm-none-eabi-newlib
   ```

2. *Set these environment variables*:

   - Set *ZEPHYR_TOOLCHAIN_VARIANT* to `cross-compile`.
   - Set `CROSS_COMPILE` to the common path prefix which your toolchain's binaries have, e.g. the path to the directory containing the compiler binaries plus the target triplet and trailing dash.

3. To check that you have set these variables correctly in your current environment, follow these example shell sessions (the `CROSS_COMPILE` value may be different on your system):

   ```
   # Linux, macOS:
   $ echo $ZEPHYR_TOOLCHAIN_VARIANT
   cross-compile
   $ echo $CROSS_COMPILE
   /usr/bin/arm-none-eabi-
   ```

   You can also set `CROSS_COMPILE` as a CMake variable.

When using this option, all of your toolchain binaries must reside in the same directory and have a common file name prefix. The `CROSS_COMPILE` variable is set to the directory concatenated with the file name prefix. In the Debian example above, the `gcc-arm-none-eabi` package installs binaries such as `arm-none-eabi-gcc` and `arm-none-eabi-ld` in directory `/usr/bin/`, so the common prefix is `/usr/bin/arm-none-eabi-` (including the trailing dash, `-`). If your toolchain is installed in `/opt/mytoolchain/bin` with binary names based on target triplet `myarch-none-elf`, `CROSS_COMPILE` would be set to `/opt/mytoolchain/bin/myarch-none-elf-`.

2.15.13 Custom CMake Toolchains

To use a custom toolchain defined in an external CMake file, *set these environment variables*:

- Set *ZEPHYR_TOOLCHAIN_VARIANT* to your toolchain's name
- Set TOOLCHAIN_ROOT to the path to the directory containing your toolchain's CMake configuration files.

Zephyr will then include the toolchain cmake files located in the TOOLCHAIN_ROOT directory:

- cmake/toolchain/<toolchain name>/generic.cmake: configures the toolchain for "generic" use, which mostly means running the C preprocessor on the generated *Devicetree* file.
- cmake/toolchain/<toolchain name>/target.cmake: configures the toolchain for "target" use, i.e. building Zephyr and your application's source code.

Here <toolchain name> is the same as the name provided in *ZEPHYR_TOOLCHAIN_VARIANT* See the zephyr files cmake/modules/FindHostTools.cmake and cmake/modules/FindTargetTools.cmake for more details on what your generic.cmake and target.cmake files should contain.

You can also set ZEPHYR_TOOLCHAIN_VARIANT and TOOLCHAIN_ROOT as CMake variables when generating a build system for a Zephyr application, like so:

```
west build ... -- -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=...
```

```
cmake -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=...
```

If you do this, -C <initial-cache> cmake option may useful. If you save your **ZEPHYR_TOOLCHAIN_VARIANT**, **TOOLCHAIN_ROOT**, and other settings in a file named my-toolchain.cmake, you can then invoke cmake as cmake -C my-toolchain.cmake ... to save typing.

Zephyr includes include/zephyr/toolchain.h which again includes a toolchain specific header based on the compiler identifier, such as __llvm__ or __GNUC__. Some custom compilers identify themselves as the compiler on which they are based, for example llvm which then gets the toolchain/llvm.h included. This included file may though not be right for the custom toolchain. In order to solve this, and thus to get the include/other.h included instead, add the set(TOOLCHAIN_USE_CUSTOM 1) cmake line to the generic.cmake and/or target.cmake files located under <TOOLCHAIN_ROOT>/cmake/toolchain/<toolchain name>/.

When **TOOLCHAIN_USE_CUSTOM** is set, the other.h must be available out-of-tree and it must include the correct header for the custom toolchain. A good location for the other.h header file, would be a directory under the directory specified in TOOLCHAIN_ROOT as include/zephyr/toolchain. To get the toolchain header included in zephyr's build, the **USERINCLUDE** can be set to point to the include directory, as shown here:

```
west build -- -DZEPHYR_TOOLCHAIN_VARIANT=... -DTOOLCHAIN_ROOT=... -DUSERINCLUDE=...
```

2.15.14 Embedded Linker (ELD)

ELD is Qualcomm's open-source, LLVM-based, GNU-compatible linker. It can be used as an alternative to LLD or GNU ld when building Zephyr with an LLVM toolchain (for example *Arm Toolchain for Embedded* or a host clang). See the ELD user guide for details on ELD itself.

ELD is just a linker (ld.eld); a separate C/C++ toolchain is still required to compile sources.

Installation

There are three ways to obtain ELD:

1. **Nightly binary release.** Prebuilt ld.eld binaries are published on the ELD releases page.

2. **Build from source.** Build ELD against an LLVM tree as described in the ELD README; an integrated `llvm-project` build produces `ld.eld` under the build tree's `bin/` directory.
3. **Full toolchain via cpullvm.** The cpullvm toolchain is a complete LLVM toolchain for Arm, AArch64 and RISC-V embedded targets that bundles both `ld.eld` and `ld.lld` along with clang, the runtime libraries and headers. Prebuilt cpullvm archives are published on its cpullvm releases page for Linux and Windows, and cpullvm 22.1.1 has been tested with Zephyr in the ELD CI.

Options 1 and 2 only provide `ld.eld`; a separate LLVM toolchain (clang, runtime libraries and headers) is still needed to build Zephyr.

Verify the installation:

```
$ ld.eld --version
eld 22.0 (GNU Compatible linker)
```

ELD 22.0 or later is required.

Usage

ELD is selected through the `host/llvm` toolchain variant. Point `LLVM_TOOLCHAIN_PATH` at the LLVM toolchain (for example a cpullvm installation, or another toolchain with `ld.eld` on its `bin/` directory or *PATH*), then enable ELD by setting `CONFIG_LLVM_USE_ELD` to `y`.

For example:

```
west build ... -- -DZEPHYR_TOOLCHAIN_VARIANT=host/llvm \
                  -DLLVM_TOOLCHAIN_PATH=<path-to-llvm-toolchain> \
                  -DCONFIG_LLVM_USE_ELD=y
```

See *Arm Toolchain for Embedded (ATfE)* or *Zephyr SDK* for more on configuring an LLVM toolchain.

2.16 Tools and IDEs

2.16.1 CLion

> **Note**
>
> This guide describes how to set up, build, and debug Zephyr's sample application in CLion, using the IDE's CMake integration. This approach is no longer optimal.
>
> CLion now features native Zephyr West integration which provides an easier and more intuitive way to open, build, and run/debug Zephyr projects. This guide will be updated soon, but is still valid if you prefer to use CMake.

CLion is a cross-platform C/C++ IDE that supports multi-threaded RTOS debugging.

This guide describes the process of setting up, building, and debugging Zephyr's multi-thread-blinky sample in CLion.

The instructions have been tested on Windows. In terms of the CLion workflow, the steps would be the same for macOS and Linux, but make sure to select the correct environment file and to adjust the paths.

Get CLion

Download CLion and install it.

Initialize a new workspace

This guide gives details on how to build and debug the multi-thread-blinky sample application, but the instructions would be similar for any Zephyr project and *workspace layout*.

Before you start, make sure you have a working Zephyr development environment, as per the instructions in the *Getting Started Guide*.

Open the project in CLion

1. In CLion, click *Open* on the Welcome screen or select *File* ▸ *Open* from the main menu.
2. Navigate to your Zephyr workspace (i.e.the `zephyrproject` folder in your HOME directory if you have followed the Getting Started instructions), then select `zephyr/samples/basic/threads` or another sample project folder.

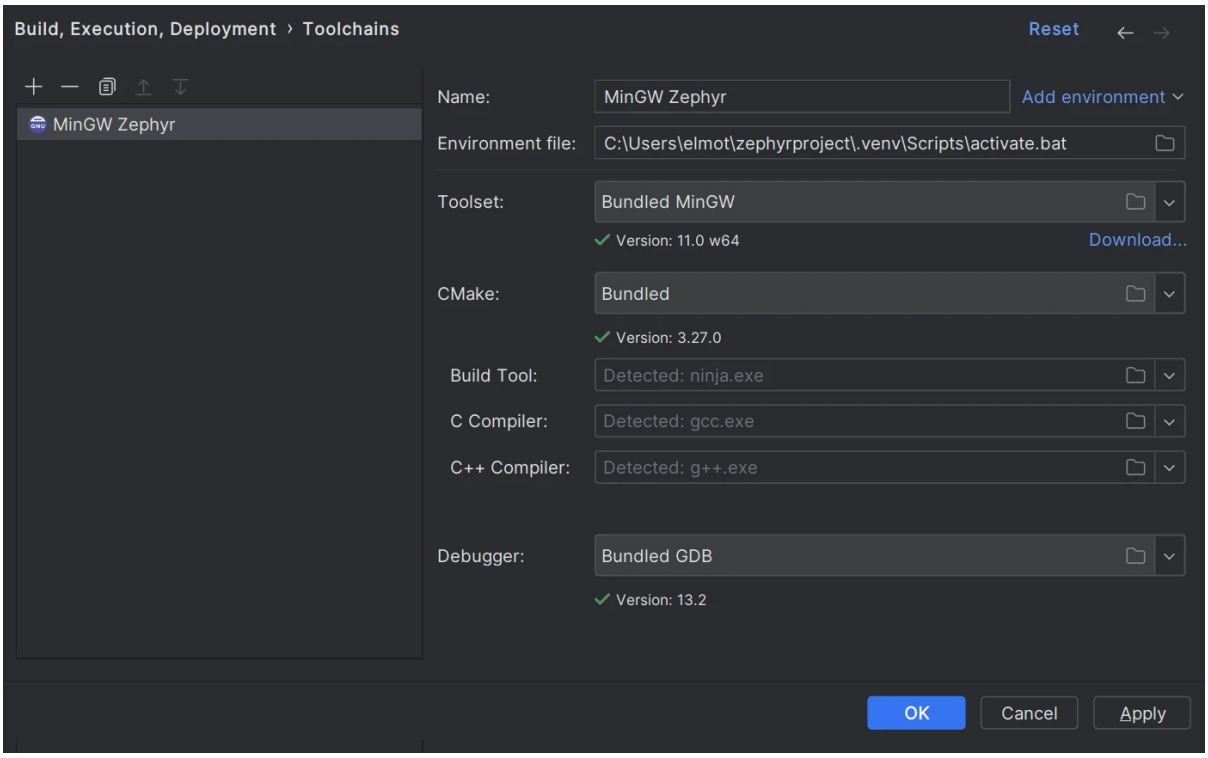
   Click *OK*.
3. If prompted, click *Trust Project*.

   See the Project security section in CLion web help for more information on project security.

Configure the toolchain and CMake profile

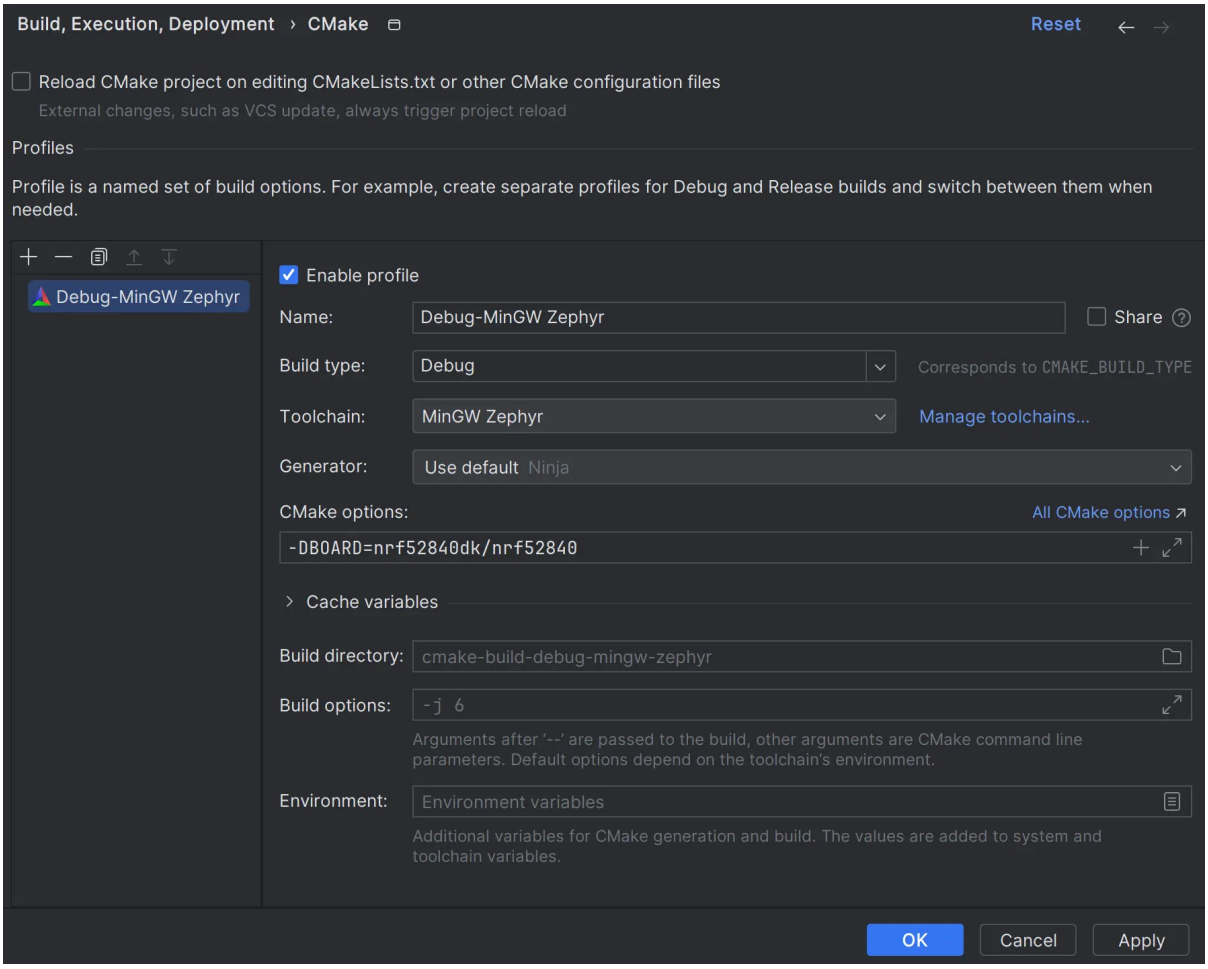
CLion will open the *Open Project Wizard* with the CMake profile settings. If that does not happen, go to *Settings* ▸ *Build, Execution, Deployment* ▸ *CMake*.

1. Click *Manage Toolchains* next to the *Toolchain* field. This will open the *Toolchain* settings dialog.
2. We recommend that you use the *Bundled MinGW* toolchain with default settings on Windows, or the *System* (default) toolchain on Unix machines.
3. Click *Add environment* ▸ *From file* and select `..\.venv\Scripts\activate.bat`.



   Click *Apply* to save the changes.
4. Back in the CMake profile settings dialog, specify your board in the *CMake options* field. For example:

```
-DBOARD=nrf52840dk/nrf52840
```



5. Click *Apply* to save the changes.

   CMake load should finish successfully.

Configure Zephyr parameters for debug

1. In the configuration switcher on the top right, select *guiconfig* and click the hammer icon.
2. Use the GUI application to set the following flags:

```
DEBUG_THREAD_INFO
THREAD_RUNTIME_STATS
DEBUG_OPTIMIZATIONS
```

Build the project

In the configuration switcher, select **zephyr_final** and click the hammer icon.

Note that other CMake targets like `puncover` or `hardenconfig` can also be called at this point.

Enable RTOS integration

1. Go to *Settings* ▸ *Build, Execution, Deployment* ▸ *Embedded Development* ▸ *RTOS Integration*.
2. Set the *Enable RTOS Integration* checkbox.

This option enables Zephyr tasks view during debugging. See Multi-threaded RTOS debug in CLion web help for more information.

You can leave the option set to *Auto*. CLion will detect Zephyr automatically.

Create an Embedded GDB Server configuration

In order to debug a Zephyr application in CLion, you need to create a run/debug configuration out of the Embedded GDB Server template.

Instructions below show the case of a Nordic Semiconductor board and a Segger J-Link debug probe. If your setup is different, make sure to adjust the configuration settings accordingly.

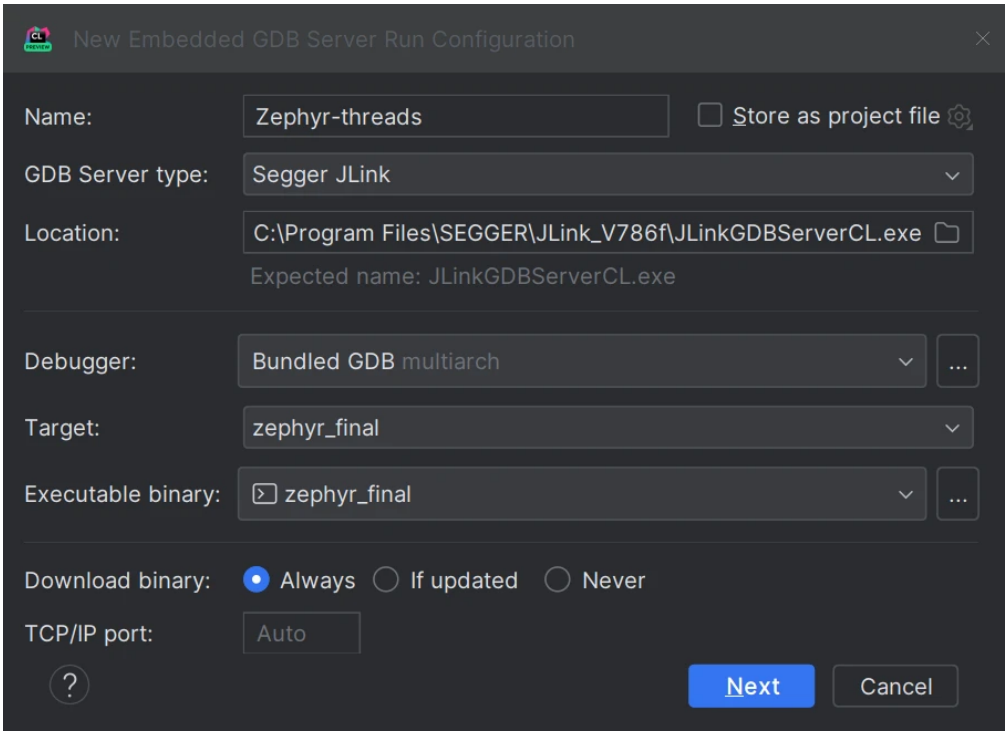
1. Select *Run* ▸ *New Embedded Configuration* from the main menu.
2. Configure the settings:

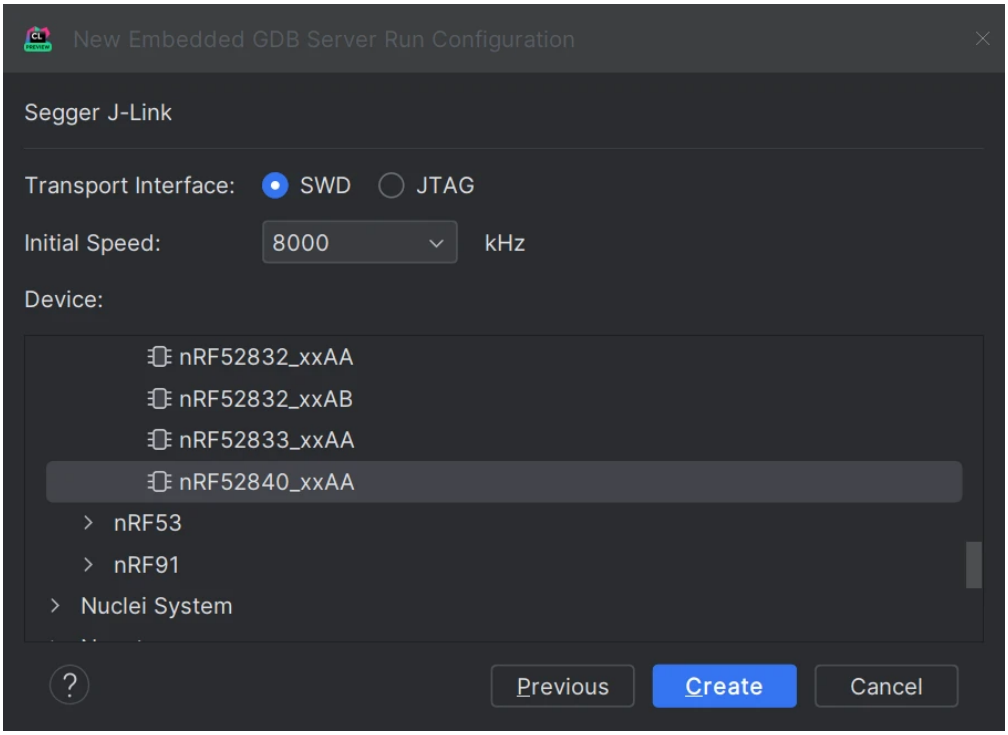
Option	Value
Name (optional)	Zephyr-threads
GDB Server Type	Segger JLink
Location	The path to `JLinkGDBServerCL.exe` on Windows or the `JLinkGDBServer` binary on macOS/Linux.
Debugger	Bundled GDB **Note** For non-ARM and non-x86 architectures, use a GDB executable from Zephyr SDK. Make sure to pick a version with Python support (for example, **riscv64-zephyr-elf-gdb-py**) and check that Python is present in the system `PATH`.
Target	zephyr-final
Executable binary	zephyr-final
Download binary	Always
TCP/IP port	Auto

3. Click *Next* to set the Segger J-Link parameters.
4. Click *Create* when ready.

Start debugging

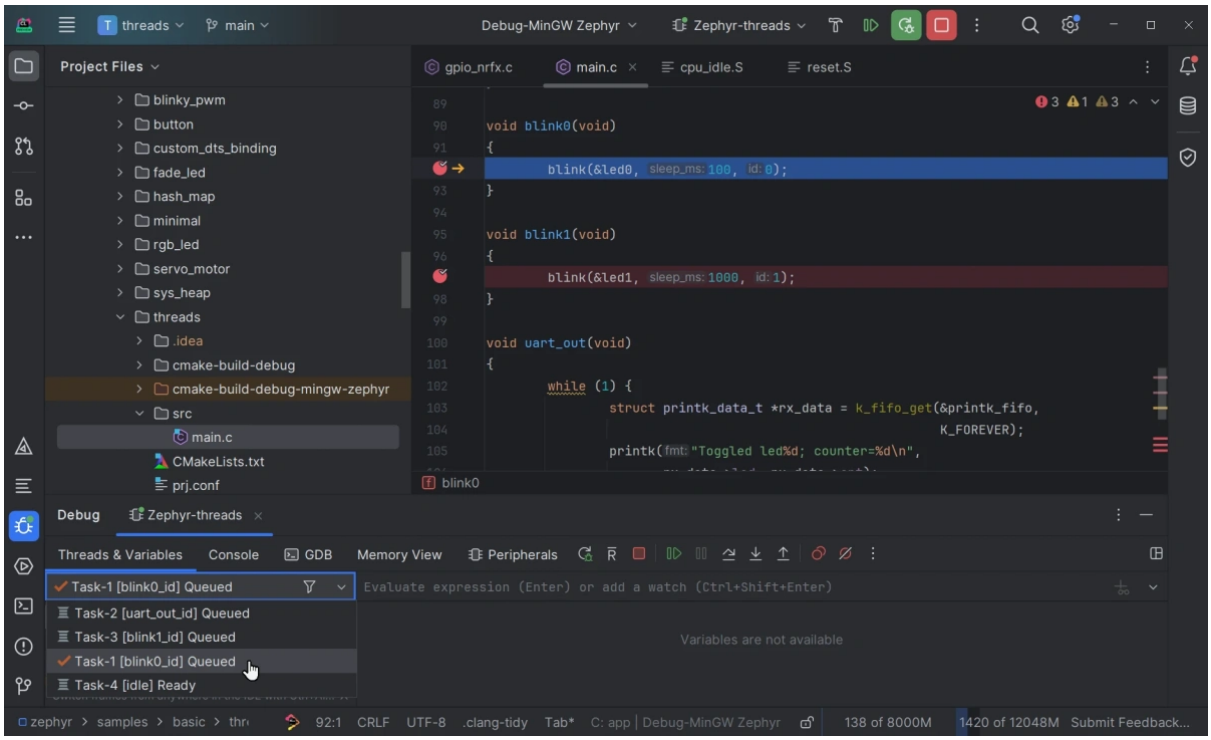
1. Place breakpoints by clicking in the left gutter next to the code lines.


New Embedded GDB Server Run Configuration
Name:
Zephyr-threads
Store as project file
GDB Server type:
Segger JLink
Location:
C:\Program Files\SEGGER\JLink_V786f\JLinkGDBServerCL.exe
Expected name: JLinkGDBServerCL.exe
Debugger:
Bundled GDB multiarch
Target:
zephyr_final
Executable binary:
zephyr_final
Download binary:
Always
If updated
Never
TCP/IP port:
Auto
Next
Cancel


New Embedded GDB Server Run Configuration
Segger J-Link
Transport Interface:
SWD
JTAG
Initial Speed:
8000
kHz
Device:
nRF52832_xxAA
nRF52832_xxAB
nRF52833_xxAA
nRF52840_xxAA
nRF53
nRF91
Nuclei System
Previous
Create
Cancel

2. Make sure that **Zephyr-threads** is selected in the configuration switcher and click the bug icon or press `Ctrl+D`.
3. When a breakpoint is hit, CLion opens the Debug tool window.

   Zephyr tasks are listed in the *Threads & Variables* pane. You can switch between them and inspect the variables for each task.



Refer to CLion web help for detailed description of the IDE debug capabilities.

2.16.2 Coccinelle

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

> **Note**
>
> Linux and macOS development environments are supported, but not Windows.

Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and `coccicheck` have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE

- Arch Linux
- NetBSD
- FreeBSD

Some distribution packages are obsolete and it is recommended to use the latest version released from the Coccinelle homepage at https://coccinelle.lip6.fr/

Or from Github at:

https://github.com/coccinelle/coccinelle

Once you have it, run the following commands:

```
./autogen
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

More detailed installation instructions to build from source can be found at:

https://github.com/coccinelle/coccinelle/blob/master/install.txt

Supplemental documentation

For Semantic Patch Language(SmPL) grammar documentation refer to:

https://coccinelle.gitlabpages.inria.fr/website/documentation.html

Using Coccinelle on Zephyr

`coccicheck` checker is the front-end to the Coccinelle infrastructure and has various modes:

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting `--mode=<mode>` or `-m=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style.Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is `report`.

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

Examples

To make a report for every semantic patch, run the following command:

```
./scripts/coccicheck --mode=report
```

To produce patches, run:

```
./scripts/coccicheck --mode=patch
```

The `coccicheck` target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire source code tree.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set `--verbose=1` option, for example:

```
./scripts/coccicheck --mode=report --verbose=1
```

Coccinelle parallelization

By default, `coccicheck` tries to run as parallel as possible. To change the parallelism, set the `--jobs=<number>` option. For example, to run across 4 CPUs:

```
./scripts/coccicheck --mode=report --jobs=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization, if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled `coccicheck` will enable dynamic load balancing by using `--chunksize 1` argument, this ensures we keep feeding threads with work one by one, so that we avoid the situation where most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, the return value of the `coccicheck` command captures this return value.

Using Coccinelle with a single semantic patch

The option `--cocci` can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
./scripts/coccicheck --mode=report --cocci=<example.cocci>
```

or:

```
./scripts/coccicheck --mode=report --cocci=./path/to/<example.cocci>
```

Controlling which files are processed by Coccinelle

By default the entire source tree is checked.

To apply Coccinelle to a specific directory, pass the path of specific directory as an argument.

For example, to check `drivers/usb/` one may write:

```
./scripts/coccicheck --mode=patch drivers/usb/
```

The `report` mode is the default. You can select another one with the `--mode=<mode>` option explained above.

Debugging Coccinelle SmPL patches

Using `coccicheck` is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using verbose option, you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr, by default stderr is redirected to /dev/null, if you'd like to capture stderr you can specify the `--debug=file.err` option to `coccicheck`. For instance:

```
rm -f cocci.err
./scripts/coccicheck --mode=patch --debug=cocci.err
cat cocci.err
```

Debugging support is only supported when using Coccinelle >= 1.0.2.

Additional Flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
./scripts/coccicheck --sp-flag="--use-glimpse"
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel, coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
./scripts/coccicheck --sp-flag="--use-idutils"
```

Alternatively you can specify the database filename explicitly, for instance:

```
./scripts/coccicheck --sp-flag="--use-idutils /full-path/to/ID"
```

Sometimes coccinelle doesn't recognize or parse complex macro variables due to insufficient definition. Therefore, to make it parsable we explicitly provide the prototype of the complex macro using the `---macro-file-builtins <headerfile.h>` flag.

The `<headerfile.h>` should contain the complete prototype of the complex macro from which spatch engine can extract the type information required during transformation.

For example:

`Z_SYSCALL_HANDLER` is not recognized by coccinelle. Therefore, we put its prototype in a header file, say for example `mymacros.h`.

```
$ cat mymacros.h
#define Z_SYSCALL_HANDLER int xxx
```

Now we pass the header file `mymacros.h` during transformation:

```
./scripts/coccicheck --sp-flag="---macro-file-builtins mymacros.h"
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

The cocci script should have the following properties:

- The script **must** have `report` mode.
- The first few lines should state the purpose of the script using `///` comments . Usually, this message would be used as the commit log when proposing a patch based on the script.

Example

```
/// Use ARRAY_SIZE instead of dividing sizeof array with sizeof an element
```

- A more detailed information about the script with exceptional cases or false positives (if any) can be listed using `//#` comments.

Example

```
//# This makes an effort to find cases where ARRAY_SIZE can be used such as
//# where there is a division of sizeof the array by the sizeof its first
//# element or by any indexed element or the element type. It replaces the
//# division of the two sizeofs by ARRAY_SIZE.
```

- Confidence: It is a property defined to specify the accuracy level of the script. It can be either `High`, `Moderate` or `Low` depending upon the number of false positives observed.

Example

```
// Confidence: High
```

- Virtual rules: These are required to support the various modes framed in the script. The virtual rule specified in the script should have the corresponding mode handling rule.

Example

```
virtual context

@depends on context@
type T;
T[] E;
@@
(
* (sizeof(E)/sizeof(*E))
|
* (sizeof(E)/sizeof(E[...]))
|
* (sizeof(E)/sizeof(T))
)
```

Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

Example Running:

```
./scripts/coccicheck --mode=report --cocci=scripts/coccinelle/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@r depends on (org || report)@
type T;
T[] E;
position p;
@@
(
(sizeof(E)@p /sizeof(*E))
|
(sizeof(E)@p /sizeof(E[...]))
|
(sizeof(E)@p /sizeof(T))
)

@script:python depends on report@
p << r.p;
@@

msg="WARNING: Use ARRAY_SIZE"
coccilib.report.print_report(p[0], msg)

</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
ext/hal/nxp/mcux/drivers/lpc/fsl_wwdt.c:66:49-50: WARNING: Use ARRAY_SIZE
ext/hal/nxp/mcux/drivers/lpc/fsl_ctimer.c:74:53-54: WARNING: Use ARRAY_SIZE
ext/hal/nxp/mcux/drivers/imx/fsl_dcp.c:944:45-46: WARNING: Use ARRAY_SIZE
```

Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

Example Running:

```
./scripts/coccicheck --mode=patch --cocci=scripts/coccinelle/misc/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@depends on patch@
type T;
T[] E;
```

(continues on next page)

(continued from previous page)

```
@@
(
- (sizeof(E)/sizeof(*E))
+ ARRAY_SIZE(E)
|
- (sizeof(E)/sizeof(E[...]))
+ ARRAY_SIZE(E)
|
- (sizeof(E)/sizeof(T))
+ ARRAY_SIZE(E)
)

</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/ext/lib/encoding/tinycbor/src/cborvalidation.c␣
↪b/ext/lib/encoding/tinycbor/src/cborvalidation.c
--- a/ext/lib/encoding/tinycbor/src/cborvalidation.c
+++ b/ext/lib/encoding/tinycbor/src/cborvalidation.c
@@ -325,7 +325,7 @@ static inline CborError validate_number(
static inline CborError validate_tag(CborValue *it, CborTag tag, int flags, int␣
↪recursionLeft)
{
  CborType type = cbor_value_get_type(it);
-    const size_t knownTagCount = sizeof(knownTagData) / sizeof(knownTagData[0]);
+    const size_t knownTagCount = ARRAY_SIZE(knownTagData);
   const struct KnownTagData *tagData = knownTagData;
   const struct KnownTagData * const knownTagDataEnd = knownTagData +␣
↪knownTagCount;
```

Detailed description of the `context` mode

`context` highlights lines of interest and their context in a diff-like style.

> **Note**
>
> The diff-like output generated is NOT an applicable patch. The intent of the `context` mode is to highlight the important lines (annotated with minus, `-`) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

Example Running:

```
./scripts/coccicheck --mode=context --cocci=scripts/coccinelle/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@depends on context@
type T;
T[] E;
@@
(
* (sizeof(E)/sizeof(*E))
```

(continues on next page)

(continued from previous page)

```
|
* (sizeof(E)/sizeof(E[...]))
|
* (sizeof(E)/sizeof(T))
)

</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p ext/lib/encoding/tinycbor/src/cborvalidation.c /
↪tmp/nothing/ext/lib/encoding/tinycbor/src/cborvalidation.c
--- ext/lib/encoding/tinycbor/src/cborvalidation.c
+++ /tmp/nothing/ext/lib/encoding/tinycbor/src/cborvalidation.c
@@ -325,7 +325,6 @@ static inline CborError validate_number(
static inline CborError validate_tag(CborValue *it, CborTag tag, int flags, int␣
↪recursionLeft)
{
  CborType type = cbor_value_get_type(it);
-    const size_t knownTagCount = sizeof(knownTagData) / sizeof(knownTagData[0]);
   const struct KnownTagData *tagData = knownTagData;
   const struct KnownTagData * const knownTagDataEnd = knownTagData +␣
↪knownTagCount;
```

Detailed description of the `org` mode

`org` generates a report in the Org mode format of Emacs.

Example Running:

```
./scripts/coccicheck --mode=org --cocci=scripts/coccinelle/misc/array_size.cocci
```

will execute the following part of the SmPL script:

```
<smpl>

@r depends on (org || report)@
type T;
T[] E;
position p;
@@
(
(sizeof(E)@p /sizeof(*E))
|
(sizeof(E)@p /sizeof(E[...]))
|
(sizeof(E)@p /sizeof(T))
)

@script:python depends on org@
p << r.p;
@@
coccilib.org.print_todo(p[0], "WARNING should use ARRAY_SIZE")

</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO␣
↪[[view:ext/lib/encoding/tinycbor/src/cborvalidation.c::face=ovl-face1::linb=328::colb=52::cole=
↪should use ARRAY_SIZE]]
```

Coccinelle Mailing List

Subscribe to the coccinelle mailing list:

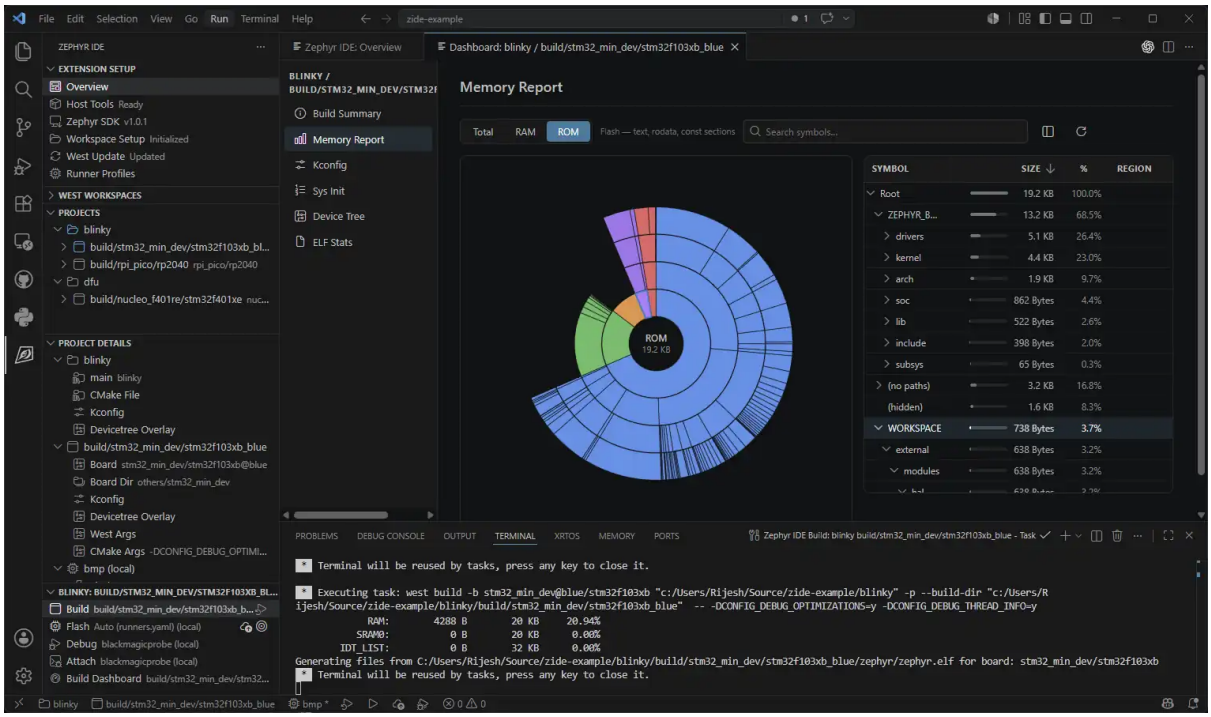
- https://systeme.lip6.fr/mailman/listinfo/cocci

Archives:

- https://lore.kernel.org/cocci/
- https://systeme.lip6.fr/pipermail/cocci/

2.16.3 IDE for Zephyr (VS Code extension)

IDE for Zephyr is a Visual Studio Code (VS Code) extension for Zephyr RTOS development. It supports **host tool management**, **west workspace setup**, **SDK management**, **project creation**, **build/flash**, and **debugging**.



Key features

- Integrate with Cortex-Debug for ST-Link, J-Link, OpenOCD, Black Magic Probe, and other probes via the built-in `zephyr-ide-cortex` and `zephyr-ide-west` debugger types that auto-resolve ELF, GDB, and runner paths
- Integrate with clangd or C/C++ for IntelliSense
- Explore memory usage, Kconfig, and devicetree from the Build Dashboard; view ROM and RAM as a sunburst chart without leaving VS Code
- Edit Kconfig options interactively with the built-in editor
- Add, run, and reconfigure Twister tests from the project panel

- Install native host tools (CMake, Python 3, Ninja, DTC, GCC, etc.) automatically on Linux, macOS, and Windows
- Install and manage Zephyr SDK versions and per-architecture toolchains
- Add projects from existing applications or Zephyr samples, with multiple builds and per-build board and configuration overrides
- Store project configuration in a version-controllable `.vscode/zephyr-ide.json`, which can specify SDKs, packages, and blobs

Compatibility

- Windows
- Linux
- macOS

Getting started

1. Install the extension

   Install IDE for Zephyr from the VS Code Marketplace or the Open VSX Registry.

   An extension pack bundling Cortex-Debug, C/C++, Serial Monitor, Devicetree LSP, and CMake support is also available on the VS Code Marketplace (Extension Pack) and the Open VSX Registry (Extension Pack).

2. Open the Overview Page and Install Host Tools

   Click the *Host Tools* card. The extension verifies that the required build dependencies (CMake, Python 3, Ninja, DTC, GCC, etc.) are on the PATH and can install any missing tools automatically.

3. Configure a West workspace

   Click the *Workspace* card and choose a setup method:

   - **IDE for Zephyr Workspace from Git** — clone a repository that already contains a pre-configured IDE for Zephyr workspace.
   - **West Workspace from Git** — clone an existing west-based Zephyr repository.
   - **Standard Workspace** — create a fresh workspace with a Python virtual environment, west installation, and Zephyr repository initialization.
   - **Open Current Directory** — adopt an existing `.west` folder or link to an external Zephyr installation via *ZEPHYR_BASE*.

   This step also prompts you to install a Zephyr SDK if needed. You can manage SDKs later from the *Zephyr SDK* card on the Overview page.

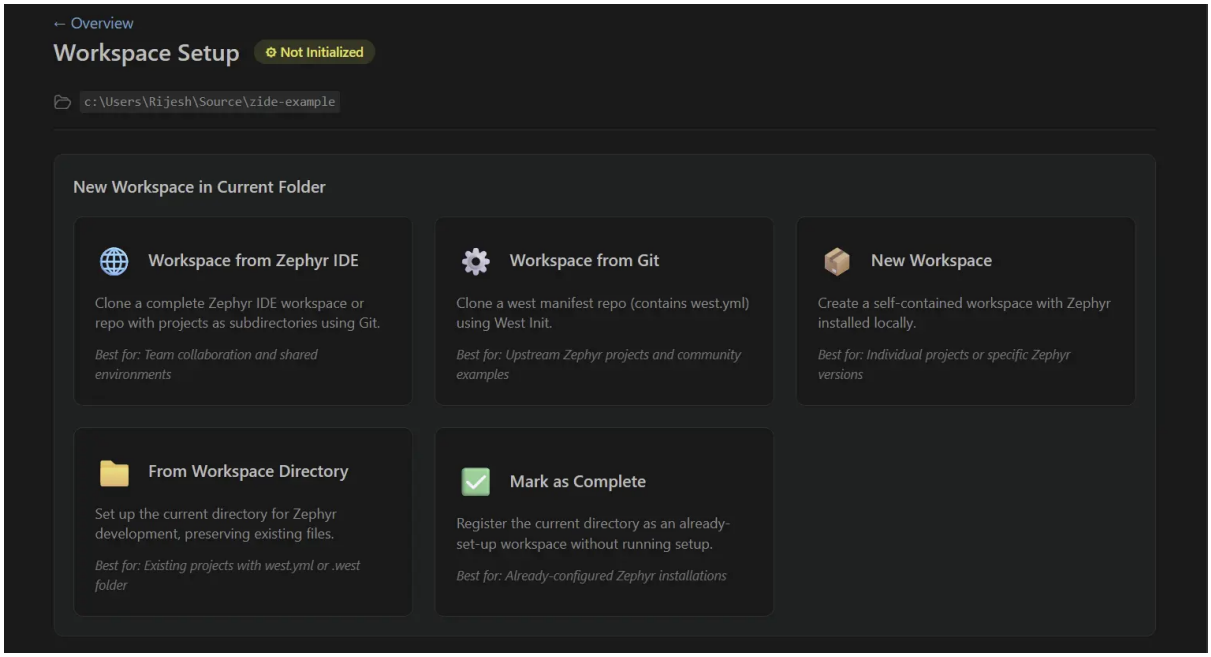
4. Add a project and a build

   In the Project panel, click *Add Project* to add an existing application or copy a Zephyr sample as a starting point.

   After adding a project, click *Add Build* to create a build configuration. Select the target board and, optionally, a runner profile. Each project can have multiple builds targeting different boards or configurations.

5. Build and flash the application

   Use the status bar buttons or the *Project Build* panel to build, flash, or run a pristine build. Build output is shown in the integrated terminal.



6. Configure and run a debug session

   IDE for Zephyr ships a built-in `zephyr-ide-west` debugger type that reads `runners.yaml` from the active build and translates it into a Cortex-Debug session automatically. No `.vscode/launch.json` entry is required to get started. The `zephyr-ide-west` provider accepts the arguments you would pass to `west debugserver`, while the `zephyr-ide-cortex` provider accepts the arguments you would pass to `cortex-debug` directly. You can also set up your own launch configuration and bind it to a build. The extension provides commands that can be resolved on launch.

   To add a launch configuration manually, use the minimal form:

```
{
  "name": "Zephyr IDE: Debug",
  "type": "zephyr-ide-west",
  "request": "launch"
}
```

   The built-in provider selects the runner from `runners.yaml`, resolves the ELF and GDB paths, and forwards the session to Cortex-Debug.

Sharing project configuration

Project settings, builds, runner profiles, Kconfig overlays, devicetree overlays, and per-build west and CMake arguments are stored in `.vscode/zephyr-ide.json`. The file is human-readable and can be committed to version control so team members share the same workspace configuration.

Useful links

- Explore the Extension repository
- Read the Full documentation
- Try the Sample project

2.16.4 Kapa.ai Documentation Assistant

The Zephyr documentation website is augmented with an AI assistant powered by Kapa.ai. The assistant is trained on Zephyr's own documentation, source code, and GitHub activity (see *Data sources* below) and can answer free-form questions in natural language, citing the documentation pages and resources it used to build each answer.

The same knowledge base is also exposed through a Model Context Protocol (MCP) server, so that external AI assistants and IDEs can query Zephyr knowledge directly (see *Using the MCP server*).

> **Note**
>
> Answers are AI-generated and may contain inaccuracies. Always verify important information against the official documentation, and treat the assistant as a starting point rather than an authoritative source.

Using the chatbot

The chatbot is available from every page of the documentation:

- Click the chatbot button displayed at the bottom of the page, or
- Press `Ctrl+K` (`Cmd+K` on macOS) to open it from anywhere.

The assistant opens in a sidebar where you can ask questions in plain language, for example *"How do I configure a UART device in the devicetree?"* or *"What is the difference between a workqueue and a thread?"*. Answers include links back to the relevant documentation pages, GitHub issues, or source files so you can dig deeper.

The first time you use the assistant, a one-time consent screen explains how the feature works and what data is collected. See the *Privacy* section below.

> **Tip**
>
> You don't have to ask your questions in English. The assistant detects the language you use and responds in the same language, so you can interact with it in your preferred language (English, Chinese, Spanish, Japanese, French, German, and many others).

AI-powered search

In addition to the chatbot, Kapa.ai can power the documentation search box:

1. Click the gear (*Search settings*) icon next to the search field.
2. Select *Kapa AI search* from the menu.
3. Type your query and press `Enter`.

Instead of the built-in keyword search, your query is answered by the AI assistant. Your selection is remembered, so subsequent searches keep using the engine you picked until you change it back.

Using the MCP server

Kapa.ai also exposes the Zephyr knowledge base as an MCP (Model Context Protocol) server. MCP is an open standard that lets AI assistants and AI-enabled IDEs connect to external tools and knowledge sources. By adding the Zephyr MCP server to your assistant, you give it the ability to query up-to-date Zephyr documentation, source code, and GitHub activity on your behalf.

The server is available at the following endpoint, using the streamable HTTP transport:

```
https://zephyrproject.mcp.kapa.ai
```

The exact configuration steps depend on the MCP client you use. Most clients read a JSON configuration file with an `mcpServers` section similar to the following:

```
{
  "mcpServers": {
    "zephyr-docs": {
      "url": "https://zephyrproject.mcp.kapa.ai"
    }
  }
}
```

Some clients instead expect the server to be launched through a local stdio bridge such as mcp-remote:

```
{
  "mcpServers": {
    "zephyr-docs": {
      "command": "npx", "args": ["-y", "mcp-remote",
↪"https://zephyrproject.mcp.kapa.ai"]
    }
  }
}
```

Refer to the documentation of your specific MCP client (AI coding assistant, IDE extension, etc.) for the precise location and syntax of its configuration.

Tip

Connecting the MCP server to an AI coding assistant lets it answer Zephyr questions and ground its code suggestions in the project's actual documentation and source, rather than relying solely on its training data.

Data sources

Kapa.ai builds its answers exclusively from the following Zephyr project sources, which are synchronized at the indicated frequencies:

Source	Details	Refresh frequency
Source code	`zephyrproject-rtos/zephyr` repository (excluding the `boards/` and `doc/` folders); C, Markdown, Python, and text files	Hourly
API reference	https://docs.zephyrproject.org/latest/doxygen/html/index.html	Daily
Devicetree bindings	https://docs.zephyrproject.org/latest/build/dts/api/bindings.html	Daily
Main documentation	https://docs.zephyrproject.org/latest/ (excluding the API reference and Devicetree bindings sections)	Daily
Project wiki	https://github.com/zephyrproject-rtos/zephyr/wiki	Daily
GitHub issues	`zephyrproject-rtos/zephyr` issues from the last 6 months	Every 5 minutes
GitHub pull requests	`zephyrproject-rtos/zephyr` pull requests from the last 6 months (excluding those closed without being merged)	Every 10 minutes

For the authoritative and most up-to-date configuration, refer to the Kapa.ai page on the Zephyr project infrastructure wiki.

Privacy

The assistant uses your questions to provide answers based on the Zephyr documentation, source code, and GitHub issues and pull requests. Questions and interactions may be anonymously collected and analyzed to help improve the documentation by identifying areas that need clarification. No personally identifiable information is collected.

2.16.5 STM32CubeIDE

STM32CubeIDE is an Eclipse-based integrated development environment from STMicroelectronics designed for the STM32 series of MCUs and MPUs.

This guide describes the process of setting up, building, and debugging Zephyr applications using the IDE.

A project must have already been created with Zephyr and west.

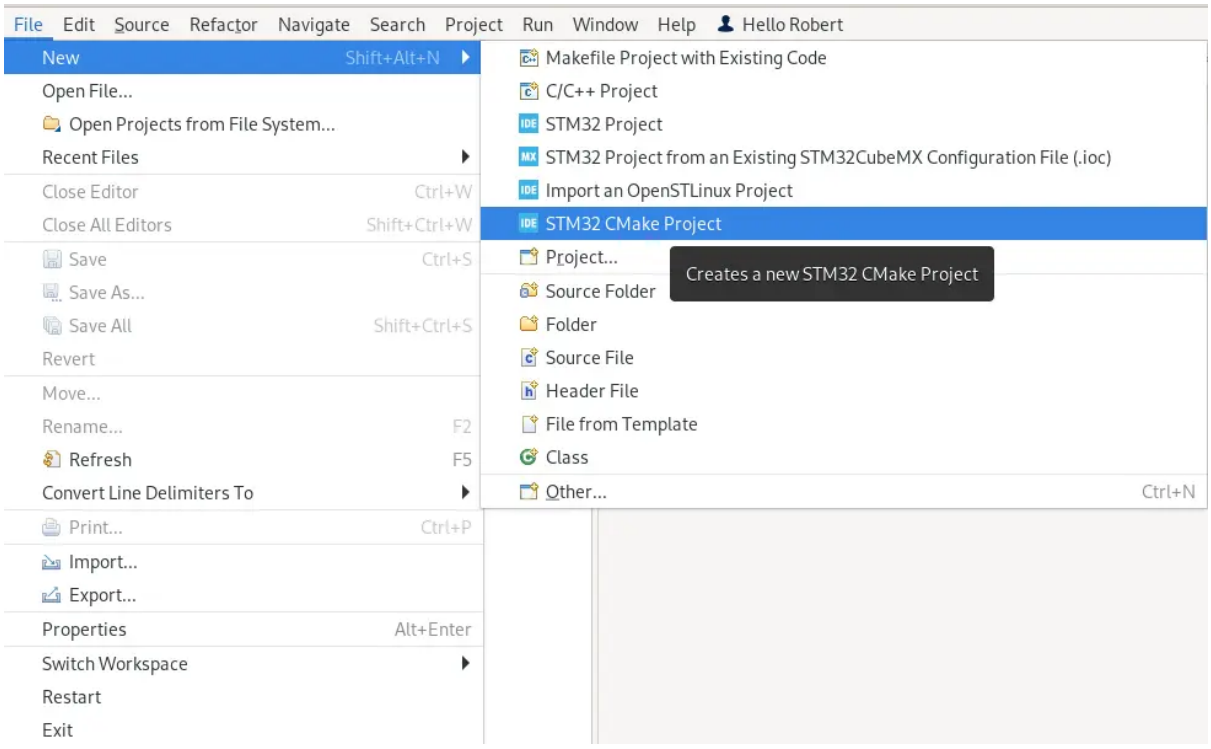
These directions have been validated to work with version 1.16.0 of the IDE on Linux.

Project Setup

1. Before you start, make sure you have a working Zephyr development environment, as per the instructions in the *Getting Started Guide*.
2. Run STM32CubeIDE from your Zephyr environment. Example:

   ```
   $ /opt/st/stm32cubeide_1.16.0/stm32cubeide
   ```

3. Open your already existing project by going to *File* ▸ *New* ▸ *STM32 CMake Project*:
4. Select *Project with existing CMake sources*, then click *Next*.
5. Select *Next* and browse to the location of your sources. The folder that is opened should have the `CMakeLists.txt` and `prj.conf` files.
6. Select *Next* and select the appropriate MCU. Press *Finish* and your project should now be available. However, more actions must still be done in order to configure it properly.



7. Right-click on the newly created project in the workspace, and select *Properties*.
8. Go to the *C/C++ Build* page and set the Generator to `Ninja`. In *Other Options*, specify the target `BOARD` in CMake argument format. If an out-of-tree board is targeted, the `BOARD_ROOT` option must also be set. The resulting settings page should look similar to this:

   These options may or may not be needed depending on if you have an out-of-tree project or not.
9. Go to the *C/C++ General* ▸ *Preprocessor Include* page. Select the *GNU C* language, and click on the *CDT User Settings Entries* option.

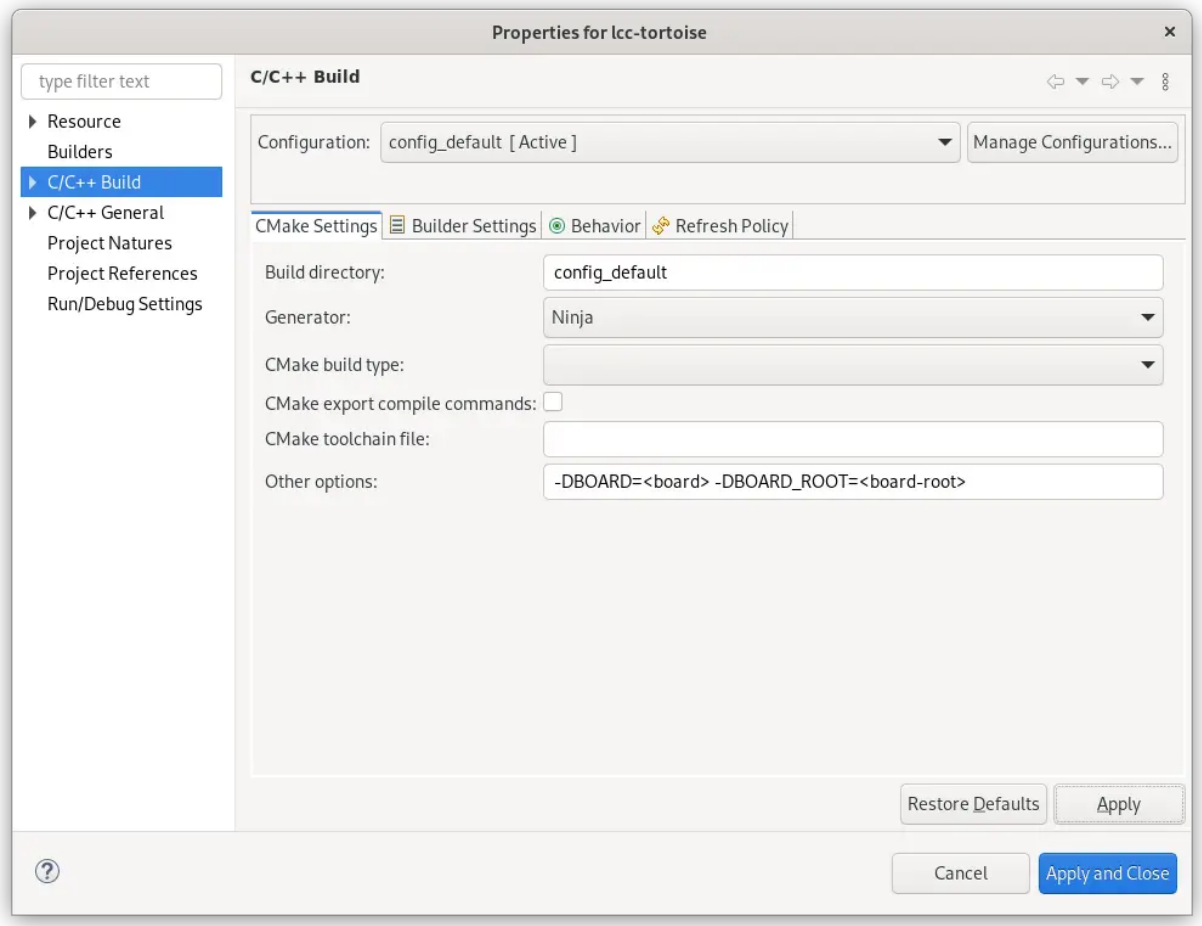
   Click *Add* to add an *Include File* that points to Zephyr's `autoconf.h`, which is located in `<build dir>/zephyr/include/generated/autoconf.h`. This will ensure that STM32CubeIDE picks up Zephyr configuration options. The following dialog will be shown. Fill it in as follows:

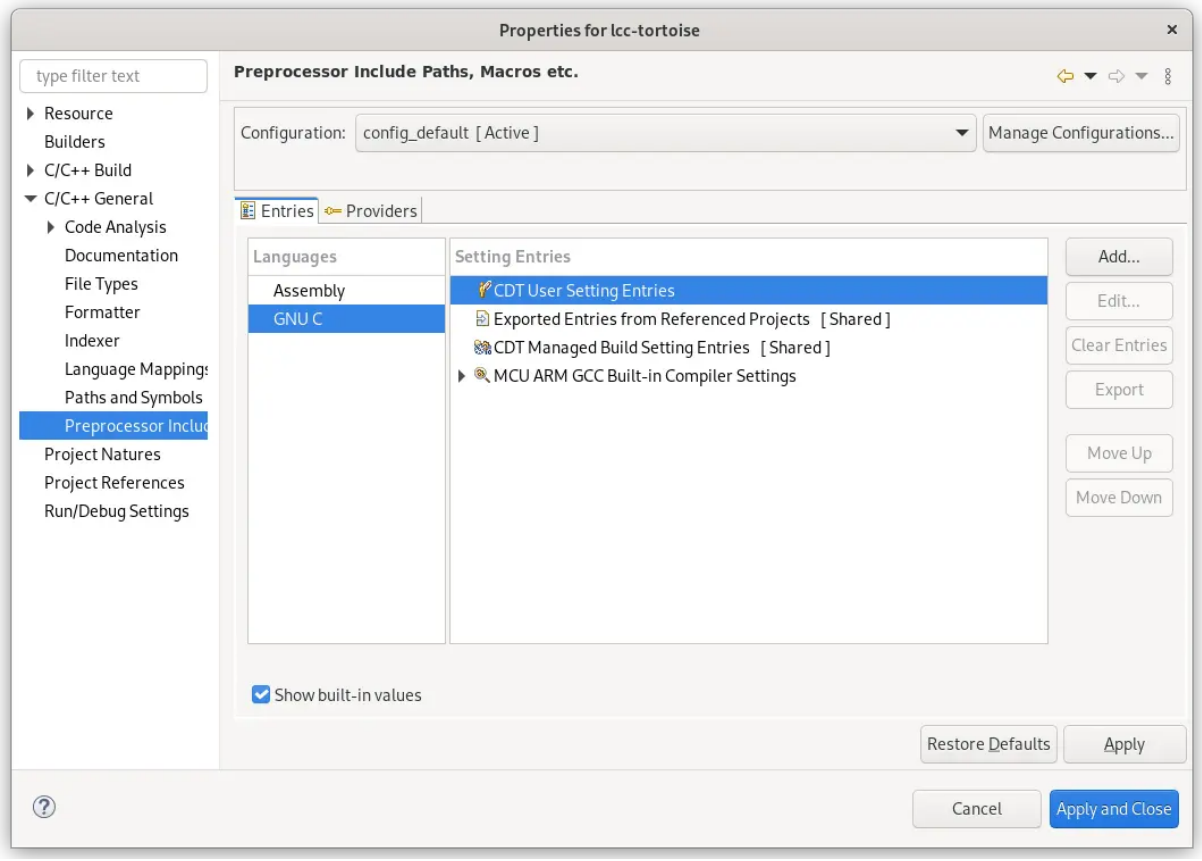
   Once the include file has been added, your properties page should look similar to the following:
10. Click *Apply and Close*
11. You may now build the project using the *Build* button on the toolbar. The project can be run using the *Run* button, as well as debugged using the *Debug* button.

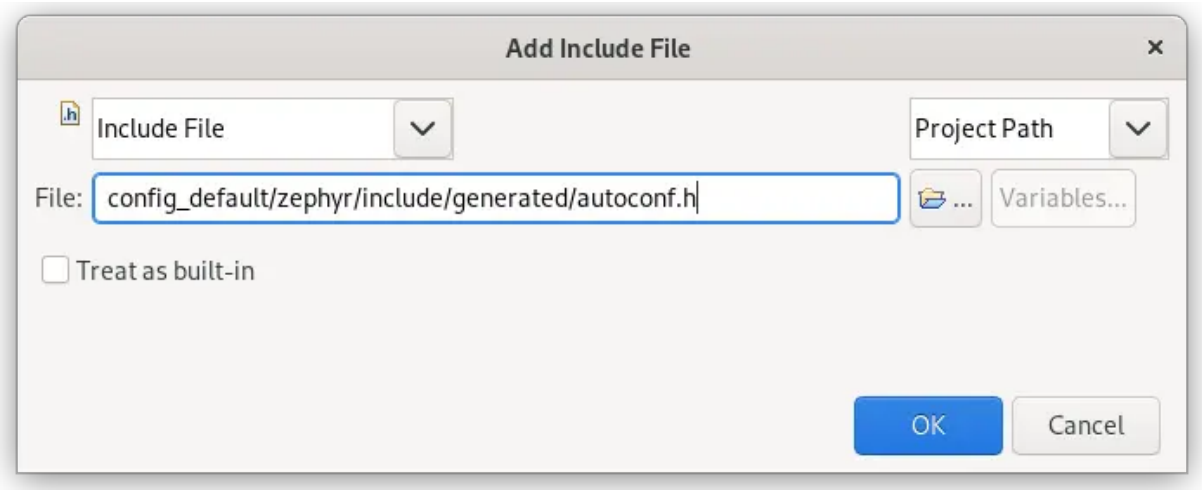
Debugging only

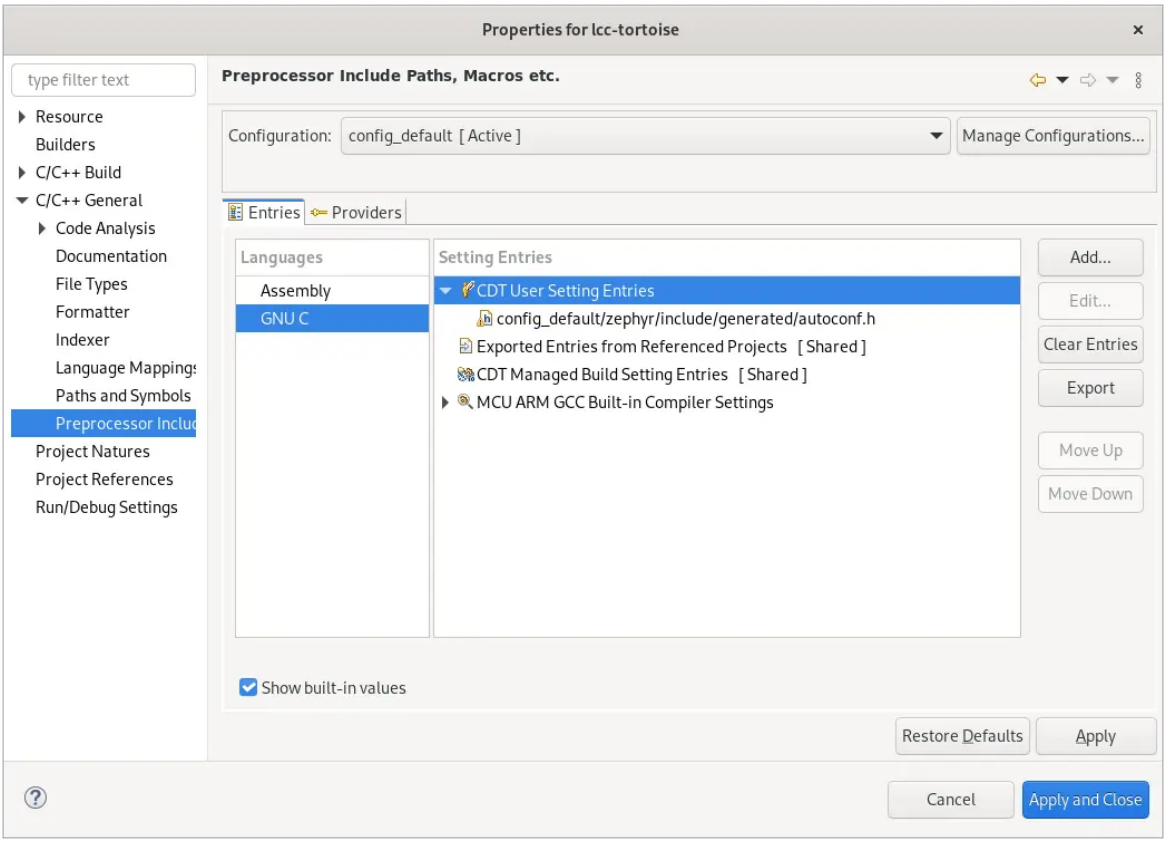
If you only want to use STM32CubeIDE to debug your project you can proceed as follows:

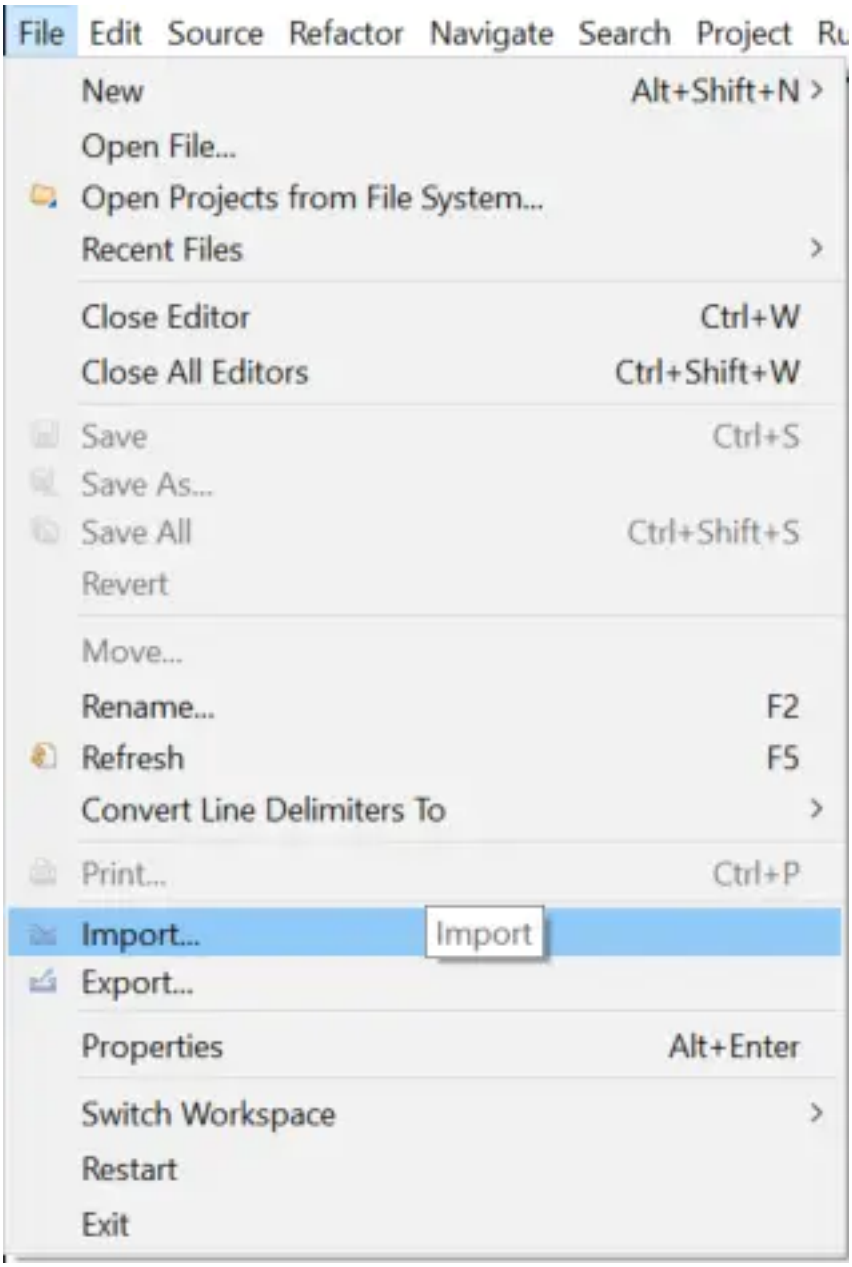
1. First, make sure to compile your project and have the `zephyr.elf` available.
2. Run STM32CubeIDE and import your project by going to *File* ▸ *Import...*:
3. Select *C/C++* ▸ *STM32 Cortex-M Executable*, then click *Next*:
4. Click on *Browse* to browse to your build folder and select your `zephyr.elf`.
5. Click on *Select* to select your MCU. If relevant, choose also your CPU and/or core.
6. Click on *Finish*.

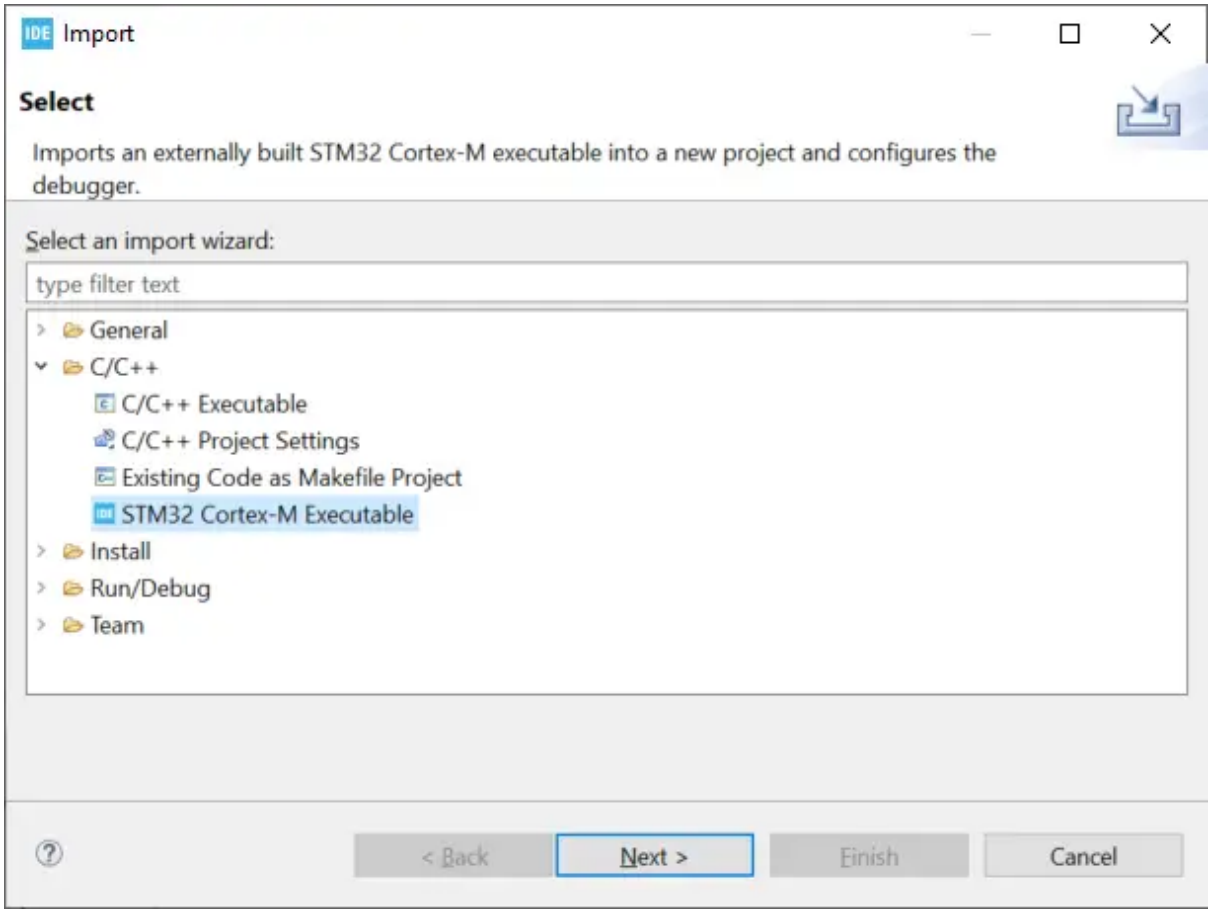

Properties for lcc-tortoise
type filter text
Resource
Builders
C/C++ Build
C/C++ General
Project Natures
Project References
Run/Debug Settings
C/C++ Build
Configuration:
config_default [Active]
Manage Configurations...
CMake Settings
Builder Settings
Behavior
Refresh Policy
Build directory:
config_default
Generator:
Ninja
CMake build type:
CMake export compile commands:
CMake toolchain file:
Other options:
-DBOARD=<board> -DBOARD_ROOT=<board-root>
Restore Defaults
Apply
Cancel
Apply and Close


Properties for lcc-tortoise
type filter text
Resource
Builders
C/C++ Build
C/C++ General
Code Analysis
Documentation
File Types
Formatter
Indexer
Language Mapping
Paths and Symbols
Preprocessor Includ
Project Natures
Project References
Run/Debug Settings
Preprocessor Include Paths, Macros etc.
Configuration:
config_default [Active]
Manage Configurations...
Entries
Providers
Languages
Assembly
GNU C
Setting Entries
CDT User Setting Entries
Exported Entries from Referenced Projects [Shared]
CDT Managed Build Setting Entries [Shared]
MCU ARM GCC Built-in Compiler Settings
Add...
Edit...
Clear Entries
Export
Move Up
Move Down
Show built-in values
Restore Defaults
Apply
Cancel
Apply and Close


Add Include File
Include File
Project Path
File:
config_default/zephyr/include/generated/autoconf.h
Variables...
Treat as built-in
OK
Cancel


Properties for lcc-tortoise
type filter text
Resource
Builders
C/C++ Build
C/C++ General
Code Analysis
Documentation
File Types
Formatter
Indexer
Language Mapping
Paths and Symbols
Preprocessor Inclu
Project Natures
Project References
Run/Debug Settings
Preprocessor Include Paths, Macros etc.
Configuration:
config_default [Active]
Manage Configurations...
Entries
Providers
Languages
Assembly
GNU C
Setting Entries
CDT User Setting Entries
config_default/zephyr/include/generated/autoconf.h
Exported Entries from Referenced Projects [Shared]
CDT Managed Build Setting Entries [Shared]
MCU ARM GCC Built-in Compiler Settings
Add...
Edit...
Clear Entries
Export
Move Up
Move Down
Show built-in values
Restore Defaults
Apply
Cancel
Apply and Close


File Edit Source Refactor Navigate Search Project Ru
New Alt+Shift+N
Open File...
Open Projects from File System...
Recent Files
Close Editor Ctrl+W
Close All Editors Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Import...
Import
Export...
Properties Alt+Enter
Switch Workspace
Restart
Exit



7. The project can now be debugged using the *Debug* button.

2.16.6 Visual Studio Code

Visual Studio Code (VS Code for short) is a popular cross-platform IDE that supports C projects and has a rich set of extensions.

This guide describes the process of setting up VS Code for Zephyr's blinky sample in VS Code.

The instructions have been tested on Linux, but the steps should be the same for macOS and Windows, just make sure to adjust the paths if needed.

Get VS Code

Download VS Code and install it.

Install the required extensions through the *Extensions* marketplace in the left panel. Search for the C/C++ Extension Pack and install it.

Initialize a new workspace

This guide gives details on how to configure the blinky sample application, but the instructions would be similar for any Zephyr project and *workspace layout*.

Before you start, make sure you have a working Zephyr development environment, as per the instructions in the *Getting Started Guide*.

Open the project in VS Code

1. In VS Code, select *File* ▸ *Open Folder* from the main menu.

2. Navigate to your Zephyr workspace and select it (i.e. the `zephyrproject` folder in your HOME directory if you have followed the Getting Started instructions).
3. If prompted, enable workspace trust.

Generate compile commands

In order to support code navigation and linting capabilities, you must compile your project once to generate the `compile_commands.json` file that will provide the C/C++ extension with the required information (ex. include paths). You may do it from VS Code embedded terminal; select *Terminal* ▸ *New Terminal* from the top menu or the command palette (`Ctrl+Shift+P`), then type:

```
$ cd zephyr
$ west build -p always -b native_sim/native/64 samples/basic/blinky
```

Configure the C/C++ extension

You'll now need to point to the generated `compile_commands.json` file to enable linting and code navigation in VS Code.

1. Go to the *File* ▸ *Preferences* ▸ *Settings* in the VS Code top menu.
2. Search for the parameter *C_Cpp* > *Default: Compile Commands* and set its value to: `zephyr/build/compile_commands.json`.

   Linting errors in the code should now be resolved, and you should be able to navigate through the code.

Additional resources

There are many other extensions that can be useful when working with Zephyr and VS Code. While this guide does not cover them yet, you may refer to their documentation to set them up:

Contribution tooling

- Checkpatch Extension
- EditorConfig Extension

Documentation languages extensions

- reStructuredText Extension Pack

IDE extensions

- CMake Extension documentation
- nRF Kconfig Extension
- nRF DeviceTree Extension
- GNU Linker Map files Extension

Additional guides

- How to Develop Zephyr Apps with a Modern, Visual IDE

> **Note**
>
> Please be aware that these extensions might not all have the same level of quality and maintenance.

2.16.7 Workbench for Zephyr

Workbench for Zephyr is a Visual Studio Code (VS Code) extension that adds Zephyr development support, including **SDK management**, **project creation wizard**, **build/flash**, and **debugging**.

Check the Getting started tutorial for a walkthrough.

Key features

- Install native Host Tools (Python, CMake, ...)
- Install and assign toolchains (Zephyr SDK, IAR, ...)
- Import West workspaces
- Create and import Zephyr applications
- Build and flash applications
- Debug applications
- Install runners automatically
- Perform memory and static analysis

Compatibility

- Windows 10-11
- Linux (x86_64)
  - Ubuntu
  - Debian
  - Fedora
  - Other distributions may also work (not tested)
- macOS

Getting started

1. Install the extension

   Install Workbench for Zephyr from the VS Code Marketplace.

2. Open the Workbench for Zephyr extension

   In the Activity Bar, click the *Workbench for Zephyr* icon.

3. Install Host Tools

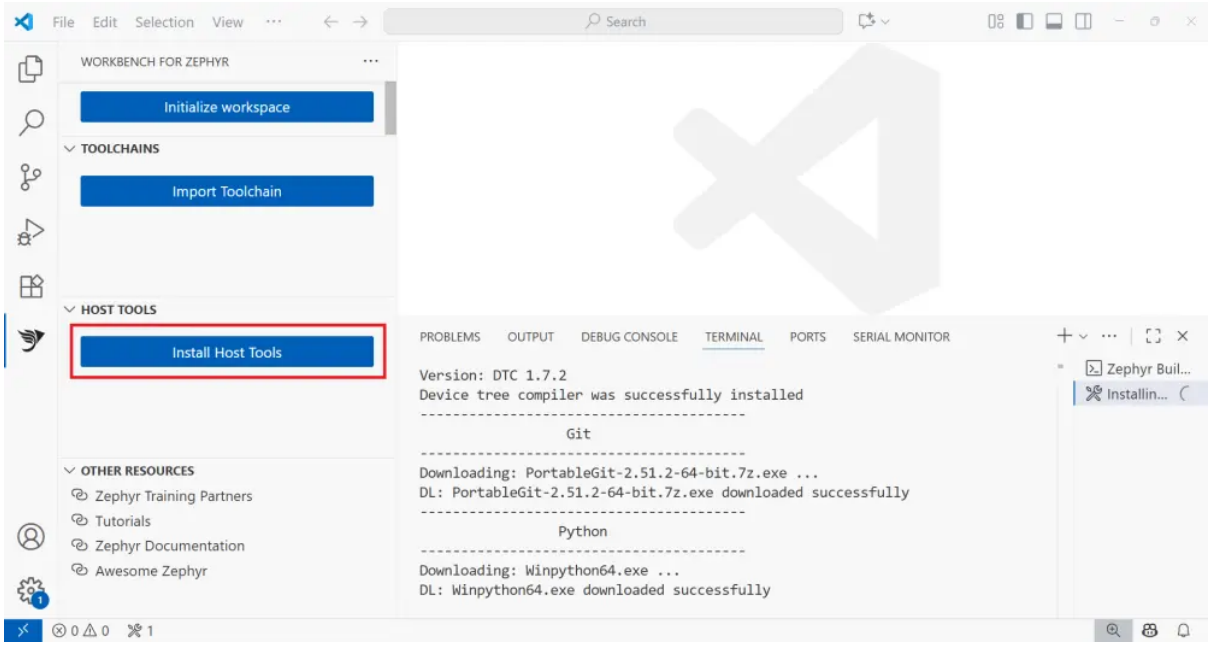
   Click *Install Host Tools* to download and install the native tools (typically into `${HOME}/.zinstaller`).

   > **Note**
   >
   > Some host tools may require administrator privileges. On Windows, this is required when installing 7z. On Linux, this is required when installing tools using the package manager, for example when running `apt install`.

4. Import a toolchain

   Click *Import Toolchain*, choose a toolchain, and select the destination folder.

   A toolchain provides the compiler and debugger required to build and debug Zephyr applications. The Zephyr SDK is the recommended option and can be installed as either the full



package or a minimal target-specific version. Workbench also supports other toolchains, such as IAR.

5. Initialize / Import a West workspace

   Click *Initialize workspace* and fill the workspace information.

   Workbench will create the workspace and parse the west manifest to configure projects.

6. Create a new application

   On Workbench for Zephyr, new projects are based on samples from Zephyr sources.

   - Click *Create New Application*.
   - Select the *West Workspace* to attach to.
   - Select the *Zephyr SDK* to use.
   - Select the target *Board* (for example, `ST STM32F4 Discovery`).
   - Select the *Sample* project to base on (for example, `hello_world`).
   - Enter the project name.
   - Enter the project location.

7. Build the application

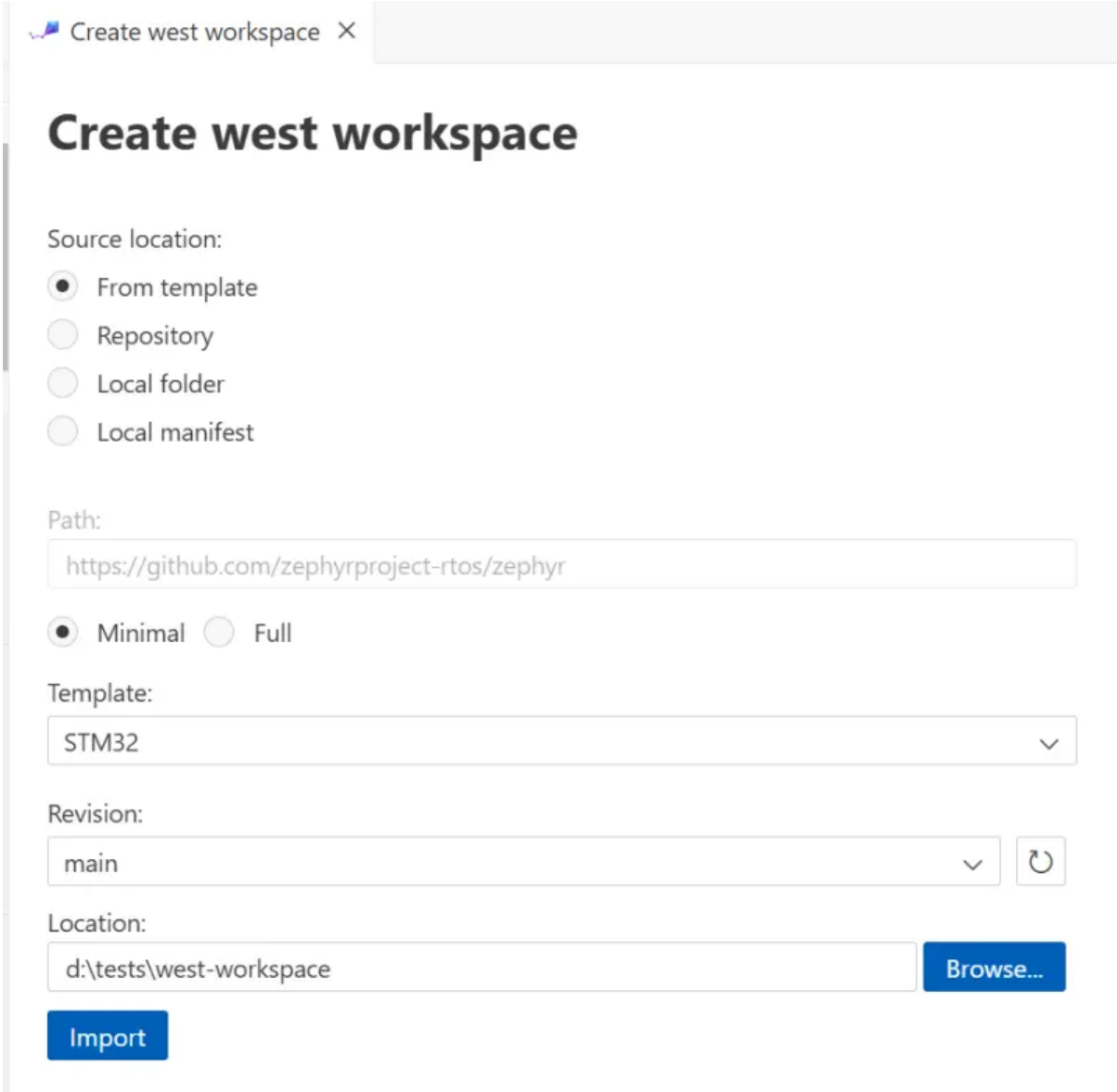
   Click *Build* in the status bar, or select the application folder to build. Build output is shown in the integrated terminal.

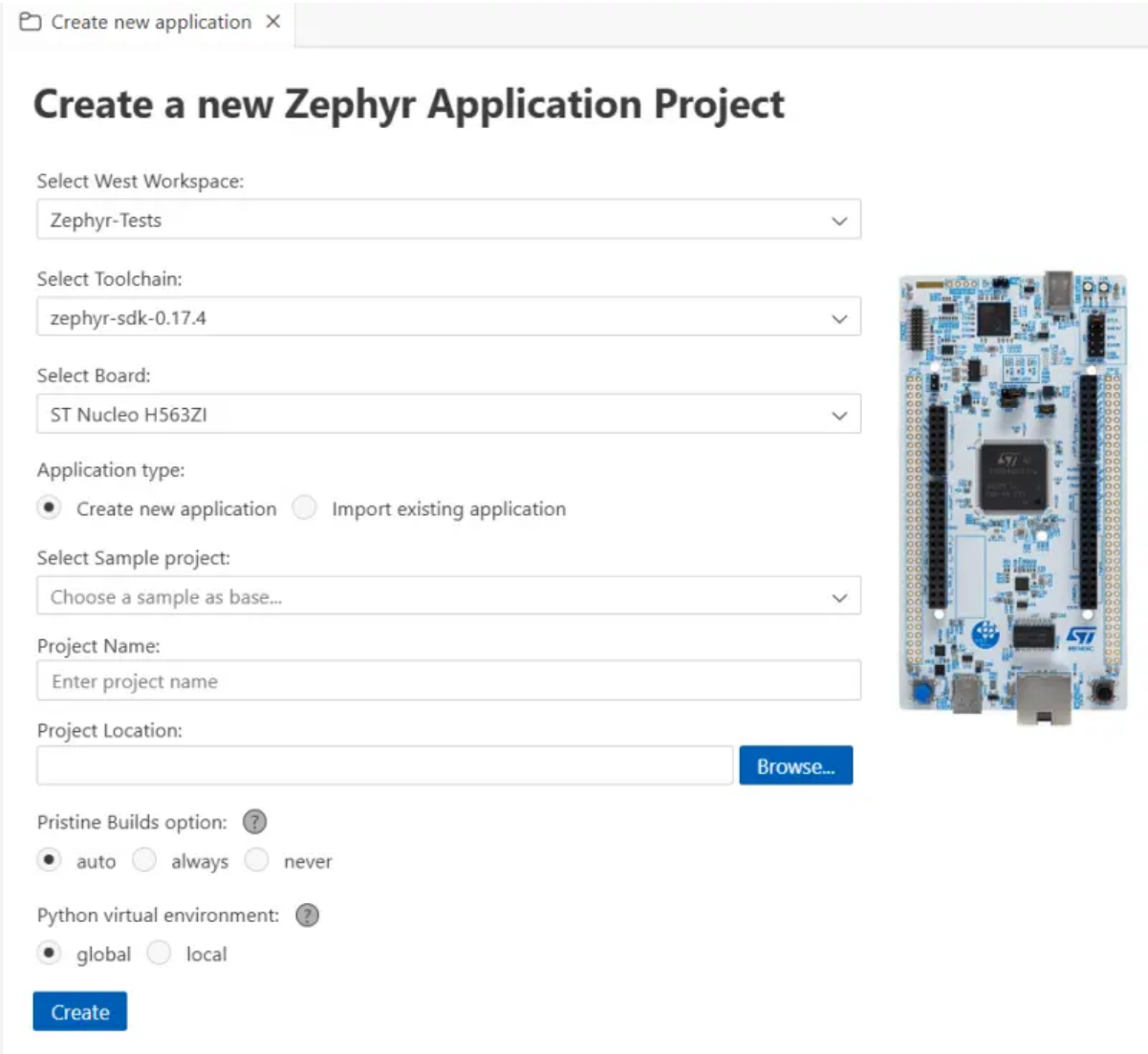
8. Configure and run a debug session

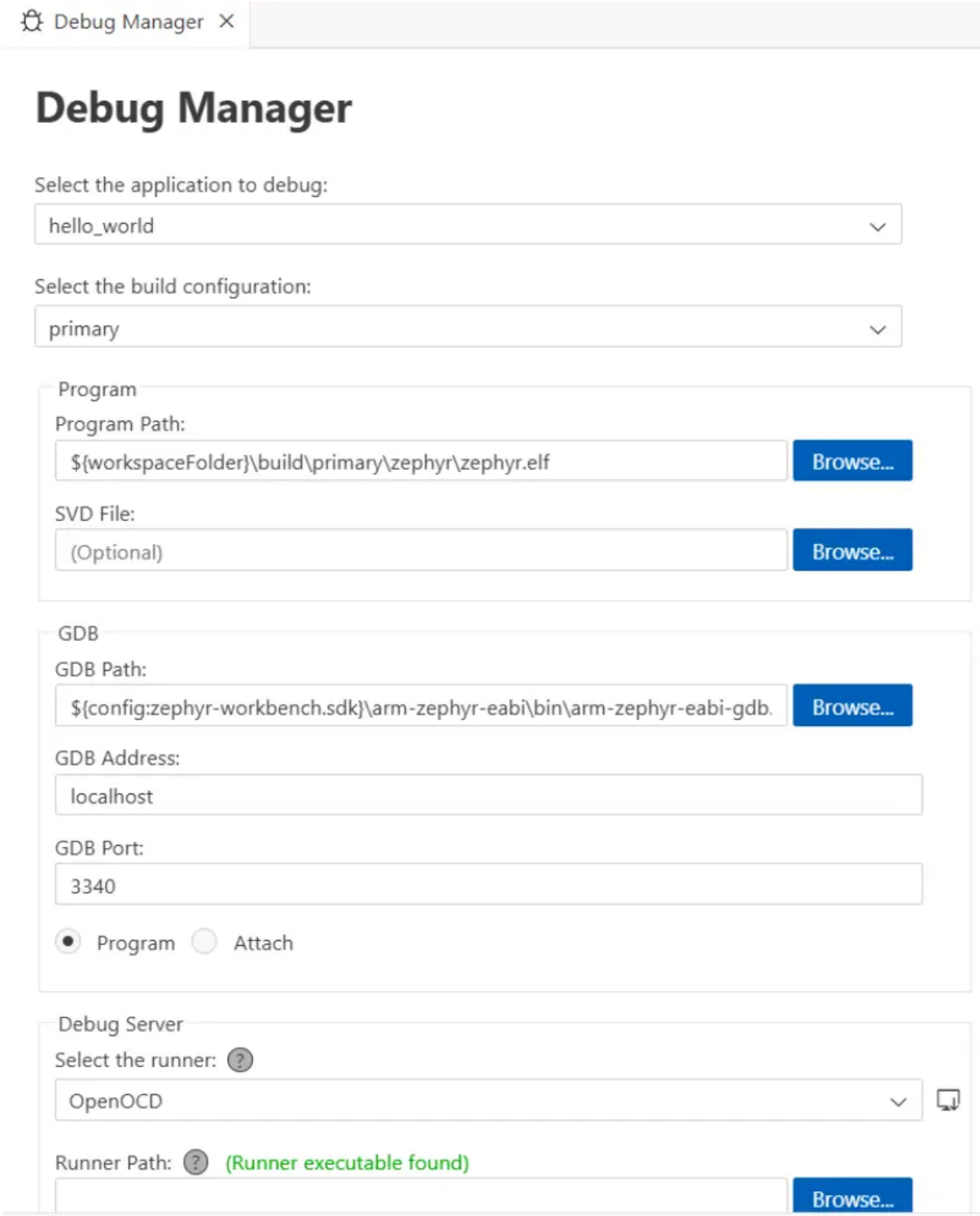
   Use *Debug Manager* to generate/update a debug configuration (`.vscode/launch.json`):

   - Generated ELF (Program Path)
   - SVD file (optional)
   - GDB/port/address (if needed)
   - Debug server/runner (OpenOCD, J-Link, LinkServer, pyOCD, ...)

   Then start debugging normally via *Run and Debug*


Create west workspace
Create west workspace
Source location:
From template
Repository
Local folder
Local manifest
Path:
https://github.com/zephyrproject-rtos/zephyr
Minimal
Full
Template:
STM32
Revision:
main
Location:
d:\tests\west-workspace
Browse...
Import


Create new application
Create a new Zephyr Application Project
Select West Workspace:
Zephyr-Tests
Select Toolchain:
zephyr-sdk-0.17.4
Select Board:
ST Nucleo H563ZI
Application type:
Create new application
Import existing application
Select Sample project:
Choose a sample as base...
Project Name:
Enter project name
Project Location:
Browse...
Pristine Builds option:
auto
always
never
Python virtual environment:
global
local
Create


Debug Manager
Debug Manager
Select the application to debug:
hello_world
Select the build configuration:
primary
Program
Program Path:
${workspaceFolder}\build\primary\zephyr\zephyr.elf
Browse...
SVD File:
(Optional)
Browse...
GDB
GDB Path:
${config:zephyr-workbench.sdk}\arm-zephyr-eabi\bin\arm-zephyr-eabi-gdb.
Browse...
GDB Address:
localhost
GDB Port:
3340
Program
Attach
Debug Server
Select the runner:
OpenOCD
Runner Path:
(Runner executable found)
Browse...

Install Runners

Workbench for Zephyr can provide installers for some runners (for example, OpenOCD and STM32CubeProgrammer). Use *Install Runners* to view supported tools and install them (or open the vendor website).

Useful links

- You can explore the Extension repository
- For more details, check out the Full documentation

Chapter 3

Kernel

3.1 Kernel Services

The Zephyr kernel lies at the heart of every Zephyr application. It provides a low footprint, high performance, multi-threaded execution environment with a rich set of available features. The rest of the Zephyr ecosystem, including device drivers, networking stack, and application-specific code, uses the kernel's features to create a complete application.

The configurable nature of the kernel allows you to incorporate only those features needed by your application, making it ideal for systems with limited amounts of memory (as little as 2 KB!) or with simple multi-threading requirements (such as a set of interrupt handlers and a single background task). Examples of such systems include: embedded sensor hubs, environmental sensors, simple LED wearable, and store inventory tags.

Applications requiring more memory (50 to 900 KB), multiple communication devices (like Wi-Fi and Bluetooth Low Energy), and complex multi-threading, can also be developed using the Zephyr kernel. Examples of such systems include: fitness wearables, smart watches, and IoT wireless gateways.

3.1.1 Scheduling, Interrupts, and Synchronization

These pages cover basic kernel services related to thread scheduling and synchronization.

Threads

Note

There is also limited support for using *Operation without Threads*.

- *Lifecycle*
  - *Thread Creation*
  - *Starting a Thread*
  - *Thread Termination*
  - *Thread Aborting*
  - *Thread Suspension*

- *Thread States*
- *Thread Stack objects*
  - *Kernel-only Stacks*
  - *Thread stacks*
- *Thread Priorities*
  - *Meta-IRQ Priorities*
- *Thread Options*
- *Thread Custom Data*
- *Thread Name*
- *Thread Introspection*
- *Implementation*
  - *Spawning a Thread*
  - *Dropping Permissions*
  - *Terminating a Thread*
- *Runtime Statistics*
- *Runtime Stack Safety*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

This section describes kernel services for creating, scheduling, and deleting independently executable threads of instructions.

A *thread* is a kernel object that is used for application processing that is too lengthy or too complex to be performed by an ISR.

Any number of threads can be defined by an application (limited only by available RAM). Each thread is referenced by a *thread id* that is assigned when the thread is spawned.

A thread has the following key properties:

- A **stack area**, which is a region of memory used for the thread's stack. The **size** of the stack area can be tailored to conform to the actual needs of the thread's processing. Special macros exist to create and work with stack memory regions.
- A **thread control block** for private kernel bookkeeping of the thread's metadata. This is an instance of type `k_thread`.
- An **entry point function**, which is invoked when the thread is started. Up to 3 **argument values** can be passed to this function.
- A **scheduling priority**, which instructs the kernel's scheduler how to allocate CPU time to the thread. (See *Scheduling*.)
- A set of **thread options**, which allow the thread to receive special treatment by the kernel under specific circumstances. (See *Thread Options*.)
- A **start delay**, which specifies how long the kernel should wait before starting the thread.
- An **execution mode**, which can either be supervisor or user mode. By default, threads run in supervisor mode and allow access to privileged CPU instructions, the entire memory

address space, and peripherals. User mode threads have a reduced set of privileges. This depends on the `CONFIG_USERSPACE` option. See *User Mode*.

Lifecycle

Thread Creation A thread must be created before it can be used. The kernel initializes the thread control block as well as one end of the stack portion. The remainder of the thread's stack is typically left uninitialized.

Specifying a start delay of `K_NO_WAIT` instructs the kernel to start thread execution immediately. Alternatively, the kernel can be instructed to delay execution of the thread by specifying a timeout value – for example, to allow device hardware used by the thread to become available.

The kernel allows a delayed start to be canceled before the thread begins executing. A cancellation request has no effect if the thread has already started. A thread whose delayed start was successfully canceled must be re-spawned before it can be used.

Starting a Thread A thread that is created with `K_FOREVER` as its start delay is not added to the scheduler's ready queue and does not begin executing until it is explicitly started. The `k_thread_start()` function starts such an inactive thread, making it eligible for scheduling. Starting a thread that has already started has no effect.

This is useful when a thread object must be created at one point in time but its execution deferred until the application has completed additional setup.

Thread Termination Once a thread is started it typically executes forever. However, a thread may synchronously end its execution by returning from its entry point function. This is known as **termination**.

A thread that terminates is responsible for releasing any shared resources it may own (such as mutexes and dynamically allocated memory) prior to returning, since the kernel does *not* reclaim them automatically.

In some cases a thread may want to sleep until another thread terminates. This can be accomplished with the `k_thread_join()` API. This will block the calling thread until either the timeout expires, the target thread self-exits, or the target thread aborts (either due to a `k_thread_abort()` call or triggering a fatal error).

Once a thread has terminated, the kernel guarantees that no use will be made of the thread struct. The memory of such a struct can then be reused for any purpose, including spawning a new thread. Note that the thread must be fully terminated, which presents race conditions where a thread's own logic signals completion which is seen by another thread before the kernel processing is complete. Under normal circumstances, application code should use `k_thread_join()` or `k_thread_abort()` to synchronize on thread termination state and not rely on signaling from within application logic.

Thread Aborting A thread may asynchronously end its execution by **aborting**. The kernel automatically aborts a thread if the thread triggers a fatal error condition, such as dereferencing a null pointer.

A thread can also be aborted by another thread (or by itself) by calling `k_thread_abort()`. However, it is typically preferable to signal a thread to terminate itself gracefully, rather than aborting it.

As with thread termination, the kernel does not reclaim shared resources owned by an aborted thread.

> **Note**
>
> The kernel does not currently make any claims regarding an application's ability to respawn a thread that aborts.

Thread Suspension A thread can be prevented from executing for an indefinite period of time if it becomes **suspended**. The function `k_thread_suspend()` can be used to suspend any thread, including the calling thread. Suspending a thread that is already suspended has no additional effect.

Once suspended, a thread cannot be scheduled until another thread calls `k_thread_resume()` to remove the suspension.

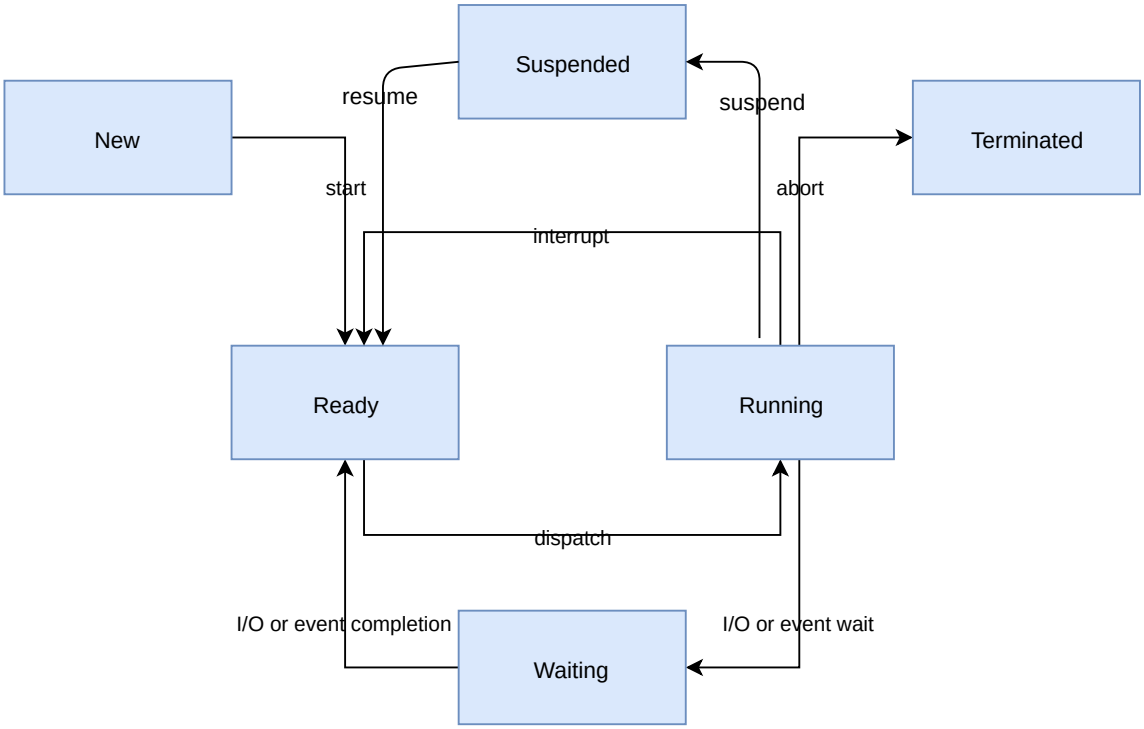
> **Note**
>
> A thread can prevent itself from executing for a specified period of time using `k_sleep()`. However, this is different from suspending a thread since a sleeping thread becomes executable automatically when the time limit is reached.

Thread States A thread that has no factors that prevent its execution is deemed to be **ready**, and is eligible to be selected as the current thread.

A thread that has one or more factors that prevent its execution is deemed to be **unready**, and cannot be selected as the current thread.

The following factors make a thread unready:

- The thread has not been started.
- The thread is waiting for a kernel object to complete an operation. (For example, the thread is taking a semaphore that is unavailable.)
- The thread is waiting for a timeout to occur.
- The thread has been suspended.
- The thread has terminated or aborted.



Note

Although the diagram above may appear to suggest that both **Ready** and **Running** are distinct thread states, that is not the correct interpretation. **Ready** is a thread state, and **Running** is a schedule state that only applies to **Ready** threads.

Thread Stack objects Every thread requires its own stack buffer for the CPU to push context. Depending on configuration, there are several constraints that must be met:

- There may need to be additional memory reserved for memory management structures
- If guard-based stack overflow detection is enabled, a small write-protected memory management region must immediately precede the stack buffer to catch overflows.
- If userspace is enabled, a separate fixed-size privilege elevation stack must be reserved to serve as a private kernel stack for handling system calls.
- If userspace is enabled, the thread's stack buffer must be appropriately sized and aligned such that a memory protection region may be programmed to exactly fit.

The alignment constraints can be quite restrictive, for example some MPUs require their regions to be of some power of two in size, and aligned to its own size.

Because of this, portable code can't simply pass an arbitrary character buffer to `k_thread_create()`. Special macros exist to instantiate stacks statically, prefixed with `K_KERNEL_STACK` and `K_THREAD_STACK`.

Additionally, stacks may be instantiated dynamically using `k_thread_stack_alloc()` and subsequently freed with `k_thread_stack_free()`.

Kernel-only Stacks If it is known that a thread will never run in user mode, or the stack is being used for special contexts like handling interrupts, it is best to define stacks using the `K_KERNEL_STACK` macros.

These stacks save memory because an MPU region will never need to be programmed to cover

the stack buffer itself, and the kernel will not need to reserve additional room for the privilege elevation stack, or memory management data structures which only pertain to user mode threads.

Attempts from user mode to use stacks declared in this way will result in a fatal error for the caller.

If `CONFIG_USERSPACE` is not enabled, the set of `K_THREAD_STACK` macros have an identical effect to the `K_KERNEL_STACK` macros.

Thread stacks If it is known that a stack will need to host user threads, or if this cannot be determined, define the stack with `K_THREAD_STACK` macros. This may use more memory but the stack object is suitable for hosting user threads.

If `CONFIG_USERSPACE` is not enabled, the set of `K_THREAD_STACK` macros have an identical effect to the `K_KERNEL_STACK` macros.

Thread Priorities A thread's priority is an integer value, and can be either negative or non-negative. Numerically lower priorities takes precedence over numerically higher values. For example, the scheduler gives thread A of priority 4 *higher* priority over thread B of priority 7; likewise thread C of priority -2 has higher priority than both thread A and thread B.

The scheduler distinguishes between two classes of threads, based on each thread's priority.

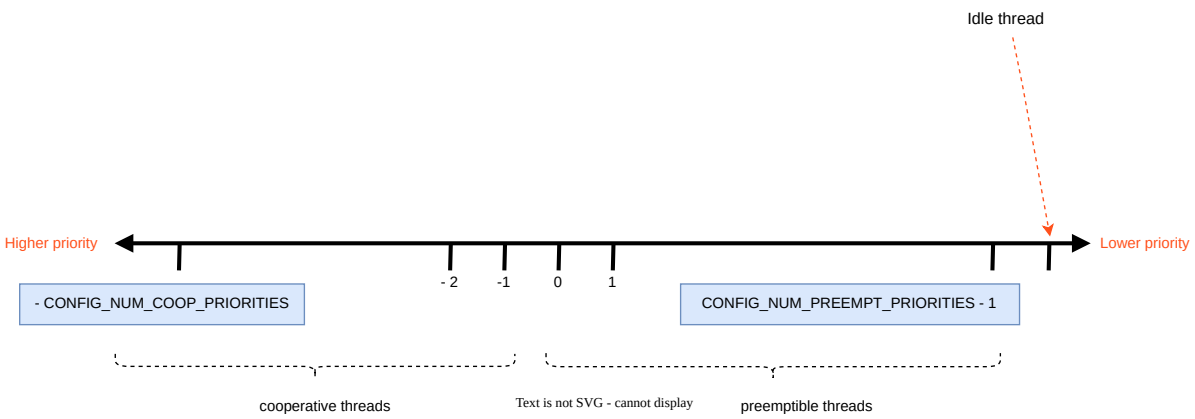
- A *cooperative thread* has a negative priority value. Once it becomes the current thread, a cooperative thread remains the current thread until it performs an action that makes it unready.
- A *preemptible thread* has a non-negative priority value. Once it becomes the current thread, a preemptible thread may be supplanted at any time if a cooperative thread, or a preemptible thread of higher or equal priority, becomes ready.

A thread's initial priority value can be altered up or down after the thread has been started. Thus it is possible for a preemptible thread to become a cooperative thread, and vice versa, by changing its priority.

> **Note**
>
> The scheduler does not make heuristic decisions to re-prioritize threads. Thread priorities are set and changed only at the application's request.

The kernel supports a virtually unlimited number of thread priority levels. The configuration options `CONFIG_NUM_COOP_PRIORITIES` and `CONFIG_NUM_PREEMPT_PRIORITIES` specify the number of priority levels for each class of thread, resulting in the following usable priority ranges:

- cooperative threads: (`-CONFIG_NUM_COOP_PRIORITIES`) to -1
- preemptive threads: 0 to (`CONFIG_NUM_PREEMPT_PRIORITIES` - 1)



For example, configuring 5 cooperative priorities and 10 preemptive priorities results in the ranges -5 to -1 and 0 to 9, respectively.

Meta-IRQ Priorities When enabled (see `CONFIG_NUM_METAIRQ_PRIORITIES`), there is a special subclass of cooperative priorities at the highest (numerically lowest) end of the priority space: meta-IRQ threads. These are scheduled according to their normal priority, but also have the special ability to preempt all other threads (and other meta-IRQ threads) at lower priorities, even if those threads are cooperative and/or have taken a scheduler lock. Meta-IRQ threads are still threads, however, and can still be interrupted by any hardware interrupt.

> **Note**
>
> When a cooperative (or schedule-locked) thread that was preempted by a meta-IRQ thread resumes its execution after the meta-IRQ thread completes, it will be on the same CPU–assuming it was neither suspended nor aborted. This helps ensure that these threads are not unexpectedly shuffled to another CPU while in the midst of querying the properties associated with their own CPU.

This behavior makes the act of unblocking a meta-IRQ thread (by any means, e.g. creating it, calling k_sem_give(), etc.) into the equivalent of a synchronous system call when done by a lower priority thread, or an ARM-like "pended IRQ" when done from true interrupt context. The intent is that this feature will be used to implement interrupt "bottom half" processing and/or "tasklet" features in driver subsystems. The thread, once woken, will be guaranteed to run before the current CPU returns into application code.

Unlike similar features in other OSes, meta-IRQ threads are true threads and run on their own stack (which must be allocated normally), not the per-CPU interrupt stack. Design work to enable the use of the IRQ stack on supported architectures is pending.

Note that because this breaks the promise made to cooperative threads by the Zephyr API (namely that the OS won't schedule other thread until the current thread deliberately blocks), it should be used only with great care from application code. These are not simply very high priority threads and should not be used as such.

Thread Options The kernel supports a small set of *thread options* that allow a thread to receive special treatment under specific circumstances. The set of options associated with a thread are specified when the thread is spawned.

A thread that does not require any thread option has an option value of zero. A thread that requires a thread option specifies it by name, using the | character as a separator if multiple options are needed (i.e. combine options using the bitwise OR operator).

The following thread options are supported.

`K_ESSENTIAL`
: This option tags the thread as an *essential thread*. This instructs the kernel to treat the termination or aborting of the thread as a fatal system error.

  By default, the thread is not considered to be an essential thread.

`K_SSE_REGS`
: This x86-specific option indicate that the thread uses the CPU's SSE registers. Also see `K_FP_REGS`.

  By default, the kernel does not attempt to save and restore the contents of these registers when scheduling the thread.

`K_FP_REGS`
: This option indicate that the thread uses the CPU's floating point registers. This instructs the kernel to take additional steps to save and restore the contents of these registers when scheduling the thread. (For more information see *Floating Point Services*.)

By default, the kernel does not attempt to save and restore the contents of this register when scheduling the thread.

K_USER
: If CONFIG_USERSPACE is enabled, this thread will be created in user mode and will have reduced privileges. See *User Mode*. Otherwise this flag does nothing.

K_INHERIT_PERMS
: If CONFIG_USERSPACE is enabled, this thread will inherit all kernel object permissions that the parent thread had, except the parent thread object. See *User Mode*.

Thread Custom Data Every thread has a 32-bit *custom data* area, accessible only by the thread itself, and may be used by the application for any purpose it chooses. The default custom data value for a thread is zero.

> **Note**
>
> Custom data support is not available to ISRs because they operate within a single shared kernel interrupt handling context.

By default, thread custom data support is disabled. The configuration option CONFIG_THREAD_CUSTOM_DATA can be used to enable support.

The k_thread_custom_data_set() and k_thread_custom_data_get() functions are used to write and read a thread's custom data, respectively. A thread can only access its own custom data, and not that of another thread.

The following code uses the custom data feature to record the number of times each thread calls a specific routine.

> **Note**
>
> Obviously, only a single routine can use this technique, since it monopolizes the use of the custom data feature.

```
int call_tracking_routine(void)
{
    uint32_t call_count;

    if (k_is_in_isr()) {
        /* ignore any call made by an ISR */
    } else {
        call_count = (uint32_t)k_thread_custom_data_get();
        call_count++;
        k_thread_custom_data_set((void *)call_count);
    }

    /* do rest of routine's processing */
    ...
}
```

Use thread custom data to allow a routine to access thread-specific information, by using the custom data as a pointer to a data structure owned by the thread.

Thread Name When CONFIG_THREAD_NAME is enabled, a human-readable name can be associated with each thread. Names are primarily an aid for debugging, logging, and shell introspection; the kernel does not use them for scheduling.

A thread name may be assigned in one of two ways:

- Statically, by passing a name to K_THREAD_DEFINE.
- At run time, using k_thread_name_set(). The supplied string must remain valid for the lifetime of the thread, as only a pointer to it is retained.

A thread's name can be retrieved with k_thread_name_get(), which returns a pointer to the name string, or with k_thread_name_copy(), which copies the name into a caller-supplied buffer. The copying variant is the safe choice from user mode, where the calling thread may not have access to the memory backing another thread's name.

If CONFIG_THREAD_NAME is not enabled, k_thread_name_set() returns an error and k_thread_name_get() returns NULL.

Thread Introspection The kernel provides several interfaces for inspecting threads at run time.

Identifying the current thread
k_current_get() returns the thread id (k_tid_t) of the thread that is currently executing. This id can be passed to the other thread APIs to operate on the calling thread.

Reading a thread's priority
k_thread_priority_get() returns the current scheduling priority of a thread. This reflects any changes made since the thread was created, for example by k_thread_priority_set().

Obtaining a thread's state
k_thread_state_str() writes a human-readable representation of a thread's current state (for example pending, suspended, or ready) into a caller-supplied buffer. This is intended for diagnostic output and should not be parsed programmatically.

Iterating over all threads
When CONFIG_THREAD_MONITOR is enabled, k_thread_foreach() invokes a caller-supplied callback once for each thread in the system. It holds an internal lock that blocks thread creation and termination for the duration of the iteration, which guarantees a consistent snapshot of the thread list but can introduce latency proportional to the number of threads. k_thread_foreach_unlocked() is a lower-latency variant that releases the lock around each callback at the cost of a less strictly consistent view.

Querying stack usage
When CONFIG_INIT_STACKS and CONFIG_THREAD_STACK_INFO are enabled, k_thread_stack_space_get() reports the number of unused bytes remaining in a thread's stack, computed by scanning the stack for its unused (still-initialized) region. This complements the build-time stack analysis tools and the runtime stack safety feature described below.

Querying a pending timeout
A thread that is blocked on a timed operation (such as k_sleep() or a kernel object operation with a timeout) has a pending timeout. Its remaining duration can be queried in ticks with k_thread_timeout_remaining_ticks(), and the absolute system tick at which it will expire with k_thread_timeout_expires_ticks(). Both return zero if the thread has no pending timeout.

Implementation

Spawning a Thread A thread is spawned by defining its stack area and its thread control block, and then calling k_thread_create().

The stack area may be statically allocated using K_THREAD_STACK_DEFINE or K_KERNEL_STACK_DEFINE to ensure it is properly set up in memory.

The size parameter for the stack must be one of three values:

- The original requested stack size passed to K_THREAD_STACK or K_KERNEL_STACK family of stack instantiation macros.
- For a stack object defined with the K_THREAD_STACK family of macros, the return value of K_THREAD_STACK_SIZEOF() for that object.
- For a stack object defined with the K_KERNEL_STACK family of macros, the return value of K_KERNEL_STACK_SIZEOF() for that object.

Alternatively, the stack area may be dynamically allocated using k_thread_stack_alloc() and freed using k_thread_stack_free().

The thread spawning function returns its thread id, which can be used to reference the thread.

The following code spawns a thread that starts immediately.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);
struct k_thread my_thread_data;

k_tid_t my_tid = k_thread_create(&my_thread_data, my_stack_area,
                                 K_THREAD_STACK_SIZEOF(my_stack_area),
                                 my_entry_point,
                                 NULL, NULL, NULL,
                                 MY_PRIORITY, 0, K_NO_WAIT);
```

Alternatively, a thread can be declared at compile time by calling K_THREAD_DEFINE. Observe that the macro defines the stack area, control block, and thread id variables automatically.

The following code has the same effect as the code segment above.

```
#define MY_STACK_SIZE 500
#define MY_PRIORITY 5

extern void my_entry_point(void *, void *, void *);

K_THREAD_DEFINE(my_tid, MY_STACK_SIZE,
                my_entry_point, NULL, NULL, NULL,
                MY_PRIORITY, 0, 0);
```

Note

The delay parameter to k_thread_create() is a k_timeout_t value, so K_NO_WAIT means to start the thread immediately. The corresponding parameter to K_THREAD_DEFINE is a duration in integral milliseconds, so the equivalent argument is 0.

The following code dynamically allocates a thread stack, waits until the thread has joined, and then frees the dynamically allocated thread stack.

```
extern void my_entry_point(void *, void *, void *);

k_tid_t my_tid;
void *my_stack_area;

my_stack_area = k_thread_stack_alloc(CONFIG_DYNAMIC_THREAD_STACK_SIZE);
```

(continues on next page)

(continued from previous page)

```
my_tid = k_thread_create(&my_thread_data, my_stack_area,
                         CONFIG_DYNAMIC_THREAD_STACK_SIZE,
                         my_entry_point,
                         NULL, NULL, NULL,
                         MY_PRIORITY, 0, K_NO_WAIT);
k_thread_join(my_tid, K_FOREVER);
k_thread_stack_free(my_stack_area);
```

User Mode Constraints This section only applies if CONFIG_USERSPACE is enabled, and a user thread tries to create a new thread. The k_thread_create() API is still used, but there are additional constraints which must be met or the calling thread will be terminated:

- The calling thread must have permissions granted on both the child thread and stack parameters; both are tracked by the kernel as kernel objects.
- The child thread and stack objects must be in an uninitialized state, i.e. it is not currently running and the stack memory is unused.
- The stack size parameter passed in must be equal to or less than the bounds of the stack object when it was declared.
- The K_USER option must be used, as user threads can only create other user threads.
- The K_ESSENTIAL option must not be used, user threads may not be considered essential threads.
- The priority of the child thread must be a valid priority value, and equal to or lower than the parent thread.

Dropping Permissions If CONFIG_USERSPACE is enabled, a thread running in supervisor mode may perform a one-way transition to user mode using the k_thread_user_mode_enter() API. This is a one-way operation which will reset and zero the thread's stack memory. The thread will be marked as non-essential.

Terminating a Thread A thread terminates itself by returning from its entry point function.

The following code illustrates the ways a thread can terminate.

```
void my_entry_point(int unused1, int unused2, int unused3)
{
    while (1) {
        ...
        if (<some condition>) {
            return; /* thread terminates from mid-entry point function */
        }
        ...
    }

    /* thread terminates at end of entry point function */
}
```

If CONFIG_USERSPACE is enabled, aborting a thread will additionally mark the thread and stack objects as uninitialized so that they may be reused.

Runtime Statistics Thread runtime statistics can be gathered and retrieved if CONFIG_THREAD_RUNTIME_STATS is enabled, for example, total number of execution cycles of a thread.

By default, the runtime statistics are gathered using the default kernel timer. For some architectures, SoCs or boards, there are timers with higher resolution available via timing functions. Using of these timers can be enabled via CONFIG_THREAD_RUNTIME_STATS_USE_TIMING_FUNCTIONS.

Here is an example:

```
k_thread_runtime_stats_t rt_stats_thread;

k_thread_runtime_stats_get(k_current_get(), &rt_stats_thread);

printk("Cycles: %llu\n", rt_stats_thread.execution_cycles);
```

The combined statistics for every thread in the system can be retrieved with k_thread_runtime_stats_all_get(), which reports the aggregate runtime usage across all threads (including the idle thread). This is useful for computing overall CPU utilization.

When CONFIG_SCHED_THREAD_USAGE is enabled, statistics collection can be toggled at run time on a per-thread basis with k_thread_runtime_stats_enable() and k_thread_runtime_stats_disable(). Whether collection is currently active for a thread can be checked with k_thread_runtime_stats_is_enabled(). Disabling collection for threads that do not need to be measured reduces the bookkeeping overhead incurred on each context switch.

Runtime Stack Safety When CONFIG_THREAD_RUNTIME_STACK_SAFETY is enabled, the kernel provides routines that scan a thread's stack at runtime to determine how much of it remains unused. If the amount of unused stack space is found to have dropped below a configurable per-thread threshold, a user-defined handler is invoked.

This feature is intended for use by monitoring software. The handler may, for example, log a warning, suspend or abort the offending thread, or even reboot the system. It complements the build-time stack analysis tools and the hardware-based stack overflow detection by allowing a system to react *before* a stack is exhausted.

Each thread has an *unused stack threshold*, expressed in bytes. When a stack safety check finds that a thread's unused stack space is below this threshold, the supplied handler is called. A threshold of 0 bytes (the default) disables the check for that thread. The default threshold applied to newly created threads is derived from CONFIG_THREAD_RUNTIME_STACK_SAFETY_DEFAULT_UNUSED_THRESHOLD_PCT, which expresses it as a percentage of each thread's total stack size.

The threshold for an individual thread can be set or queried at runtime:

- k_thread_runtime_stack_unused_threshold_pct_set() sets the threshold as a percentage (0 to 99) of the thread's total stack size.
- k_thread_runtime_stack_unused_threshold_set() sets the threshold as an absolute number of bytes.
- k_thread_runtime_stack_unused_threshold_get() retrieves the current threshold (in bytes).

Two routines perform the actual check. Both accept a pointer that, on return, receives the amount of unused stack space, and a k_thread_stack_safety_handler_t handler (plus a user argument) that is invoked if the threshold has been crossed:

- k_thread_runtime_stack_safety_full_check() scans the entire stack to compute the exact amount of unused space.
- k_thread_runtime_stack_safety_threshold_check() performs an abbreviated scan that only looks for evidence that the thread has crossed its configured threshold. This is cheaper than a full check but does not yield an exact measure of unused space.

Here is an example that configures a thread to invoke a handler once its unused stack space drops below 10% of its total stack size:

```
void stack_safety_handler(const struct k_thread *thread,
                          size_t unused_space, void *arg)
{
        printk("Thread %p low on stack: %zu bytes unused\n",
               thread, unused_space);
}

/* Trigger the handler once less than 10% of the stack remains unused */
k_thread_runtime_stack_unused_threshold_pct_set(my_tid, 10);

/* Periodically check the thread from a monitoring context */
k_thread_runtime_stack_safety_full_check(my_tid, NULL,
                                         stack_safety_handler, NULL);
```

Suggested Uses Use threads to handle processing that cannot be handled in an ISR.

Use separate threads to handle logically distinct processing operations that can execute in parallel.

Configuration Options Related configuration options:

- CONFIG_MAIN_THREAD_PRIORITY
- CONFIG_MAIN_STACK_SIZE
- CONFIG_IDLE_STACK_SIZE
- CONFIG_THREAD_CUSTOM_DATA
- CONFIG_NUM_COOP_PRIORITIES
- CONFIG_NUM_PREEMPT_PRIORITIES
- CONFIG_TIMESLICING
- CONFIG_TIMESLICE_SIZE
- CONFIG_TIMESLICE_PRIORITY
- CONFIG_USERSPACE
- CONFIG_THREAD_RUNTIME_STACK_SAFETY
- CONFIG_THREAD_RUNTIME_STACK_SAFETY_DEFAULT_UNUSED_THRESHOLD_PCT

API Reference

Thread APIs

Related code samples

- Basic SynchronizationManipulate basic kernel synchronization primitives.
- Basic thread manipulationSpawn multiple threads that blink LEDs and print information to the console.
- Dining PhilosophersImplement a solution to the Dining Philosophers problem using Zephyr kernel services.
- Dumb HTTP server (multi-threaded)Implement a simple HTTP server supporting simultaneous connections using BSD sockets.

Thread Stack APIs

Scheduling

The kernel's priority-based scheduler allows an application's threads to share the CPU.

Concepts The scheduler determines which thread is allowed to execute at any point in time; this thread is known as the **current thread**.

There are various points in time when the scheduler is given an opportunity to change the identity of the current thread, meaning when the scheduler switches the CPU's execution from one thread to another. These points are called **reschedule points**. Some potential reschedule points are:

- transition of a thread from running state to a suspended or waiting state, for example by `k_sem_take()` or `k_sleep()`.
- transition of a thread to the *ready state*, for example by `k_sem_give()` or `k_thread_start()`
- return to thread context after processing an interrupt
- when a running thread invokes `k_yield()`

A thread **sleeps** when it voluntarily initiates an operation that transitions itself to a suspended or waiting state.

Whenever the scheduler changes the identity of the current thread, or when execution of the current thread is replaced by an ISR, the kernel first saves the current thread's CPU register values. These register values get restored when the thread later resumes execution.

Scheduling Algorithm The kernel's scheduler selects the highest priority ready thread to be the current thread. When multiple ready threads of the same priority exist, the scheduler chooses the one that has been waiting longest.

A thread's relative priority is primarily determined by its static priority. However, when both earliest-deadline-first scheduling is enabled (`CONFIG_SCHED_DEADLINE`) and a choice of threads have equal static priority, then the thread with the earlier deadline is considered to have the higher priority. Thus, when earliest-deadline-first scheduling is enabled, two threads are only considered to have the same priority when both their static priorities and deadlines are equal. The routine `k_thread_deadline_set()` is used to set a thread's deadline.

> **Note**
>
> Execution of ISRs takes precedence over thread execution, so the execution of the current thread may be replaced by an ISR at any time unless interrupts have been masked. This applies to both cooperative threads and preemptive threads.

The kernel can be built with one of several choices for the ready queue implementation, offering different choices between code size, constant factor runtime overhead and performance scaling when many threads are added.

- Simple linked-list ready queue (`CONFIG_SCHED_SIMPLE`)

  The scheduler ready queue will be implemented as a simple unordered list, with very fast constant time performance for single threads and very low code size. This implementation should be selected on systems with constrained code size that will never see more than a small number (3, maybe) of runnable threads in the queue at any given time. On most platforms (that are not otherwise using the red/black tree) this results in a savings of ~2k of code size.

- Red/black tree ready queue (`CONFIG_SCHED_SCALABLE`)

  The scheduler ready queue will be implemented as a red/black tree. This has rather slower constant-time insertion and removal overhead, and on most platforms (that are not oth-

erwise using the red/black tree somewhere) requires an extra ~2kb of code. The resulting behavior will scale cleanly and quickly into the many thousands of threads.

Use this for applications needing many concurrent runnable threads (> 20 or so). Most applications won't need this ready queue implementation.

- Traditional multi-queue ready queue (`CONFIG_SCHED_MULTIQ`)

  When selected, the scheduler ready queue will be implemented as the classic/textbook array of lists, one per priority.

  This corresponds to the scheduler algorithm used in Zephyr versions prior to 1.12.

  It incurs only a tiny code size overhead vs. the "dumb" scheduler and runs in O(1) time in almost all circumstances with very low constant factor. But it requires a fairly large RAM budget to store those list heads, and the limited features make it incompatible with features like deadline scheduling that need to sort threads more finely, and SMP affinity which need to traverse the list of threads.

  Typical applications with small numbers of runnable threads probably want the simple scheduler.

The wait_q abstraction used in IPC primitives to pend threads for later wakeup shares the same backend data structure choices as the scheduler, and can use the same options.

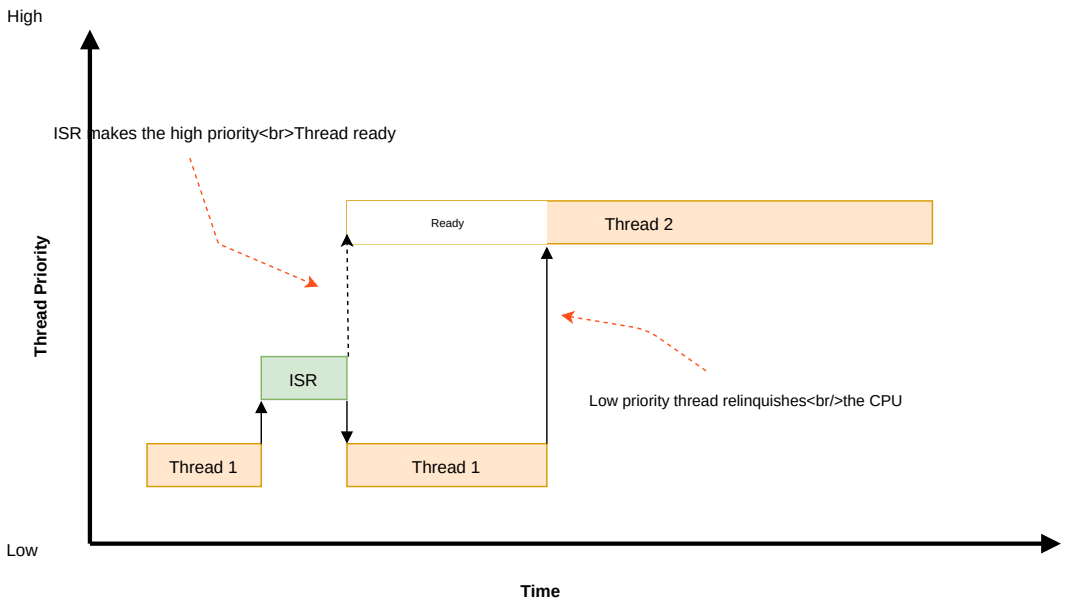
- Scalable wait_q implementation (`CONFIG_WAITQ_SCALABLE`)

  When selected, the wait_q will be implemented with a balanced tree. Choose this if you expect to have many threads waiting on individual primitives. There is a ~2kb code size increase over `CONFIG_WAITQ_SIMPLE` (which may be shared with `CONFIG_SCHED_SCALABLE`) if the red/black tree is not used elsewhere in the application, and pend/unpend operations on "small" queues will be somewhat slower (though this is not generally a performance path).

- Simple linked-list wait_q (`CONFIG_WAITQ_SIMPLE`)

  When selected, the wait_q will be implemented with a doubly-linked list. Choose this if you expect to have only a few threads blocked on any single IPC primitive.

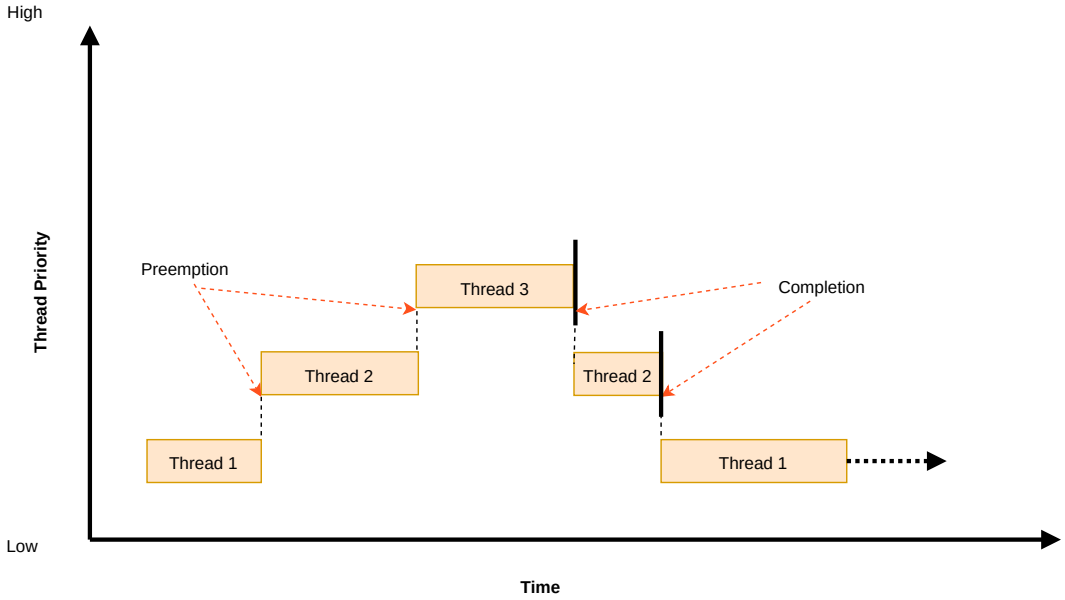
Cooperative Time Slicing Once a cooperative thread becomes the current thread, it remains the current thread until it performs an action that makes it unready. Consequently, if a cooperative thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of higher priority and equal priority.



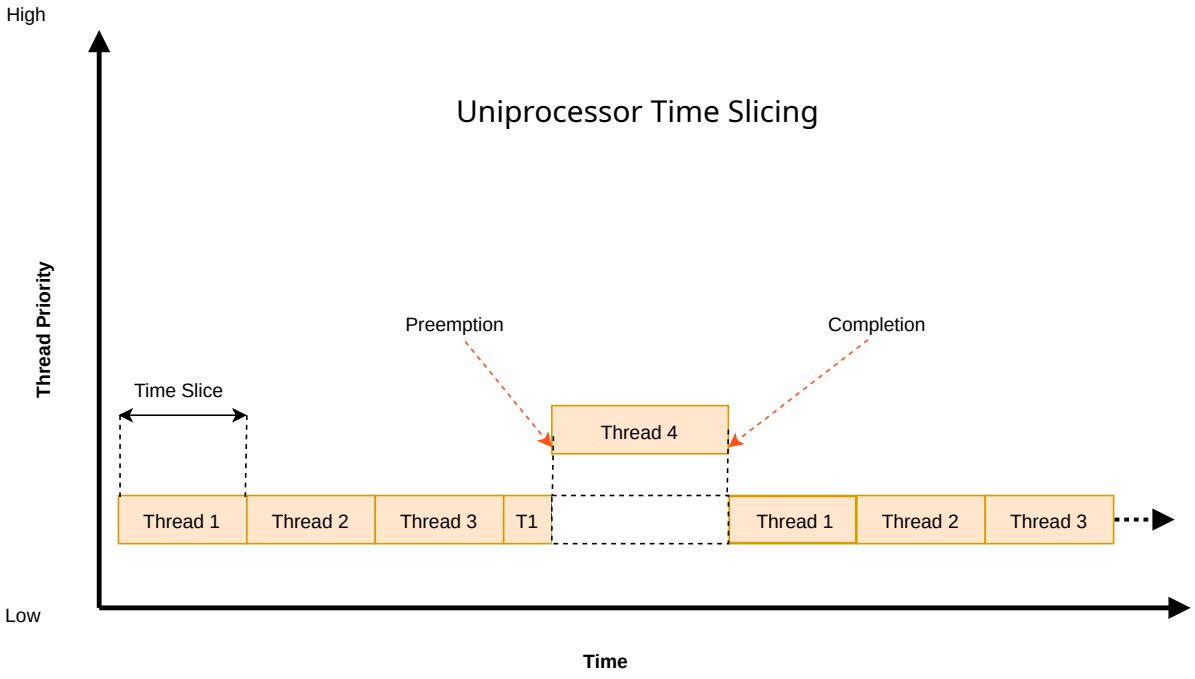
To overcome such problems, a cooperative thread can voluntarily relinquish the CPU from time to time to permit other threads to execute. A thread can relinquish the CPU in two ways:

- Calling `k_yield()` puts the thread at the back of the scheduler's prioritized list of ready threads, and then invokes the scheduler. All ready threads whose priority is higher or equal to that of the yielding thread are then allowed to execute before the yielding thread is rescheduled. If no such ready threads exist, the scheduler immediately reschedules the yielding thread without context switching.
- Calling `k_sleep()` makes the thread unready for a specified time period. Ready threads of *all* priorities are then allowed to execute; however, there is no guarantee that threads whose priority is lower than that of the sleeping thread will actually be scheduled before the sleeping thread becomes ready once again.

Preemptive Time Slicing Once a preemptive thread becomes the current thread, it remains the current thread until a higher priority thread becomes ready, or until the thread performs an action that makes it unready. Consequently, if a preemptive thread performs lengthy computations, it may cause an unacceptable delay in the scheduling of other threads, including those of equal priority.



To overcome such problems, a preemptive thread can perform cooperative time slicing (as described above), or the scheduler's time slicing capability can be used to allow other threads of the same priority to execute.



Note

For SMP, the behavior would be similar to the above UP image, except that Thread 1 would not immediately follow Thread 4. Instead it would be Thread 2, Thread 3, Thread 1, ...

The scheduler divides time on each CPU into a series of **time slices**, where slices are measured in system clock ticks. The time slice size is configurable, but this size can be changed while the application is running. Scheduling a new thread causes the time slice timer to be reset.

At the end of every time slice, the scheduler checks to see if the current thread is preemptible and, if so, implicitly invokes `k_yield()` on behalf of the thread. This gives other ready threads of the same priority the opportunity to execute before the current thread is scheduled again. If no other threads of equal priority are ready, then the current thread remains the current thread.

Threads with a priority higher than specified limit are exempt from preemptive time slicing, and are never preempted by a thread of equal priority. This allows an application to use preemptive time slicing only when dealing with lower priority threads that are less time-sensitive.

Note

The kernel's time slicing algorithm does *not* ensure that a set of equal-priority threads receive an equitable amount of CPU time, since it does not measure the amount of time a thread actually gets to execute. However, the algorithm *does* ensure that a thread never executes for longer than a single time slice without being required to yield.

Per-Thread Time Slicing The time slice configured with `k_sched_time_slice_set()` applies globally to every preemptible thread at or below a given priority. When `CONFIG_TIMESLICE_PER_THREAD` is enabled, an individual thread can instead be given its own time slice with `k_thread_time_slice_set()`. A per-thread slice takes precedence over the global value, and is applied even to threads whose priority is above the global time-slicing limit.

In addition to a slice duration (expressed in ticks), a per-thread slice registers a callback that the kernel invokes when the slice expires. This callback runs in interrupt context while the affected thread is still the current thread, and may, for example, adjust the thread's priority or slice for

its next execution, or suspend it.

Scheduler Locking A preemptible thread that does not wish to be preempted while performing a critical operation can instruct the scheduler to temporarily treat it as a cooperative thread by calling `k_sched_lock()`. This prevents other threads from interfering while the critical operation is being performed.

Once the critical operation is complete the preemptible thread must call `k_sched_unlock()` to restore its normal, preemptible status.

If a thread calls `k_sched_lock()` and subsequently performs an action that makes it unready, the scheduler will switch the locking thread out and allow other threads to execute. When the locking thread again becomes the current thread, its non-preemptible status is maintained.

> **Note**
>
> Locking out the scheduler is a more efficient way for a preemptible thread to prevent preemption than changing its priority level to a negative value.

Thread Sleeping A thread can call `k_sleep()` to delay its processing for a specified time period. During the time the thread is sleeping the CPU is relinquished to allow other ready threads to execute. Once the specified delay has elapsed the thread becomes ready and is eligible to be scheduled once again.

A sleeping thread can be woken up prematurely by another thread using `k_wakeup()`. This technique can sometimes be used to permit the secondary thread to signal the sleeping thread that something has occurred *without* requiring the threads to define a kernel synchronization object, such as a semaphore. Waking up a thread that is not sleeping is allowed, but has no effect.

Busy Waiting A thread can call `k_busy_wait()` to perform a busy `wait` that delays its processing for a specified time period *without* relinquishing the CPU to another ready thread.

A busy wait is typically used instead of thread sleeping when the required delay is too short to warrant having the scheduler context switch from the current thread to another thread and then back again.

Forcing a Scheduling Decision A thread can force the scheduler to make an immediate scheduling decision on the current CPU by calling `k_reschedule()`. When invoked from a thread (with interrupts unlocked) the scheduler runs immediately; when invoked from an ISR the decision is deferred until the ISR exits.

Unlike `k_yield()`, this routine does not guarantee a switch to a thread of equal or higher priority – it simply asks the kernel to re-evaluate which thread should run next given the current state. Most applications never need this routine.

Querying Preemptibility Code whose behavior depends on whether it can be preempted can query the current context with `k_is_preempt_thread()`. This returns a non-zero value only when the caller is a thread (not an ISR), the thread's priority is in the preemptible range, and the thread has not locked the scheduler.

The related `k_can_yield()` routine reports whether the current context is able to yield or invoke blocking APIs at all. It returns false in contexts such as ISRs, the pre-kernel initialization phase, or the idle thread, where yielding is not possible.

Suggested Uses Use cooperative threads for device drivers and other performance-critical work.

Use cooperative threads to implement mutual exclusion without the need for a kernel object, such as a mutex.

Use preemptive threads to give priority to time-sensitive processing over less time-sensitive processing.

Configuration Options

- CONFIG_TIMESLICING
- CONFIG_TIMESLICE_SIZE
- CONFIG_TIMESLICE_PRIORITY

CPU Idling

Although normally reserved for the idle thread, in certain special applications, a thread might want to make the CPU idle.

- *Concepts*
- *Implementation*
  - *Making the CPU idle*
  - *Making the CPU idle in an atomic fashion*
- *Suggested Uses*
- *API Reference*

Concepts Making the CPU idle causes the kernel to pause all operations until an event, normally an interrupt, wakes up the CPU. In a regular system, the idle thread is responsible for this. However, in some constrained systems, it is possible that another thread takes this duty.

Implementation

Making the CPU idle Making the CPU idle is simple: call the `k_cpu_idle()` API. The CPU will stop executing instructions until an event occurs. Most likely, the function will be called within a loop. Note that in certain architectures, upon return, `k_cpu_idle()` unconditionally unmasks interrupts.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

int main(void)
{
    k_sem_init(&my_sem, 0, 1);

    /* wait for semaphore from ISR, then do related work */

    for (;;) {

        /* wait for ISR to trigger work to perform */
```

(continues on next page)

(continued from previous page)

```
        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            /* ... do processing */

        }

        /* put CPU to sleep to save power */
        k_cpu_idle();
    }
}
```

Making the CPU idle in an atomic fashion It is possible that there is a need to do some work atomically before making the CPU idle. In such a case, `k_cpu_atomic_idle()` should be used instead.

In fact, there is a race condition in the previous example: the interrupt could occur between the time the semaphore is taken, finding out it is not available and making the CPU idle again. In some systems, this can cause the CPU to idle until *another* interrupt occurs, which might be *never*, thus hanging the system completely. To prevent this, `k_cpu_atomic_idle()` should have been used, like in this example.

```
static k_sem my_sem;

void my_isr(void *unused)
{
    k_sem_give(&my_sem);
}

int main(void)
{
    k_sem_init(&my_sem, 0, 1);

    for (;;) {

        unsigned int key = irq_lock();

        /*
         * Wait for semaphore from ISR; if acquired, do related work, then
         * go to next loop iteration (the semaphore might have been given
         * again); else, make the CPU idle.
         */

        if (k_sem_take(&my_sem, K_NO_WAIT) == 0) {

            irq_unlock(key);

            /* ... do processing */


        } else {
            /* put CPU to sleep to save power */
            k_cpu_atomic_idle(key);
        }
    }
}
```

Suggested Uses Use `k_cpu_atomic_idle()` when a thread has to do some real work in addition to idling the CPU to wait for an event. See example above.

Use `k_cpu_idle()` only when a thread is only responsible for idling the CPU, i.e. not doing any real work, like in this example below.

```
int main(void)
{
    /* ... do some system/application initialization */


    /* thread is only used for CPU idling from this point on */
    for (;;) {
        k_cpu_idle();
    }
}
```

> **Note**
>
> **Do not use these APIs unless absolutely necessary.** In a normal system, the idle thread takes care of power management, including CPU idling.

API Reference
CPU Idling APIs

System Threads

- *Implementation*
  - *Writing a main() function*
- *Suggested Uses*

A *system thread* is a thread that the kernel spawns automatically during system initialization.

The kernel spawns the following system threads:

Main thread
This thread performs kernel initialization, then calls the application's `main()` function (if one is defined).

By default, the main thread uses the highest configured preemptible thread priority (i.e. 0). If the kernel is not configured to support preemptible threads, the main thread uses the lowest configured cooperative thread priority (i.e. -1).

The main thread is an essential thread while it is performing kernel initialization or executing the application's `main()` function; this means a fatal system error is raised if the thread aborts. If `main()` is not defined, or if it executes and then does a normal return, the main thread terminates normally and no error is raised.

Idle thread
This thread executes when there is no other work for the system to do. If possible, the idle thread activates the board's power management support to save power; otherwise, the idle thread simply performs a "do nothing" loop. The idle thread remains in existence as long as the system is running and never terminates.

The idle thread always uses the lowest configured thread priority.

The idle thread is an essential thread, which means a fatal system error is raised if the thread aborts.

Additional system threads may also be spawned, depending on the kernel and board configuration options specified by the application. For example, enabling the system workqueue spawns a system thread that services the work items submitted to it. (See *Workqueue Threads*.)

Implementation

Writing a main() function An application-supplied `main()` function begins executing once kernel initialization is complete. The kernel does not pass any arguments to the function, unless `CONFIG_BOOTARGS` is selected. In such case the kernel passes arguments to it and `main(int, char **)` can be used.

The following code outlines a trivial `main(void)` function. The function used by a real application can be as complex as needed.

```
int main(void)
{
    /* initialize a semaphore */
    ...

    /* register an ISR that gives the semaphore */
    ...

    /* monitor the semaphore forever */
    while (1) {
        /* wait for the semaphore to be given by the ISR */
        ...
        /* do whatever processing is now needed */
        ...
    }
}
```

Suggested Uses Use the main thread to perform thread-based processing in an application that only requires a single thread, rather than defining an additional application-specific thread.

Workqueue Threads

- *Work Item Lifecycle*
- *Delayable Work*
- *Triggered Work*
- *System Workqueue*
- *How to Use Workqueues*
- *Workqueue Best Practices*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

A *workqueue* is a kernel object that uses a dedicated thread to process work items in a first in, first out manner. Each work item is processed by calling the function specified by the work item. A

workqueue is typically used by an ISR or a high-priority thread to offload non-urgent processing to a lower-priority thread so it does not impact time-sensitive processing.

Any number of workqueues can be defined (limited only by available RAM). Each workqueue is referenced by its memory address.

A workqueue has the following key properties:

- A **queue** of work items that have been added, but not yet processed.
- A **thread** that processes the work items in the queue. The priority of the thread is configurable, allowing it to be either cooperative or preemptive as required.

Regardless of workqueue thread priority the workqueue thread will yield between each submitted work item, to prevent a cooperative workqueue from starving other threads.

A workqueue must be initialized before it can be used. This sets its queue to empty and spawns the workqueue's thread. The thread runs forever, but sleeps when no work items are available.

Work Item Lifecycle Any number of **work items** can be defined. Each work item is referenced by its memory address.

A work item is assigned a **handler function**, which is the function executed by the workqueue's thread when the work item is processed. This function accepts a single argument, which is the address of the work item itself. The work item also maintains information about its status.

A work item must be initialized before it can be used. This records the work item's handler function and marks it as not pending.

A work item may be **queued** (`K_WORK_QUEUED`) by submitting it to a workqueue by an ISR or a thread. Submitting a work item appends the work item to the workqueue's queue. Once the workqueue's thread has processed all of the preceding work items in its queue the thread will remove the next work item from the queue and invoke the work item's handler function. Depending on the scheduling priority of the workqueue's thread, and the work required by other items in the queue, a queued work item may be processed quickly or it may remain in the queue for an extended period of time.

A delayable work item may be **scheduled** (`K_WORK_DELAYED`) to a workqueue; see *Delayable Work*.

A work item will be **running** (`K_WORK_RUNNING`) when it is running on a work queue, and may also be **canceling** (`K_WORK_CANCELING`) if it started running before a thread has requested that it be cancelled.

A work item can be in multiple states; for example it can be:

- running on a queue;
- marked canceling (because a thread used `k_work_cancel_sync()` to wait until the work item completed);
- queued to run again on the same queue;
- scheduled to be submitted to a (possibly different) queue

all simultaneously. A work item that is in any of these states is **pending** (`k_work_is_pending()`) or **busy** (`k_work_busy_get()`).

A handler function can use any kernel API available to threads. However, operations that are potentially blocking (e.g. taking a semaphore) must be used with care, since the workqueue cannot process subsequent work items in its queue until the handler function finishes executing.

The single argument that is passed to a handler function can be ignored if it is not required. If the handler function requires additional information about the work it is to perform, the work item can be embedded in a larger data structure. The handler function can then use the argument value to compute the address of the enclosing data structure with `CONTAINER_OF`, and thereby obtain access to the additional information it needs.

A work item is typically initialized once and then submitted to a specific workqueue whenever work needs to be performed. If an ISR or a thread attempts to submit a work item that is already queued the work item is not affected; the work item remains in its current place in the workqueue's queue, and the work is only performed once.

A handler function is permitted to re-submit its work item argument to the workqueue, since the work item is no longer queued at that time. This allows the handler to execute work in stages, without unduly delaying the processing of other work items in the workqueue's queue.

> **Important**
>
> A pending work item *must not* be altered until the item has been processed by the workqueue thread. This means a work item must not be re-initialized while it is busy. Furthermore, any additional information the work item's handler function needs to perform its work must not be altered until the handler function has finished executing.

Delayable Work An ISR or a thread may need to schedule a work item that is to be processed only after a specified period of time, rather than immediately. This can be done by **scheduling** a **delayable work item** to be submitted to a workqueue at a future time.

A delayable work item contains a standard work item but adds fields that record when and where the item should be submitted.

A delayable work item is initialized and scheduled to a workqueue in a similar manner to a standard work item, although different kernel APIs are used. When the schedule request is made the kernel initiates a timeout mechanism that is triggered after the specified delay has elapsed. Once the timeout occurs the kernel submits the work item to the specified workqueue, where it remains queued until it is processed in the standard manner.

Note that work handler used for delayable still receives a pointer to the underlying non-delayable work structure, which is not publicly accessible from `k_work_delayable`. To get access to an object that contains the delayable work object use this idiom:

```
static void work_handler(struct k_work *work)
{
        struct k_work_delayable *dwork = k_work_delayable_from_work(work);
        struct work_context *ctx = CONTAINER_OF(dwork, struct work_context,
                                                timed_work);
        ...
```

Triggered Work The `k_work_poll_submit()` interface schedules a triggered work item in response to a **poll event** (see *Polling API*), that will call a user-defined function when a monitored resource becomes available or poll signal is raised, or a timeout occurs. In contrast to `k_poll()`, the triggered work does not require a dedicated thread waiting or actively polling for a poll event.

A triggered work item is a standard work item that has the following added properties:

- A pointer to an array of poll events that will trigger work item submissions to the workqueue
- A size of the array containing poll events.

A triggered work item is initialized and submitted to a workqueue in a similar manner to a standard work item, although dedicated kernel APIs are used. When a submit request is made, the kernel begins observing kernel objects specified by the poll events. Once at least one of the observed kernel object's changes state, the work item is submitted to the specified workqueue, where it remains queued until it is processed in the standard manner.

Important

The triggered work item as well as the referenced array of poll events have to be valid and cannot be modified for a complete triggered work item lifecycle, from submission to work item execution or cancellation.

An ISR or a thread may **cancel** a triggered work item it has submitted as long as it is still waiting for a poll event. In such case, the kernel stops waiting for attached poll events and the specified work is not executed. Otherwise the cancellation cannot be performed.

System Workqueue The kernel defines a workqueue known as the *system workqueue*, which is available to any application or kernel code that requires workqueue support. The system workqueue is optional, and only exists if the application makes use of it.

Important

Additional workqueues should only be defined when it is not possible to submit new work items to the system workqueue, since each new workqueue incurs a significant cost in memory footprint. A new workqueue can be justified if it is not possible for its work items to co-exist with existing system workqueue work items without an unacceptable impact; for example, if the new work items perform blocking operations that would delay other system workqueue processing to an unacceptable degree.

Also note: The system workqueue has the lowest cooperative priority by default. This means that draining the system workqueue takes precedence over *all preemptible threads*. Configuring the system workqueue's priority to be preemptible is risky, because subsystems and drivers may implicitly rely on work items being executed in a cooperative context. Therefore, a separate workqueue with lower priority may be appropriate for low-priority tasks.

How to Use Workqueues

Defining and Controlling a Workqueue A workqueue is defined using a variable of type `k_work_q`. The workqueue is initialized by defining the stack area used by its thread, initializing the `k_work_q`, either zeroing its memory or calling `k_work_queue_init()`, and then calling `k_work_queue_start()`. The stack area must be defined using `K_THREAD_STACK_DEFINE` to ensure it is properly set up in memory.

The following code defines and initializes a workqueue:

```
#define MY_STACK_SIZE 512
#define MY_PRIORITY 5

K_THREAD_STACK_DEFINE(my_stack_area, MY_STACK_SIZE);

struct k_work_q my_work_q;

k_work_queue_init(&my_work_q);

k_work_queue_start(&my_work_q, my_stack_area,
                   K_THREAD_STACK_SIZEOF(my_stack_area), MY_PRIORITY,
                   NULL);
```

In addition the queue identity and certain behavior related to thread rescheduling can be controlled by the optional final parameter; see `k_work_queue_start()` for details.

The following API can be used to interact with a workqueue:

- k_work_queue_drain() can be used to block the caller until the work queue has no items left. Work items resubmitted from the workqueue thread are accepted while a queue is draining, but work items from any other thread or ISR are rejected. The restriction on submitting more work can be extended past the completion of the drain operation in order to allow the blocking thread to perform additional work while the queue is "plugged". Note that draining a queue has no effect on scheduling or processing delayable items, but if the queue is plugged and the deadline expires the item will silently fail to be submitted.
- k_work_queue_unplug() removes any previous block on submission to the queue due to a previous drain operation.

Submitting a Work Item A work item is defined using a variable of type k_work. It must be initialized by calling k_work_init(), unless it is defined using K_WORK_DEFINE in which case initialization is performed at compile-time.

An initialized work item can be submitted to the system workqueue by calling k_work_submit(), or to a specified workqueue by calling k_work_submit_to_queue().

The following code demonstrates how an ISR can offload the printing of error messages to the system workqueue. Note that if the ISR attempts to resubmit the work item while it is still queued, the work item is left unchanged and the associated error message will not be printed.

```
struct device_info {
    struct k_work work;
    char name[16]
} my_device;

void my_isr(void *arg)
{
    ...
    if (error detected) {
        k_work_submit(&my_device.work);
    }
    ...
}

void print_error(struct k_work *item)
{
    struct device_info *the_device =
        CONTAINER_OF(item, struct device_info, work);
    printk("Got error on device %s\n", the_device->name);
}

/* initialize name info for a device */
strcpy(my_device.name, "FOO_dev");

/* initialize work item for printing device's error messages */
k_work_init(&my_device.work, print_error);

/* install my_isr() as interrupt handler for the device (not shown) */
...
```

The following API can be used to check the status of or synchronize with the work item:

- k_work_busy_get() returns a snapshot of flags indicating work item state. A zero value indicates the work is not scheduled, submitted, being executed, or otherwise still being referenced by the workqueue infrastructure.
- k_work_is_pending() is a helper that indicates true if and only if the work is scheduled, queued, or running.

- k_work_flush() may be invoked from threads to block until the work item has completed. It returns immediately if the work is not pending.
- k_work_cancel() attempts to prevent the work item from being executed. This may or may not be successful. This is safe to invoke from ISRs.
- k_work_cancel_sync() may be invoked from threads to block until the work completes; it will return immediately if the cancellation was successful or not necessary (the work wasn't submitted or running). This can be used after k_work_cancel() is invoked (from an ISR) to confirm completion of an ISR-initiated cancellation.

Scheduling a Delayable Work Item A delayable work item is defined using a variable of type k_work_delayable. It must be initialized by calling k_work_init_delayable().

For delayed work there are two common use cases, depending on whether a deadline should be extended if a new event occurs. An example is collecting data that comes in asynchronously, e.g. characters from a UART associated with a keyboard. There are two APIs that submit work after a delay:

- k_work_schedule() (or k_work_schedule_for_queue()) schedules work to be executed at a specific time or after a delay. Further attempts to schedule the same item with this API before the delay completes will not change the time at which the item will be submitted to its queue. Use this if the policy is to keep collecting data until a specified delay since the **first** unprocessed data was received;
- k_work_reschedule() (or k_work_reschedule_for_queue()) unconditionally sets the deadline for the work, replacing any previous incomplete delay and changing the destination queue if necessary. Use this if the policy is to keep collecting data until a specified delay since the **last** unprocessed data was received.

If the work item is not scheduled both APIs behave the same. If K_NO_WAIT is specified as the delay the behavior is as if the item was immediately submitted directly to the target queue, without waiting for a minimal timeout (unless k_work_schedule() is used and a previous delay has not completed).

Both also have variants that allow control of the queue used for submission.

The helper function k_work_delayable_from_work() can be used to get a pointer to the containing k_work_delayable from a pointer to k_work that is passed to a work handler function.

The following additional API can be used to check the status of or synchronize with the work item:

- k_work_delayable_busy_get() is the analog to k_work_busy_get() for delayable work.
- k_work_delayable_is_pending() is the analog to k_work_is_pending() for delayable work.
- k_work_flush_delayable() is the analog to k_work_flush() for delayable work.
- k_work_cancel_delayable() is the analog to k_work_cancel() for delayable work; similarly with k_work_cancel_delayable_sync().

Synchronizing with Work Items While the state of both regular and delayable work items can be determined from any context using k_work_busy_get() and k_work_delayable_busy_get() some use cases require synchronizing with work items after they've been submitted. k_work_flush(), k_work_cancel_sync(), and k_work_cancel_delayable_sync() can be invoked from thread context to wait until the requested state has been reached.

These APIs must be provided with a k_work_sync object that has no application-inspectable components but is needed to provide the synchronization objects. These objects should not be allocated on a stack if the code is expected to work on architectures with CONFIG_KERNEL_COHERENCE.

Workqueue Best Practices

Avoid Race Conditions Sometimes the data a work item must process is naturally thread-safe, for example when it's put into a `k_queue` by some thread and processed in the work thread. More often external synchronization is required to avoid data races: cases where the work thread might inspect or manipulate shared state that's being accessed by another thread or interrupt. Such state might be a flag indicating that work needs to be done, or a shared object that is filled by an ISR or thread and read by the work handler.

For simple flags *Atomic Services* may be sufficient. In other cases spin locks (`k_spinlock`) or thread-aware locks (`k_sem`, `k_mutex` , ...) may be used to ensure data races don't occur.

If the selected lock mechanism can *sleep* then allowing the work thread to sleep will starve other work queue items, which may need to make progress in order to get the lock released. Work handlers should try to take the lock with its no-wait path. For example:

```
static void work_handler(struct work *work)
{
        struct work_context *parent = CONTAINER_OF(work, struct work_context,
                                                   work_item);

        if (k_mutex_lock(&parent->lock, K_NO_WAIT) != 0) {
                /* NB: Submit will fail if the work item is being cancelled. */
                (void)k_work_submit(work);
                return;
        }

        /* do stuff under lock */
        k_mutex_unlock(&parent->lock);
        /* do stuff without lock */
}
```

Be aware that if the lock is held by a thread with a lower priority than the work queue the resubmission may starve the thread that would release the lock, causing the application to fail. Where the idiom above is required a delayable work item is preferred, and the work should be (re-)scheduled with a non-zero delay to allow the thread holding the lock to make progress.

Note that submitting from the work handler can fail if the work item had been cancelled. Generally this is acceptable, since the cancellation will complete once the handler finishes. If it is not, the code above must take other steps to notify the application that the work could not be performed.

Work items in isolation are self-locking, so you don't need to hold an external lock just to submit or schedule them. Even if you use external state protected by such a lock to prevent further resubmission, it's safe to do the resubmit as long as you're sure that eventually the item will take its lock and check that state to determine whether it should do anything. Where a delayable work item is being rescheduled in its handler due to inability to take the lock, some other self-locking state, such as an atomic flag set by the application/driver when the cancel is initiated, would be required to detect the cancellation and avoid the cancelled work item being submitted again after the deadline.

Check Return Values All work API functions return status of the underlying operation, and in many cases it is important to verify that the intended result was obtained.

- Submitting a work item (`k_work_submit_to_queue()`) can fail if the work is being cancelled or the queue is not accepting new items. If this happens the work will not be executed, which could cause a subsystem that is animated by work handler activity to become non-responsive.
- Asynchronous cancellation (`k_work_cancel()` or `k_work_cancel_delayable()`) can complete while the work item is still being run by a handler. Proceeding to manipulate state shared with the work handler will result in data races that can cause failures.

Many race conditions have been present in Zephyr code because the results of an operation were not checked.

There may be good reason to believe that a return value indicating that the operation did not complete as expected is not a problem. In those cases the code should clearly document this, by (1) casting the return value to `void` to indicate that the result is intentionally ignored, and (2) documenting what happens in the unexpected case. For example:

```
/* If this fails, the work handler will check pub->active and
 * exit without transmitting.
 */
(void)k_work_cancel_delayable(&pub->timer);
```

However in such a case the following code must still avoid data races, as it cannot guarantee that the work thread is not accessing work-related state.

Don't Optimize Prematurely The workqueue API is designed to be safe when invoked from multiple threads and interrupts. Attempts to externally inspect a work item's state and make decisions based on the result are likely to create new problems.

So when new work comes in, just submit it. Don't attempt to "optimize" by checking whether the work item is already submitted by inspecting snapshot state with `k_work_is_pending()` or `k_work_busy_get()`, or checking for a non-zero delay from `k_work_delayable_remaining_get()`. Those checks are fragile: a "busy" indication can be obsolete by the time the test is returned, and a "not-busy" indication can also be wrong if work is submitted from multiple contexts, or (for delayable work) if the deadline has completed but the work is still in queued or running state.

A general best practice is to always maintain in shared state some condition that can be checked by the handler to confirm whether there is work to be done. This way you can use the work handler as the standard cleanup path: rather than having to deal with cancellation and cleanup at points where items are submitted, you may be able to have everything done in the work handler itself.

A rare case where you could safely use `k_work_is_pending()` is as a check to avoid invoking `k_work_flush()` or `k_work_cancel_sync()`, if you are *certain* that nothing else might submit the work while you're checking (generally because you're holding a lock that prevents access to state used for submission).

Suggested Uses Use the system workqueue to defer complex interrupt-related processing from an ISR to a shared thread. This allows the interrupt-related processing to be done promptly without compromising the system's ability to respond to subsequent interrupts, and does not require the application to define and manage an additional thread to do the processing.

Configuration Options Related configuration options:

- `CONFIG_SYSTEM_WORKQUEUE_STACK_SIZE`
- `CONFIG_SYSTEM_WORKQUEUE_PRIORITY`
- `CONFIG_SYSTEM_WORKQUEUE_NO_YIELD`

API Reference
Work Queue APIs

Operation without Threads

Thread support is not necessary in some applications:

- Bootloaders
- Simple event-driven applications

- Examples intended to demonstrate core functionality

Thread support can be disabled by setting `CONFIG_MULTITHREADING` to `n`. Since this configuration has a significant impact on Zephyr's functionality and testing of it has been limited, there are conditions on what can be expected to work in this configuration.

What Can be Expected to Work These core capabilities shall function correctly when `CONFIG_MULTITHREADING` is disabled:

- The *build system*
- The ability to boot the application to `main()`
- *Interrupt management*
- The system clock including `k_uptime_get()`
- Timers, i.e. `k_timer()`
- Non-sleeping delays e.g. `k_busy_wait()`.
- Sleeping `k_cpu_idle()`.
- Pre `main()` drivers and subsystems initialization e.g. `SYS_INIT`.
- *Memory Management*
- Specifically identified drivers in certain subsystems, listed below.

The expectations above affect selection of other features; for example `CONFIG_SYS_CLOCK_EXISTS` cannot be set to `n`.

What Cannot be Expected to Work Functionality that will not work when `CONFIG_MULTITHREADING` is disabled includes majority of the kernel API:

- *Threads*
- *Scheduling*
- *Workqueue Threads*
- *Polling API*
- *Semaphores*
- *Mutexes*
- *Condition Variables*
- *Data Passing*

Subsystem Behavior Without Thread Support The sections below list driver and functional subsystems that are expected to work to some degree when `CONFIG_MULTITHREADING` is disabled. Subsystems that are not listed here should not be expected to work.

Some existing drivers within the listed subsystems do not work when threading is disabled, but are within scope based on their subsystem, or may be sufficiently isolated that supporting them on a particular platform is low-impact. Enhancements to add support to existing capabilities that were not originally implemented to work with threads disabled will be considered.

Flash The *Flash* is expected to work for all SoC flash peripheral drivers. Bus-accessed devices like serial memories may not be supported.

List/table of supported drivers to go here

GPIO The *General-Purpose Input/Output (GPIO)* is expected to work for all SoC GPIO peripheral drivers. Bus-accessed devices like GPIO extenders may not be supported.

List/table of supported drivers to go here

UART A subset of the *Universal Asynchronous Receiver-Transmitter (UART)* is expected to work for all SoC UART peripheral drivers.

- Applications that select `CONFIG_UART_INTERRUPT_DRIVEN` may work, depending on driver implementation.
- Applications that select `CONFIG_UART_ASYNC_API` may work, depending on driver implementation.
- Applications that do not select either `CONFIG_UART_ASYNC_API` or `CONFIG_UART_INTERRUPT_DRIVEN` are expected to work.

List/table of supported drivers to go here, including which API options are supported

Interrupts

An *interrupt service routine* (ISR) is a function that executes asynchronously in response to a hardware or software interrupt. An ISR normally preempts the execution of the current thread, allowing the response to occur with very low latency. Thread execution resumes only once all ISR work has been completed.

- *Concepts*
  - *Multi-level Interrupt Handling*
  - *Preventing Interruptions*
  - *Offloading ISR Work*
  - *Sharing interrupt lines*
- *Implementation*
  - *Defining a regular ISR*
  - *Defining a 'direct' ISR*
  - *RAM-based ISR Execution*
  - *Sharing an interrupt line*
  - *Dynamically disconnecting an ISR*
  - *Implementation Details*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of ISRs can be defined (limited only by available RAM), subject to the constraints imposed by underlying hardware.

An ISR has the following key properties:

- An **interrupt request (IRQ) signal** that triggers the ISR.
- A **priority level** associated with the IRQ.
- An **interrupt service routine** that is invoked to handle the interrupt.

- An **argument value** that is passed to that function.

An IDT (Interrupt Descriptor Table) or a vector table is used to associate a given interrupt source with a given ISR. Only a single ISR can be associated with a specific IRQ at any given time.

Multiple ISRs can utilize the same function to process interrupts, allowing a single function to service a device that generates multiple types of interrupts or to service multiple devices (usually of the same type). The argument value passed to an ISR's function allows the function to determine which interrupt has been signaled.

The kernel provides a default ISR for all unused IDT entries. This ISR generates a fatal system error if an unexpected interrupt is signaled.

The kernel supports **interrupt nesting**. This allows an ISR to be preempted in mid-execution if a higher priority interrupt is signaled. The lower priority ISR resumes execution once the higher priority ISR has completed its processing.

An ISR executes in the kernel's **interrupt context**. This context has its own dedicated stack area (or, on some architectures, stack areas). The size of the interrupt context stack must be capable of handling the execution of multiple concurrent ISRs if interrupt nesting support is enabled.

> **Important**
>
> Many kernel APIs can be used only by threads, and not by ISRs. In cases where a routine may be invoked by both threads and ISRs the kernel provides the `k_is_in_isr()` API to allow the routine to alter its behavior depending on whether it is executing as part of a thread or as part of an ISR.

Multi-level Interrupt Handling A hardware platform can support more interrupt lines than natively-provided through the use of one or more nested interrupt controllers. Sources of hardware interrupts are combined into one line that is then routed to the parent controller.

If nested interrupt controllers are supported, `CONFIG_MULTI_LEVEL_INTERRUPTS` should be enabled, and `CONFIG_2ND_LEVEL_INTERRUPTS` and `CONFIG_3RD_LEVEL_INTERRUPTS` configured as well, based on the hardware architecture.

A unique 32-bit interrupt number is assigned with information embedded in it to select and invoke the correct Interrupt Service Routine (ISR). Each interrupt level is given a byte within this 32-bit number, providing support for up to four interrupt levels using this arch, as illustrated and explained below:

```
          9                           2       0
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬
│   │   │   │   │   │   │   │   │ A │   │   │   │              (LEVEL 1)
└───┴───┴───┴─┬─┴───┴───┴───┴───┴───┴───┴─┬─┴───┴───
              └─────────────────┐         └─────────────────┐
      5                         v                           v
┌───┬───┬───┬───┬───┬───┬───┬───┐   ┌───┬───┬───┬───┬───┬───┬───┬───┐
│   │   │   │   │ C │   │   │   │   │   │   │   │   │   │ B │   │   │   (LEVEL 2)
└───┴───┴───┴─┬─┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴
              └─────────────────┐
                                v
┌───┬───┬───┬───┬───┬───┬───┬───┬
│   │   │   │   │   │ D │   │   │                              (LEVEL 3)
└───┴───┴───┴───┴───┴───┴───┴───┴
```

There are three interrupt levels shown here.

- A cell represents an interrupt line and is numbered from 0 (right most).

- LEVEL 1 has 12 interrupt lines, with two lines (2 and 9) connected to nested controllers and one device 'A' on line 4.
- One of the LEVEL 2 controllers has interrupt line 5 connected to a LEVEL 3 nested controller and one device 'C' on line 3.
- The other LEVEL 2 controller has no nested controllers but has one device 'B' on line 2.
- The LEVEL 3 controller has one device 'D' on line 2.

Here's how unique interrupt numbers are generated for each hardware interrupt. Let's consider four interrupts shown above as A, B, C, and D:

```
A -> 0x00000004
B -> 0x00000302
C -> 0x00000409
D -> 0x00030609
```

Note

The bit positions for LEVEL 2 and onward are offset by 1, as 0 means that interrupt number is not present for that level. For our example, the LEVEL 3 controller has device D on line 2, connected to the LEVEL 2 controller's line 5, that is connected to the LEVEL 1 controller's line 9 (2 -> 5 -> 9). Because of the encoding offset for LEVEL 2 and onward, device D is given the number 0x00030609.

Preventing Interruptions In certain situations it may be necessary for the current thread to prevent ISRs from executing while it is performing time-sensitive or critical section operations.

A thread may temporarily prevent all IRQ handling in the system using an **IRQ lock**. This lock can be applied even when it is already in effect, so routines can use it without having to know if it is already in effect. The thread must unlock its IRQ lock the same number of times it was locked before interrupts can be once again processed by the kernel while the thread is running.

Important

A thread is not allowed to *sleep* while holding an IRQ lock; only *isr-ok* functions may be called.

Alternatively, a thread may temporarily **disable** a specified IRQ so its associated ISR does not execute when the IRQ is signaled. The IRQ must be subsequently **enabled** to permit the ISR to execute.

Important

Disabling an IRQ prevents *all* threads in the system from being preempted by the associated ISR, not just the thread that disabled the IRQ.

Zero Latency Interrupts Preventing interruptions by applying an IRQ lock may increase the observed interrupt latency. A high interrupt latency, however, may not be acceptable for certain low-latency use-cases.

The kernel addresses such use-cases by allowing interrupts with critical latency constraints to execute at a priority level that cannot be blocked by interrupt locking. These interrupts are defined as *zero-latency interrupts*. The support for zero-latency interrupts requires `CONFIG_ZERO_LATENCY_IRQS` to be enabled. Any interrupts configured as zero-latency must also be declared as *direct ISRs* (and must not use the `ISR_DIRECT_PM` in them), since regular ISRs interact with the kernel. In addition to that, the flag `IRQ_ZERO_LATENCY` must be passed to the

IRQ_DIRECT_CONNECT macro to configure the particular interrupt with zero latency. Declaring a zero-latency interrupt ISR to be both direct and dynamic is possible on some architectures, see *Defining a 'direct' ISR*.

Zero-latency interrupts are expected to be used to manage hardware events directly, and not to interoperate with the kernel code at all. They should treat all kernel APIs as undefined behavior (i.e. an application that uses the APIs inside a zero-latency interrupt context is responsible for directly verifying correct behavior). Zero-latency interrupts may not modify any data inspected by kernel APIs invoked from normal Zephyr contexts and shall not generate exceptions that need to be handled synchronously (e.g. kernel panic).

When system power management keeps interrupts locked across PM resume, a zero-latency interrupt is outside the locked-resume ordering and may be dispatched during PM suspend/resume logic, before PM resume bookkeeping and SoC/device hardware restore have completed. Such an ISR must be PM-wake-safe, or the interrupt source must be masked or disabled while the system state does not allow the ISR to execute. See *device power policy constraints* for one way to keep unsafe states out of policy selection while a device path that can produce a zero-latency interrupt is active.

Important

Zero-latency interrupts are supported on an architecture-specific basis. The feature is currently implemented in the ARM Cortex-M architecture variant.

Tip

To mitigate flash access latency, consider relocating ISRs and all related symbols to RAM.

Offloading ISR Work An ISR should execute quickly to ensure predictable system operation. If time consuming processing is required the ISR should offload some or all processing to a thread, thereby restoring the kernel's ability to respond to other interrupts.

The kernel supports several mechanisms for offloading interrupt-related processing to a thread.

- An ISR can signal a helper thread to do interrupt-related processing using a kernel object, such as a FIFO, LIFO, or semaphore.
- An ISR can instruct the system workqueue thread to execute a work item. (See *Workqueue Threads*.)

When an ISR offloads work to a thread, there is typically a single context switch to that thread when the ISR completes, allowing interrupt-related processing to continue almost immediately. However, depending on the priority of the thread handling the offload, it is possible that the currently executing cooperative thread or other higher-priority threads may execute before the thread handling the offload is scheduled.

Sharing interrupt lines In the case of some hardware platforms, the same interrupt lines may be used by different IPs. For example, interrupt 17 may be used by a DMA controller to signal that a data transfer has been completed or by a DAI controller to signal that the transfer FIFO has reached its watermark. To make this work, one would have to either employ some special logic or find a workaround (for example, using the shared_irq interrupt controller), which doesn't scale very well.

To solve this problem, one may use shared interrupts, which can be enabled using CONFIG_SHARED_INTERRUPTS. Whenever an attempt to register a second ISR/argument pair on the same interrupt line is made (using IRQ_CONNECT or irq_connect_dynamic()), the interrupt line will become shared, meaning the two ISR/argument pairs (previous one and the one that has just been registered) will be invoked each time the interrupt is triggered. The entities that make use

of an interrupt line in the shared interrupt context are known as clients. The maximum number of allowed clients for an interrupt is controlled by CONFIG_SHARED_IRQ_MAX_NUM_CLIENTS.

Interrupt sharing is transparent to the user. As such, the user may register interrupts using IRQ_CONNECT and irq_connect_dynamic() as they normally would. The interrupt sharing is taken care of behind the scenes.

Enabling the shared interrupt support and dynamic interrupt support will allow users to dynamically disconnect ISRs using irq_disconnect_dynamic(). After an ISR is disconnected, whenever the interrupt line for which it was register gets triggered, the ISR will no longer get invoked.

Please note that enabling CONFIG_SHARED_INTERRUPTS will result in a non-negligible increase in the binary size. Use with caution.

Implementation

Defining a regular ISR An ISR is defined at runtime by calling IRQ_CONNECT. It must then be enabled by calling irq_enable().

> **Note**
>
> The unprefixed interrupt control APIs such as irq_enable() and irq_lock() are the legacy spelling. New code should use their namespaced equivalents, k_irq_enable(), k_irq_lock() and so on. The unprefixed names remain fully supported.

> **Important**
>
> IRQ_CONNECT() is not a C function and does some inline assembly magic behind the scenes. All its arguments must be known at build time. Drivers that have multiple instances may need to define per-instance config functions to configure each instance of the interrupt.

The following code defines and enables an ISR.

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_ISR_ARG  DEVICE_GET(my_device)
#define MY_IRQ_FLAGS 0       /* IRQ flags */

void my_isr(void *arg)
{
   ... /* ISR code */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Since the IRQ_CONNECT macro requires that all its parameters be known at build time, in some cases this may not be acceptable. It is also possible to install interrupts at runtime with irq_connect_dynamic(). It is used in exactly the same way as IRQ_CONNECT:

```
void my_isr_installer(void)
{
   ...
   irq_connect_dynamic(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG,
                       MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

Dynamic interrupts require the `CONFIG_DYNAMIC_INTERRUPTS` option to be enabled. Removing or re-configuring a dynamic interrupt is currently unsupported.

Defining a 'direct' ISR Regular Zephyr interrupts introduce some overhead which may be unacceptable for some low-latency use-cases. Specifically:

- The argument to the ISR is retrieved and passed to the ISR
- If power management is enabled and the system was idle, all the hardware will be resumed from low-power state before the ISR is executed, which can be very time-consuming
- Although some architectures will do this in hardware, other architectures need to switch to the interrupt stack in code
- After the interrupt is serviced, the OS then performs some logic to potentially make a scheduling decision
- *Zero Latency Interrupts* must always be declared as direct ISRs, since regular ISRs interact with the kernel

Zephyr supports so-called 'direct' interrupts, which are installed via `IRQ_DIRECT_CONNECT` and whose handlers are declared using `ISR_DIRECT_DECLARE`. These direct interrupts have some special implementation requirements and a reduced feature set; see the definitions of `IRQ_DIRECT_CONNECT` and `ISR_DIRECT_DECLARE` for details.

Direct interrupts are only available on architectures selecting `CONFIG_ARCH_HAS_DIRECT_INTERRUPTS`. Using `IRQ_DIRECT_CONNECT` or `ISR_DIRECT_DECLARE` on any other architecture fails the build.

The following code demonstrates a direct ISR:

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
#define MY_IRQ_FLAGS 0       /* IRQ flags */

ISR_DIRECT_DECLARE(my_isr)
{
   do_stuff();
   /* PM done after servicing interrupt for best latency. This cannot be
   used for zero-latency IRQs because it accesses kernel data. */
   ISR_DIRECT_PM();
   /* Ask the kernel to check if scheduling decision should be made. If the
   ISR is for a zero-latency IRQ then the return value must always be 0. */
   return 1;
}

void my_isr_installer(void)
{
   ...
   IRQ_DIRECT_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Installation of dynamic direct interrupts is supported on an architecture-specific basis. The feature is currently implemented in the Arm Cortex-M architecture variant via the macro `ARM_IRQ_DIRECT_DYNAMIC_CONNECT`, which can be used to declare a direct and dynamic interrupt.

RAM-based ISR Execution For ultra-low latency, ISRs and vector tables can be relocated to RAM to eliminate flash access delays. That can be achieved by enabling the `CONFIG_SRAM_VECTOR_TABLE` and `CONFIG_SRAM_SW_ISR_TABLE` options which will result in vector tables being placed in RAM. Then, Zephyr *Code and Data Relocation* can be used to relocate the ISR code and all the related symbols to RAM as well.

Sharing an interrupt line The following code defines two ISRs using the same interrupt number.

```
#define MY_DEV_IRQ 24               /* device uses INTID 24 */
#define MY_DEV_IRQ_PRIO 2           /* device uses interrupt priority 2 */
/*  this argument may be anything */
#define MY_FST_ISR_ARG INT_TO_POINTER(1)
/*  this argument may be anything */
#define MY_SND_ISR_ARG INT_TO_POINTER(2)
#define MY_IRQ_FLAGS 0              /* IRQ flags */

void my_first_isr(void *arg)
{
    ... /* some magic happens here */
}

void my_second_isr(void *arg)
{
    ... /* even more magic happens here */
}

void my_isr_installer(void)
{
    ...
    IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_first_isr, MY_FST_ISR_ARG,␣
↪MY_IRQ_FLAGS);
    IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_second_isr, MY_SND_ISR_ARG,␣
↪MY_IRQ_FLAGS);
    ...
}
```

The same restrictions regarding `IRQ_CONNECT` described in *Defining a regular ISR* are applicable here. If `CONFIG_SHARED_INTERRUPTS` is disabled, the above code will generate a build error. Otherwise, the above code will result in the two ISRs being invoked each time interrupt 24 is triggered.

If `CONFIG_SHARED_IRQ_MAX_NUM_CLIENTS` is set to a value lower than 2 (current number of clients), a build error will be generated.

If dynamic interrupts are enabled, `irq_connect_dynamic()` will allow sharing interrupts during runtime. Exceeding the configured maximum number of allowed clients will result in a failed assertion.

Dynamically disconnecting an ISR The following code defines two ISRs using the same interrupt number. The second ISR is disconnected during runtime.

```
#define MY_DEV_IRQ 24               /* device uses INTID 24 */
#define MY_DEV_IRQ_PRIO 2           /* device uses interrupt priority 2 */
/*  this argument may be anything */
#define MY_FST_ISR_ARG INT_TO_POINTER(1)
/*  this argument may be anything */
#define MY_SND_ISR_ARG INT_TO_POINTER(2)
#define MY_IRQ_FLAGS 0              /* IRQ flags */

void my_first_isr(void *arg)
{
   ... /* some magic happens here */
}

void my_second_isr(void *arg)
{
   ... /* even more magic happens here */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_first_isr, MY_FST_ISR_ARG,␣
↪MY_IRQ_FLAGS);
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_second_isr, MY_SND_ISR_ARG,␣
↪MY_IRQ_FLAGS);
   ...
}

void my_isr_uninstaller(void)
{
   ...
   irq_disconnect_dynamic(MY_DEV_IRQ, MY_DEV_IRQ_PRIO, my_first_isr,␣
↪MY_FST_ISR_ARG, MY_IRQ_FLAGS);
   ...
}
```

The `irq_disconnect_dynamic()` call will result in interrupt 24 becoming unshared, meaning the system will act as if the first `IRQ_CONNECT` call never happened. This behaviour is only allowed if `CONFIG_DYNAMIC_INTERRUPTS` is enabled, otherwise a linker error will be generated.

Implementation Details Interrupt tables are set up at build time using some special build tools. The details laid out here apply to all architectures except x86, which are covered in the *x86 Details* section below.

The invocation of `IRQ_CONNECT` will declare an instance of struct _isr_list which is placed in a special .intList section. This section is placed in compiled code on precompilation stages only. It is meant to be used by Zephyr script to generate interrupt tables and is removed from the final build. The script implements different parsers to process the data from .intList section and produce the required output.

The default parser generates C arrays filled with arguments and interrupt handlers in a form of addresses directly taken from .intList section entries. It works with all the architectures and compilers (with the exception mentioned above). The limitation of this parser is the fact that after the arrays are generated it is expected for the code not to relocate. Any relocation on this stage may lead to the situation where the entry in the interrupt array is no longer pointing to the function that was expected. It means that this parser, being more compatible, is limiting us

from using Link Time Optimization.

The local ISR declaration parser uses a different approach to construct the same arrays at the binary level. All the entries to the arrays are declared and defined locally, directly in the file where `IRQ_CONNECT` is used. They are placed in a section with a unique, synthesized name. The name of that section is then placed in the .intList section, which is used to generate a linker script fragment placing the entry at the right address. This parser is currently limited to the supported architectures and toolchains, but in return, it preserves the information about object relations for the linker, thus enabling Link Time Optimization.

Implementation using C arrays This is the default configuration available for all Zephyr supported architectures.

Any invocation of `IRQ_CONNECT` will declare an instance of struct _isr_list which is placed in a special .intList section:

```
struct _isr_list {
    /** IRQ line number */
    int32_t irq;
    /** Flags for this IRQ, see ISR_FLAG_* definitions */
    int32_t flags;
    /** ISR to call */
    void *func;
    /** Parameter for non-direct IRQs */
    void *param;
};
```

Zephyr is built in two phases; the first phase of the build produces `${ZEPHYR_PREBUILT_EXECUTABLE}`.elf which contains all the entries in the .intList section preceded by a header:

```
struct {
    void *spurious_irq_handler;
    void *sw_irq_handler;
    uint32_t num_isrs;
    uint32_t num_vectors;
    struct _isr_list isrs[];  <- of size num_isrs
};
```

This data consisting of the header and instances of struct _isr_list inside `${ZEPHYR_PREBUILT_EXECUTABLE}`.elf is then used by the gen_isr_tables.py script to generate a C file defining a vector table and software ISR table that are then compiled and linked into the final application.

The priority level of any interrupt is not encoded in these tables, instead `IRQ_CONNECT` also has a runtime component which programs the desired priority level of the interrupt to the interrupt controller. Some architectures do not support the notion of interrupt priority, in which case the priority argument is ignored.

Vector Table A vector table is generated when `CONFIG_GEN_IRQ_VECTOR_TABLE` is enabled. This data structure is used natively by the CPU and is simply an array of function pointers, where each element n corresponds to the IRQ handler for IRQ line n, and the function pointers are:

1. For 'direct' interrupts declared with `IRQ_DIRECT_CONNECT`, the handler function will be placed here.
2. For regular interrupts declared with `IRQ_CONNECT`, the address of the common software IRQ handler is placed here. This code does common kernel interrupt bookkeeping and looks up the ISR and parameter from the software ISR table.

3. For interrupt lines that are not configured at all, the address of the spurious IRQ handler will be placed here. The spurious IRQ handler causes a system fatal error if encountered.

Some architectures have a common entry point for all interrupts and do not support a vector table, in which case the `CONFIG_GEN_IRQ_VECTOR_TABLE` option should be disabled.

Some architectures may reserve some initial vectors for system exceptions and declare this in a table elsewhere, in which case CONFIG_GEN_IRQ_START_VECTOR needs to be set to properly offset the indices in the table.

SW ISR Table This is an array of struct _isr_table_entry:

```
struct _isr_table_entry {
    void *arg;
    void (*isr)(void *);
};
```

This is used by the common software IRQ handler to look up the ISR and its argument and execute it. The active IRQ line is looked up in an interrupt controller register and used to index this table.

Shared SW ISR Table This is an array of struct z_shared_isr_table_entry:

```
struct z_shared_isr_table_entry {
    struct _isr_table_entry clients[CONFIG_SHARED_IRQ_MAX_NUM_CLIENTS];
    size_t client_num;
};
```

This table keeps track of the registered clients for each of the interrupt lines. Whenever an interrupt line becomes shared, `z_shared_isr()` will replace the currently registered ISR in _sw_isr_table. This special ISR will iterate through the list of registered clients and invoke the ISRs.

Implementation using linker script This way of prepare and parse .isrList section to implement interrupt vectors arrays is called local ISR declaration. The name comes from the fact that all the entries to the arrays that would create interrupt vectors are created locally in place of invocation of `IRQ_CONNECT` macro. Then automatically generated linker scripts are used to place it in the right place in the memory.

This option requires enabling by the choose of `CONFIG_ISR_TABLES_LOCAL_DECLARATION`. If this configuration is supported by the used architecture and toolchaing the `CONFIG_ISR_TABLES_LOCAL_DECLARATION_SUPPORTED` is set. See details of this option for the information about currently supported configurations.

Any invocation of `IRQ_CONNECT` or `IRQ_DIRECT_CONNECT` will declare an instance of `struct _isr_list_sname` which is placed in a special .intList section:

```
struct _isr_list_sname {
    /** IRQ line number */
    int32_t irq;
    /** Flags for this IRQ, see ISR_FLAG_* definitions */
    int32_t flags;
    /** The section name */
    const char sname[];
};
```

Note that the section name is placed in flexible array member. It means that the size of the initialized structure will vary depending on the structure name length. The whole entry is used by the script during the build of the application and has all the information needed for proper interrupt placement.

Beside of the _isr_list_sname the IRQ_CONNECT macro generates an entry that would be the part of the interrupt array:

```
struct _isr_table_entry {
    const void *arg;
    void (*isr)(const void *);
};
```

This array is placed in a section with the name saved in _isr_list_sname structure.

The values created by IRQ_DIRECT_CONNECT macro depends on the architecture. It can be changed to variable that points to a interrupt handler:

```
static uintptr_t <unique name> = ((uintptr_t)func);
```

Or to actually naked function that implements a jump to the interrupt handler:

```
static void <unique name>(void)
{
    __asm(ARCH_IRQ_VECTOR_JUMP_CODE(func));
}
```

Similar like for IRQ_CONNECT, the created variable or function is placed in a section, saved in _isr_list_sname section.

Files generated by the script The interrupt tables generator script creates 3 files: `isr_tables.c`, `isr_tables_swi.ld`, and `isr_tables_vt.ld`.

The `isr_tables.c` will contain all the structures for interrupts, direct interrupts and shared interrupts (if enabled). This file implements only all the structures that are not implemented by the application, leaving a comment where the interrupt not implemented here can be found.

Then two linker files are used. The `isr_tables_vt.ld` file is included in place where the interrupt vectors are required to be placed in the selected architecture. The `isr_tables_swi.ld` file describes the placement of the software interrupt table elements. The separated file is required as it might be placed in writable or nonwritable section, depending on the current configuration.

x86 Details The x86 architecture has a special type of vector table called the Interrupt Descriptor Table (IDT) which must be laid out in a certain way per the x86 processor documentation. It is still fundamentally a vector table, and the *arch/x86/gen_idt.py* tool uses the .intList section to create it. However, on APIC-based systems the indexes in the vector table do not correspond to the IRQ line. The first 32 vectors are reserved for CPU exceptions, and all remaining vectors (up to index 255) correspond to the priority level, in groups of 16. In this scheme, interrupts of priority level 0 will be placed in vectors 32-47, level 1 48-63, and so forth. When the *arch/x86/gen_idt.py* tool is constructing the IDT, when it configures an interrupt it will look for a free vector in the appropriate range for the requested priority level and set the handler there.

On x86 when an interrupt or exception vector is executed by the CPU, there is no foolproof way to determine which vector was fired, so a software ISR table indexed by IRQ line is not used. Instead, the IRQ_CONNECT call creates a small assembly language function which calls the common interrupt code in `_interrupt_enter()` with the ISR and parameter as arguments. It is the address of this assembly interrupt stub which gets placed in the IDT. For interrupts declared with IRQ_DIRECT_CONNECT the parameterless ISR is placed directly in the IDT.

On systems where the position in the vector table corresponds to the interrupt's priority level, the interrupt controller needs to know at runtime what vector is associated with an IRQ line. *arch/x86/gen_idt.py* additionally creates an _irq_to_interrupt_vector array which maps an IRQ line to its configured vector in the IDT. This is used at runtime by IRQ_CONNECT to program the IRQ-to-vector association in the interrupt controller.

For dynamic interrupts, the build must generate some 4-byte dynamic interrupt stubs, one stub per dynamic interrupt in use. The number of stubs is controlled by the `CONFIG_X86_DYNAMIC_IRQ_STUBS` option. Each stub pushes an unique identifier which is then used to fetch the appropriate handler function and parameter out of a table populated when the dynamic interrupt was connected.

Going Beyond the Default Supported Number of Interrupts When generating interrupts in the multi-level configuration, 8-bits per level is the default mask used when determining which level a given interrupt code belongs to. This can become a problem when dealing with CPUs that support more than 255 interrupts per single aggregator. In this case it may be desirable to override these defaults and use a custom number of bits per level. Regardless of how many bits used for each level, the sum of the total bits used between all levels must sum to be less than or equal to 32-bits, fitting into a single 32-bit integer. To modify the bit total per level, override the default 8 in `Kconfig.multilevel` by setting `CONFIG_1ST_LEVEL_INTERRUPT_BITS` for the first level, `CONFIG_2ND_LEVEL_INTERRUPT_BITS` for the second level and `CONFIG_3RD_LEVEL_INTERRUPT_BITS` for the third level. These masks control the length of the bit masks and shift to apply when generating interrupt values, when checking the interrupts level and converting interrupts to a different level. The logic controlling this can be found in `irq_multilevel.h`

Suggested Uses Use a regular or direct ISR to perform interrupt processing that requires a very rapid response, and can be done quickly without blocking.

> **Note**
>
> Interrupt processing that is time consuming, or involves blocking, should be handed off to a thread. See *Offloading ISR Work* for a description of various techniques that can be used in an application.

Configuration Options Related configuration options:

- `CONFIG_ISR_STACK_SIZE`

Additional architecture-specific and device-specific configuration options also exist.

API Reference

Interrupt Service Routine APIs

Polling API

The polling API is used to wait concurrently for any one of multiple conditions to be fulfilled.

- *Concepts*
- *Implementation*
  - *Using k_poll()*
  - *Using k_poll_signal_raise()*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts The polling API's main function is `k_poll()`, which is very similar in concept to the POSIX `poll()` function, except that it operates on kernel objects rather than on file descriptors.

The polling API allows a single thread to wait concurrently for one or more conditions to be fulfilled without actively looking at each one individually.

There is a limited set of such conditions:

- a semaphore becomes available
- a kernel FIFO contains data ready to be retrieved
- a kernel LIFO contains data ready to be retrieved
- a kernel message queue contains data ready to be retrieved
- a kernel pipe contains data ready to be retrieved
- a poll signal is raised

A thread that wants to wait on multiple conditions must define an array of **poll events**, one for each condition.

All events in the array must be initialized before the array can be polled on.

Each event must specify which **type** of condition must be satisfied so that its state is changed to signal the requested condition has been met.

Each event must specify what **kernel object** it wants the condition to be satisfied for.

Each event must specify which **mode** of operation is used when the condition is satisfied.

Each event can optionally specify a **tag** to group multiple events together, at the user's discretion.

Apart from the kernel objects, there is also a **poll signal** pseudo-object type that be directly signaled.

The `k_poll()` function returns as soon as one of the conditions it is waiting for is fulfilled. It is possible for more than one to be fulfilled when `k_poll()` returns, if they were fulfilled before `k_poll()` was called, or due to the preemptive multi-threading nature of the kernel. The caller must look at the state of all the poll events in the array to figure out which ones were fulfilled and what actions to take.

Currently, there is only one mode of operation available: the object is not acquired. As an example, this means that when `k_poll()` returns and the poll event states that the semaphore is available, the caller of `k_poll()` must then invoke `k_sem_take()` to take ownership of the semaphore. If the semaphore is contested, there is no guarantee that it will be still available when `k_sem_take()` is called.

Implementation

Using k_poll() The main API is `k_poll()`, which operates on an array of poll events of type `k_poll_event`. Each entry in the array represents one event with a condition, which a call to `k_poll()` will wait for to be fulfilled.

Poll events can be initialized using either the runtime initializers `K_POLL_EVENT_INITIALIZER()` or `k_poll_event_init()`, or the static initializer `K_POLL_EVENT_STATIC_INITIALIZER()`. An object that matches the **type** specified must be passed to the initializers. The **mode** *must* be set to `K_POLL_MODE_NOTIFY_ONLY`. The state *must* be set to `K_POLL_STATE_NOT_READY` (the initializers take care of this). The user **tag** is optional and completely opaque to the API: it is there to help a user to group similar events together. Being optional, it is passed to the static initializer, but not the runtime ones for performance reasons. If using runtime initializers, the user must set it separately in the `k_poll_event` data structure. If an event in the array is to be ignored, most likely temporarily, its type can be set to `K_POLL_TYPE_IGNORE`.

```
struct k_poll_event events[4] = {
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_SEM_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_sem, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_fifo, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_MSGQ_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_msgq, 0),
    K_POLL_EVENT_STATIC_INITIALIZER(K_POLL_TYPE_PIPE_DATA_AVAILABLE,
                                    K_POLL_MODE_NOTIFY_ONLY,
                                    &my_pipe, 0),
};
```

or at runtime

```
struct k_poll_event events[4];
void some_init(void)
{
    k_poll_event_init(&events[0],
                      K_POLL_TYPE_SEM_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_sem);

    k_poll_event_init(&events[1],
                      K_POLL_TYPE_FIFO_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_fifo);

    k_poll_event_init(&events[2],
                      K_POLL_TYPE_MSGQ_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_msgq);

    k_poll_event_init(&events[3],
                      K_POLL_TYPE_PIPE_DATA_AVAILABLE,
                      K_POLL_MODE_NOTIFY_ONLY,
                      &my_pipe);

    // tags are left uninitialized if unused
}
```

After the events are initialized, the array can be passed to `k_poll()`. A timeout can be specified to wait only for a specified amount of time, or the special values `K_NO_WAIT` and `K_FOREVER` to either not wait or wait until an event condition is satisfied and not sooner.

A list of pollers is offered on each semaphore or FIFO and as many events can wait in it as the app wants. Notice that the waiters will be served in first-come-first-serve order, not in priority order.

In case of success, `k_poll()` returns 0. If it times out, it returns -`EAGAIN`.

```
// assume there is no contention on this semaphore and FIFO
// -EADDRINUSE will not occur; the semaphore and/or data will be available

void do_stuff(void)
{
```

(continues on next page)

(continued from previous page)

```
    rc = k_poll(events, ARRAY_SIZE(events), K_MSEC(1000));
    if (rc == 0) {
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        } else if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        } else if (events[2].state == K_POLL_STATE_MSGQ_DATA_AVAILABLE) {
            ret = k_msgq_get(events[2].msgq, buf, K_NO_WAIT);
            // handle data
        } else if (events[3].state == K_POLL_STATE_PIPE_DATA_AVAILABLE) {
            bytes_read = k_pipe_read(events[3].pipe, buf, bytes_to_read,␣
↪K_NO_WAIT);
            // handle data
        }
    } else {
        // handle timeout
    }
}
```

When `k_poll()` is called in a loop, the events state must be reset to `K_POLL_STATE_NOT_READY` by the user.

```
void do_stuff(void)
{
    for(;;) {
        rc = k_poll(events, ARRAY_SIZE(events), K_FOREVER);
        if (events[0].state == K_POLL_STATE_SEM_AVAILABLE) {
            k_sem_take(events[0].sem, 0);
        }
        if (events[1].state == K_POLL_STATE_FIFO_DATA_AVAILABLE) {
            data = k_fifo_get(events[1].fifo, 0);
            // handle data
        }
        if (events[2].state == K_POLL_STATE_MSGQ_DATA_AVAILABLE) {
            ret = k_msgq_get(events[2].msgq, buf, K_NO_WAIT);
            // handle data
        }
        if (events[3].state == K_POLL_STATE_PIPE_DATA_AVAILABLE) {
            bytes_read = k_pipe_read(events[3].pipe, buf, bytes_to_read,␣
↪K_NO_WAIT);
            // handle data
        }
        events[0].state = K_POLL_STATE_NOT_READY;
        events[1].state = K_POLL_STATE_NOT_READY;
        events[2].state = K_POLL_STATE_NOT_READY;
        events[3].state = K_POLL_STATE_NOT_READY;
    }
}
```

Using k_poll_signal_raise() One of the types of events is `K_POLL_TYPE_SIGNAL`: this is a "direct" signal to a poll event. This can be seen as a lightweight binary semaphore only one thread can wait for.

A poll signal is a separate object of type `k_poll_signal` that must be attached to a k_poll_event, similar to a semaphore or FIFO. It must first be initialized either via `K_POLL_SIGNAL_INITIALIZER()` or `k_poll_signal_init()`.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);
}
```

It is signaled via the `k_poll_signal_raise()` function. This function takes a user **result** parameter that is opaque to the API and can be used to pass extra information to the thread waiting on the event.

```
struct k_poll_signal signal;

// thread A
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal),
    };

    k_poll(events, 1, K_FOREVER);

    int signaled, result;

    k_poll_signal_check(&signal, &signaled, &result);

    if (signaled && (result == 0x1337)) {
        // A-OK!
    } else {
        // weird error
    }
}

// thread B
void signal_do_stuff(void)
{
    k_poll_signal_raise(&signal, 0x1337);
}
```

If the signal is to be polled in a loop, *both* its event state must be reset to `K_POLL_STATE_NOT_READY` *and* its `result` must be reset using `k_poll_signal_reset()` on each iteration if it has been signaled.

```
struct k_poll_signal signal;
void do_stuff(void)
{
    k_poll_signal_init(&signal);

    struct k_poll_event events[1] = {
        K_POLL_EVENT_INITIALIZER(K_POLL_TYPE_SIGNAL,
                                 K_POLL_MODE_NOTIFY_ONLY,
                                 &signal),
    };
```

(continues on next page)

(continued from previous page)

```
    for (;;) {
        k_poll(events, 1, K_FOREVER);

        int signaled, result;

        k_poll_signal_check(&signal, &signaled, &result);

        if (signaled && (result == 0x1337)) {
            // A-OK!
        } else {
            // weird error
        }

        k_poll_signal_reset(&signal);
        events[0].state = K_POLL_STATE_NOT_READY;
    }
}
```

Note that poll signals are not internally synchronized. A `k_poll()` call that is passed a signal will return after any code in the system calls `k_poll_signal_raise()`. But if the signal is being externally managed and reset via `k_poll_signal_init()`, it is possible that by the time the application checks, the event state may no longer be equal to `K_POLL_STATE_SIGNALED`, and a (naive) application will miss events. Best practice is always to reset the signal only from within the thread invoking the `k_poll()` loop, or else to use some other event type which tracks event counts: semaphores and FIFOs are more error-proof in this sense because they can't "miss" events, architecturally.

Suggested Uses Use `k_poll()` to consolidate multiple threads that would be pending on one object each, saving possibly large amounts of stack space.

Use a poll signal as a lightweight binary semaphore if only one thread pends on it.

> **Note**
>
> Because objects are only signaled if no other thread is waiting for them to become available and only one thread can poll on a specific object, polling is best used when objects are not subject of contention between multiple threads, basically when a single thread operates as a main "server" or "dispatcher" for multiple objects and is the only one trying to acquire these objects.

Configuration Options Related configuration options:

- `CONFIG_POLL`

API Reference

Async polling APIs

Semaphores

A *semaphore* is a kernel object that implements a traditional counting semaphore.

- *Concepts*

- *Implementation*
  - *Defining a Semaphore*
  - *Giving a Semaphore*
  - *Taking a Semaphore*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*
- *User Mode Semaphore API Reference*

Concepts Any number of semaphores can be defined (limited only by available RAM). Each semaphore is referenced by its memory address.

A semaphore has the following key properties:

- A **count** that indicates the number of times the semaphore can be taken. A count of zero indicates that the semaphore is unavailable.
- A **limit** that indicates the maximum value the semaphore's count can reach.

A semaphore must be initialized before it can be used. Its count must be set to a non-negative value that is less than or equal to its limit.

A semaphore may be **given** by a thread or an ISR. Giving the semaphore increments its count, unless the count is already equal to the limit.

A semaphore may be **taken** by a thread. Taking the semaphore decrements its count, unless the semaphore is unavailable (i.e. at zero). When a semaphore is unavailable, a thread may choose to wait for it to be given. Any number of threads may wait on an unavailable semaphore simultaneously. When the semaphore is given, it is taken by the highest priority thread that has waited longest.

> **Note**
>
> You may initialize a "full" semaphore (count equal to limit) to limit the number of threads able to execute the critical section at the same time. You may also initialize an empty semaphore (count equal to 0, with a limit greater than 0) to create a gate through which no waiting thread may pass until the semaphore is incremented. All standard use cases of the common semaphore are supported.

> **Note**
>
> The kernel does allow an ISR to take a semaphore, however the ISR must not attempt to wait if the semaphore is unavailable.

Implementation

Defining a Semaphore A semaphore is defined using a variable of type `k_sem`. It must then be initialized by calling `k_sem_init()`.

The following code defines a semaphore, then configures it as a binary semaphore by setting its count to 0 and its limit to 1.

```
struct k_sem my_sem;

k_sem_init(&my_sem, 0, 1);
```

Alternatively, a semaphore can be defined and initialized at compile time by calling K_SEM_DEFINE.

The following code has the same effect as the code segment above.

```
K_SEM_DEFINE(my_sem, 0, 1);
```

Giving a Semaphore A semaphore is given by calling k_sem_give().

The following code builds on the example above, and gives the semaphore to indicate that a unit of data is available for processing by a consumer thread.

```
void input_data_interrupt_handler(void *arg)
{
    /* notify thread that data is available */
    k_sem_give(&my_sem);

    ...
}
```

Taking a Semaphore A semaphore is taken by calling k_sem_take().

The following code builds on the example above, and waits up to 50 milliseconds for the semaphore to be given. A warning is issued if the semaphore is not obtained in time.

```
void consumer_thread(void)
{
    ...

    if (k_sem_take(&my_sem, K_MSEC(50)) != 0) {
        printk("Input data not available!");
    } else {
        /* fetch available data */
        ...
    }
    ...
}
```

Suggested Uses Use a semaphore to control access to a set of resources by multiple threads.

Use a semaphore to synchronize processing between producing and consuming threads or ISRs.

Configuration Options Related configuration options:

- None.

API Reference
Semaphore APIs

Related code samples

- Basic SynchronizationManipulate basic kernel synchronization primitives.

- Dining PhilosophersImplement a solution to the Dining Philosophers problem using Zephyr kernel services.

User Mode Semaphore API Reference The `sys_sem` exists in user memory, working as counter semaphore for user mode threads, when user mode is enabled. When user mode isn't enabled, `sys_sem` behaves like `k_sem`.

User mode semaphore APIs

Mutexes

A *mutex* is a kernel object that implements a traditional reentrant mutex. A mutex allows multiple threads to safely share an associated hardware or software resource by ensuring mutually exclusive access to the resource.

- *Concepts*
  - *Reentrant Locking*
  - *Priority Inheritance*
- *Implementation*
  - *Defining a Mutex*
  - *Locking a Mutex*
  - *Unlocking a Mutex*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*
- *Futex API Reference*
- *User Mode Mutex API Reference*

Concepts Any number of mutexes can be defined (limited only by available RAM). Each mutex is referenced by its memory address.

A mutex has the following key properties:

- A **lock count** that indicates the number of times the mutex has been locked by the thread that has locked it. A count of zero indicates that the mutex is unlocked.
- An **owning thread** that identifies the thread that has locked the mutex, when it is locked.

A mutex must be initialized before it can be used. This sets its lock count to zero.

A thread that needs to use a shared resource must first gain exclusive rights to access it by **locking** the associated mutex. If the mutex is already locked by another thread, the requesting thread may choose to wait for the mutex to be unlocked.

After locking a mutex, the thread may safely use the associated resource for as long as needed; however, it is considered good practice to hold the lock for as short a time as possible to avoid negatively impacting other threads that want to use the resource. When the thread no longer needs the resource it must **unlock** the mutex to allow other threads to use the resource.

A mutex must not be freed, or have its memory reused, while it is locked or while threads are waiting on it. A mutex embedded in a dynamically allocated object must be unlocked before

that object is released; otherwise the owning thread is left tracking a mutex in memory that no longer belongs to it.

Any number of threads may wait on a locked mutex simultaneously. When the mutex becomes unlocked it is then locked by the highest-priority thread that has waited the longest.

> **Note**
>
> Mutex objects are *not* designed for use by ISRs.

Reentrant Locking A thread is permitted to lock a mutex it has already locked. This allows the thread to access the associated resource at a point in its execution when the mutex may or may not already be locked.

A mutex that is repeatedly locked by a thread must be unlocked an equal number of times before the mutex becomes fully unlocked so it can be claimed by another thread.

Priority Inheritance The thread that has locked a mutex is eligible for *priority inheritance*. This means the kernel will *temporarily* elevate the thread's priority if a higher priority thread begins waiting on the mutex. This allows the owning thread to complete its work and release the mutex more rapidly by executing at the same priority as the waiting thread. Once the mutex has been unlocked, the unlocking thread resets its priority to the level it had before locking that mutex.

> **Note**
>
> The `CONFIG_PRIORITY_CEILING` configuration option limits how high the kernel can raise a thread's priority due to priority inheritance. The default value of -128 permits unlimited. Setting this to a priority level that is at or lower than the priority of the idle thread disables it.

A thread's priority prior to any mutex-related boost is saved on the thread itself the first time it is raised, and each mutex it holds is tracked in a per-thread list. Each time a higher priority thread waits on a mutex, the kernel adjusts the owning thread's priority. When the owning thread releases a lock (or if the high priority waiting thread times out), the kernel rescans the thread's remaining held mutexes and recalculates its priority from the highest-priority waiter still pending on any of them, falling back to the thread's original priority if none remain. This gives correct results regardless of the order in which held mutexes are unlocked.

If the owner of a mutex is itself blocked waiting on another mutex, a priority boost is propagated along that ownership chain so that every thread blocking progress towards the waiter is raised in turn, up to an implementation-defined limit on the number of chain hops walked per lock attempt. Owners beyond that limit keep their pre-boost priority until a later lock attempt walks the chain again.

Implementation

Defining a Mutex A mutex is defined using a variable of type `k_mutex`. It must then be initialized by calling `k_mutex_init()`.

The following code defines and initializes a mutex.

```
struct k_mutex my_mutex;

k_mutex_init(&my_mutex);
```

Alternatively, a mutex can be defined and initialized at compile time by calling K_MUTEX_DEFINE.

The following code has the same effect as the code segment above.

```
K_MUTEX_DEFINE(my_mutex);
```

Locking a Mutex A mutex is locked by calling k_mutex_lock().

The following code builds on the example above, and waits indefinitely for the mutex to become available if it is already locked by another thread.

```
k_mutex_lock(&my_mutex, K_FOREVER);
```

The following code waits up to 100 milliseconds for the mutex to become available, and gives a warning if the mutex does not become available.

```
if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
    /* mutex successfully locked */
} else {
    printf("Cannot lock XYZ display\n");
}
```

Unlocking a Mutex A mutex is unlocked by calling k_mutex_unlock().

The following code builds on the example above, and unlocks the mutex that was previously locked by the thread.

```
k_mutex_unlock(&my_mutex);
```

Suggested Uses Use a mutex to provide exclusive access to a resource, such as a physical device.

Configuration Options Related configuration options:

- CONFIG_PRIORITY_CEILING

API Reference

Mutex APIs

Related code samples

- Dining PhilosophersImplement a solution to the Dining Philosophers problem using Zephyr kernel services.

Futex API Reference k_futex is a lightweight mutual exclusion primitive designed to minimize kernel involvement. Uncontended operation relies only on atomic access to shared memory. k_futex are not kernel objects and do not use the kernel object permission management. They live in user memory and can be used by any thread which has read-write access to this memory.

FUTEX APIs

User Mode Mutex API Reference sys_mutex behaves almost exactly like k_mutex, with the added advantage that a sys_mutex instance can reside in user memory. When user mode isn't enabled, sys_mutex behaves like k_mutex.

User mode mutex APIs

Condition Variables

A *condition variable* is a synchronization primitive that enables threads to wait until a particular condition occurs.

- *Concepts*
- *Implementation*
  - *Defining a Condition Variable*
  - *Waiting on a Condition Variable*
  - *Signaling a Condition Variable*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of condition variables can be defined (limited only by available RAM). Each condition variable is referenced by its memory address.

To wait for a condition to become true, a thread can make use of a condition variable.

A condition variable is basically a queue of threads that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition). The function `k_condvar_wait()` performs atomically the following steps;

1. Releases the last acquired mutex.
2. Puts the current thread in the condition variable queue.

Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue by signaling on the condition using `k_condvar_signal()` or `k_condvar_broadcast()` then it:

1. Re-acquires the mutex previously released.
2. Returns from `k_condvar_wait()`.

Regardless of why the wait completes – whether the thread was signaled, the supplied timeout elapsed, or the wait was requested without blocking – `k_condvar_wait()` always returns with the associated mutex re-locked by the calling thread.

A condition variable must be initialized before it can be used.

Implementation

Defining a Condition Variable A condition variable is defined using a variable of type `k_condvar`. It must then be initialized by calling `k_condvar_init()`.

The following code defines a condition variable:

```
struct k_condvar my_condvar;

k_condvar_init(&my_condvar);
```

Alternatively, a condition variable can be defined and initialized at compile time by calling `K_CONDVAR_DEFINE`.

The following code has the same effect as the code segment above.

```
K_CONDVAR_DEFINE(my_condvar);
```

Waiting on a Condition Variable A thread can wait on a condition by calling `k_condvar_wait()`.

The following code waits on the condition variable.

```
K_MUTEX_DEFINE(mutex);
K_CONDVAR_DEFINE(condvar)

int main(void)
{
    k_mutex_lock(&mutex, K_FOREVER);

    /* block this thread until another thread signals cond. While
     * blocked, the mutex is released, then re-acquired before this
     * thread is woken up and the call returns.
     */
    k_condvar_wait(&condvar, &mutex, K_FOREVER);
    ...
    k_mutex_unlock(&mutex);
}
```

Signaling a Condition Variable A condition variable is signaled on by calling `k_condvar_signal()` for one thread or by calling `k_condvar_broadcast()` for multiple threads.

The following code builds on the example above.

```
void worker_thread(void)
{
    k_mutex_lock(&mutex, K_FOREVER);

    /*
     * Do some work and fulfill the condition
     */
    ...
    ...
    k_condvar_signal(&condvar);
    k_mutex_unlock(&mutex);
}
```

Suggested Uses Use condition variables with a mutex to signal changing states (conditions) from one thread to another thread. Condition variables are not the condition itself and they are not events. The condition is contained in the surrounding programming logic.

Mutexes alone are not designed for use as a notification/synchronization mechanism. They are meant to provide mutually exclusive access to a shared resource only.

Configuration Options Related configuration options:

- None.

API Reference

Condition Variables APIs

Related code samples

- Condition VariablesManipulate condition variables in a multithreaded application.

Events

An *event object* is a kernel object that implements traditional events.

- *Concepts*
- *Implementation*
  - *Defining an Event Object*
  - *Setting Events*
  - *Posting Events*
  - *Waiting for Events (without removal)*
  - *Waiting for Events (with removal)*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of event objects can be defined (limited only by available RAM). Each event object is referenced by its memory address. One or more threads may wait on an event object until the desired set of events has been delivered to the event object. When new events are delivered to the event object, all threads whose wait conditions have been satisfied become ready simultaneously.

An event object has the following key properties:

- A 32-bit value that tracks which events have been delivered to it.

An event object must be initialized before it can be used.

Events may be **delivered** by a thread or an ISR. When delivering events, the events may either overwrite the existing set of events or add to them in a bitwise fashion. When overwriting the existing set of events, this is referred to as setting. When adding to them in a bitwise fashion, this is referred to as posting. Both posting and setting events have the potential to fulfill match conditions of multiple threads waiting on the event object. All threads whose match conditions have been met are made active at the same time.

Threads may wait on one or more events. They may either wait for all of the requested events, or for any of them. Furthermore, threads making a wait request have the option of resetting the current set of events tracked by the event object prior to waiting. Care must be taken with this option when multiple threads wait on the same event object.

Note

The kernel does allow an ISR to query an event object, however the ISR must not attempt to wait for the events.

Implementation

Defining an Event Object An event object is defined using a variable of type `k_event`. It must then be initialized by calling `k_event_init()`.

The following code defines an event object.

```
struct k_event my_event;

k_event_init(&my_event);
```

Alternatively, an event object can be defined and initialized at compile time by calling `K_EVENT_DEFINE`.

The following code has the same effect as the code segment above.

```
K_EVENT_DEFINE(my_event);
```

Setting Events Events in an event object are set by calling `k_event_set()`.

The following code builds on the example above, and sets the events tracked by the event object to 0x001.

```
void input_available_interrupt_handler(void *arg)
{
    /* notify threads that data is available */

    k_event_set(&my_event, 0x001);

    ...
}
```

Posting Events Events are posted to an event object by calling `k_event_post()`.

The following code builds on the example above, and posts a set of events to the event object.

```
void input_available_interrupt_handler(void *arg)
{
    ...

    /* notify threads that more data is available */

    k_event_post(&my_event, 0x120);

    ...
}
```

Waiting for Events (without removal) Threads wait for events by calling `k_event_wait()`.

The following code builds on the example above, and waits up to 50 milliseconds for any of the specified events to be posted. A warning is issued if none of the events are posted in time.

```
void consumer_thread(void)
{
    uint32_t  events;

    events = k_event_wait(&my_event, 0xFFF, false, K_MSEC(50));
    if (events == 0) {
        printk("No input devices are available!");
    } else {
```

(continues on next page)

(continued from previous page)

```
        /* Access the desired input device(s) */
        ...
    }
    ...
}
```

Alternatively, the consumer thread may desire to wait for all the events before continuing using `k_event_wait_all()`.

```
void consumer_thread(void)
{
    uint32_t  events;

    events = k_event_wait_all(&my_event, 0x121, false, K_MSEC(50));
    if (events == 0) {
        printk("At least one input device is not available!");
    } else {
        /* Access the desired input devices */
        ...
    }
    ...
}
```

Waiting for Events (with removal) Threads wait for events (with atomic removal upon receipt) by calling `k_event_wait_safe()`.

The following code builds on the example above, and waits up to 50 milliseconds for any of the specified events to be posted. A warning is issued if none of the events are posted in time.

If events are received on time, then they will not be present in the event object until the next time that the events are set or posted.

```
void consumer_thread(void)
{
    uint32_t  events;

    events = k_event_wait_safe(&my_event, 0xFFF, false, K_MSEC(50));
    if (events == 0) {
        printk("No input devices are available!");
    } else {
        /* Access the desired input device(s) */
        ...
    }
    ...
}
```

Alternatively, the consumer thread may desire to wait for all the events (with atomic removal upon receipt) before continuing using `k_event_wait_all_safe()`.

If all events are received on time, then they will not be present in the event object until the next time that the events are set or posted.

```
void consumer_thread(void)
{
    uint32_t  events;

    events = k_event_wait_all_safe(&my_event, 0x121, false, K_MSEC(50));
```

(continues on next page)

(continued from previous page)

```
    if (events == 0) {
        printk("At least one input device is not available!");
    } else {
        /* Access the desired input devices */
        ...
    }
    ...
}
```

Suggested Uses Use events to indicate that a set of conditions have occurred.

Use events to pass small amounts of data to multiple threads at once.

Configuration Options Related configuration options:

- `CONFIG_EVENTS`

API Reference

Event APIs

Spinlocks

- *Usage*
  - *Acquiring without spinning*
  - *Rules*
- *Spinlocks on uniprocessor systems*
- *Spinlock validation*
- *Fair spinlocks*
- *Suggested uses*
- *API reference*

A spinlock is the lowest-level mutual-exclusion primitive in Zephyr. It guards a brief critical section so that only one execution context can access a shared resource at a time. A context that finds the lock already held "spins", busy waiting until the lock becomes available, rather than blocking and yielding the CPU. This is why spinlocks are only appropriate for short critical sections.

Acquiring a spinlock also masks interrupts on the local CPU for as long as the lock is held. This is what makes a spinlock safe to share between threads and interrupt handlers: while a context holds the lock, no ISR on the same CPU can preempt it and no other CPU can take the same lock, so neither can observe or corrupt the protected data mid-update. For these reasons, `k_spinlock` is the primary synchronization primitive used within the Zephyr kernel itself, protecting access to its core data structures.

Application code may use it directly as well, but it is generally recommended to use a higher-level synchronization primitive such as a `k_mutex` or `k_sem` instead.

Usage Declare a `k_spinlock` for each independent resource you need to protect. Acquire it with `k_spin_lock()`, which returns a `k_spinlock_key_t` that must be passed back to `k_spin_unlock()` to release the lock:

```
static struct k_spinlock lock;

void update_shared_state(void)
{
        k_spinlock_key_t key = k_spin_lock(&lock);

        /* critical section: exclusive access */

        k_spin_unlock(&lock, key);
}
```

The `K_SPINLOCK` helper acquires the lock for the duration of the enclosed block and releases it automatically on exiting the block.

```
K_SPINLOCK(&lock) {
        /* critical section: exclusive access */
}
```

The block must either run to its end or be left with `K_SPINLOCK_BREAK`. Leaving it with a plain `break`, `goto` or `return` skips the release and leaks the lock:

```
K_SPINLOCK(&lock) {
        if (nothing_to_do) {
                K_SPINLOCK_BREAK;
        }

        /* critical section: exclusive access */
}
```

Acquiring without spinning `k_spin_trylock()` makes a single attempt to take the lock and reports failure instead of waiting, which is useful when the caller has other work to do or must not stall. On success it stores the key, which is released exactly as for `k_spin_lock()`:

```
k_spinlock_key_t key;

if (k_spin_trylock(&lock, &key) == 0) {
        /* critical section: exclusive access */
        k_spin_unlock(&lock, key);
} else {
        /* lock is held elsewhere, do something else */
}
```

Rules Follow these rules when using a spinlock:

- Keep the critical section as short as possible. Interrupts are masked on the local CPU while the lock is held, and other CPUs may be spinning on it.
- Never perform a blocking or sleeping operation while holding a spinlock.
- Do not acquire a spinlock recursively. A context that already holds a lock must not try to take it again, or it will deadlock. Nesting **distinct** spinlocks is allowed, but must follow a consistent lock ordering to avoid deadlock.

Spinlocks on uniprocessor systems In a kernel built with `CONFIG_SMP` disabled, a spinlock does not actually spin. With only one CPU there is nothing to contend with, so `k_spin_lock()` reduces to masking interrupts on the local CPU. This prevents both ISRs and context switches from observing the protected data mid-update, which is all that is needed for mutual exclusion on a uniprocessor.

Spinlock validation A validation layer, enabled with `CONFIG_SPIN_VALIDATE`, is available to detect various misuses of spinlocks. It can detect the following:

- Recursive acquisition of a spinlock
- Release of a spinlock that is not held by the current context
- Out-of-order release of spinlocks (an outer lock released while an inner lock is still held)
- Context switching while a spinlock is held, or while interrupts are masked by a nested `irq_lock()`

On a uniprocessor system, only the recursive acquisition check is meaningful. The other misuses cannot occur, since there is no contention and interrupts are masked while the lock is held.

Fair spinlocks The default spinlock implementation is built on a single `atomic_t` variable and does not guarantee fairness between contending CPUs: It is possible for one CPU to repeatedly win the contention, thus starving the others. Where this matters, enabling `CONFIG_TICKET_SPINLOCKS` switches to a ticket-based implementation that grants a contended lock to requesting CPUs in FIFO order at the cost of a slightly larger lock object.

Suggested uses Use a spinlock to protect a short critical section that is shared between threads and interrupt handlers, or between CPUs on an SMP system.

Prefer another primitive when it fits better:

- `k_mutex` for a critical section that may be long, may block, or may sleep. Only threads can take a mutex.
- `k_sem` for signalling between contexts, and for counting access to a pool of resources.
- *atomic services* when the shared state is a single word that can be updated with one atomic operation, in which case no lock is needed.

API reference
Spinlock APIs

Symmetric Multiprocessing

On multiprocessor architectures, Zephyr supports the use of multiple physical CPUs running Zephyr application code. This support is "symmetric" in the sense that no specific CPU is treated specially by default. Any processor is capable of running any Zephyr thread, with access to all standard Zephyr APIs supported.

No special application code needs to be written to take advantage of this feature. If there are two Zephyr application threads runnable on a supported dual processor device, they will both run simultaneously.

SMP configuration is controlled under the `CONFIG_SMP` kconfig variable. This must be set to "y" to enable SMP features, otherwise a uniprocessor kernel will be built. In general the platform default will have enabled this anywhere it's supported. When enabled, the number of physical CPUs available is visible at build time as `CONFIG_MP_MAX_NUM_CPUS`. Likewise, the default for this will be the number of available CPUs on the platform and it is not expected that typical apps will change it. But it is legal and supported to set this to a smaller (but obviously not larger) number for special purposes (e.g. for testing, or to reserve a physical CPU for running non-Zephyr code).

Synchronization At the application level, core Zephyr IPC and synchronization primitives all behave identically under an SMP kernel. For example semaphores used to implement blocking mutual exclusion continue to be a proper application choice.

At the lowest level, however, Zephyr code has often used the `irq_lock()`/`irq_unlock()` primitives to implement fine grained critical sections using interrupt masking. These APIs continue to

work via an emulation layer (see below), but the masking technique does not: the fact that your CPU will not be interrupted while you are in your critical section says nothing about whether a different CPU will be running simultaneously and be inspecting or modifying the same data!

Spinlocks SMP systems provide a more constrained `k_spin_lock()` primitive that masks interrupts locally, like `irq_lock()`, while also atomically guarding a shared lock variable so that only one CPU enters the critical section at a time. Unlike the legacy IRQ locks, spinlocks are separable and must not be acquired recursively. See *Spinlocks* for a full description, including ticket spinlocks and the API reference.

Legacy irq_lock() emulation For the benefit of applications written to the uniprocessor locking API, `irq_lock()` and `irq_unlock()` continue to work compatibly on SMP systems with identical semantics to their legacy versions. They are implemented as a single global spinlock, with a nesting count and the ability to be atomically reacquired on context switch into locked threads. The kernel will ensure that only one thread across all CPUs can hold the lock at any time, that it is released on context switch, and that it is re-acquired when necessary to restore the lock state when a thread is switched in. Other CPUs will spin waiting for the release to happen.

The overhead involved in this process has measurable performance impact, however. Unlike uniprocessor apps, SMP apps using `irq_lock()` are not simply invoking a very short (often ~1 instruction) interrupt masking operation. That, and the fact that the IRQ lock is global, means that code expecting to be run in an SMP context should be using the spinlock API wherever possible.

Memory Coherence Some multiprocessor architectures are *cache-incoherent*: the per-CPU caches are not automatically kept consistent with each other, so data written by one CPU may not be visible to another until it is flushed from the cache. On such systems, shared kernel data structures must reside in memory that all CPUs observe consistently.

When `CONFIG_KERNEL_COHERENCE` is enabled, the kernel places all shared data into multiprocessor-coherent (generally uncached) memory. Thread stacks remain cached, as does application memory explicitly declared with `__incoherent`. This mode is intended only for SMP kernels running on cache-incoherent architectures, and it carries an implicit API contract: any memory passed to the kernel is assumed to be cache-coherent, so kernel data structures must not be created in uncached regions.

CPU Mask It is often desirable for real time applications to deliberately partition work across physical CPUs instead of relying solely on the kernel scheduler to decide on which threads to execute. Zephyr provides an API, controlled by the `CONFIG_SCHED_CPU_MASK` kconfig variable, which can associate a specific set of CPUs with each thread, indicating on which CPUs it can run.

By default, new threads can run on any CPU. Calling `k_thread_cpu_mask_disable()` with a particular CPU ID will prevent that thread from running on that CPU in the future. Likewise `k_thread_cpu_mask_enable()` will re-enable execution. There are also `k_thread_cpu_mask_clear()` and `k_thread_cpu_mask_enable_all()` APIs available for convenience. For obvious reasons, these APIs are illegal if called on a runnable thread. The thread must be blocked or suspended, otherwise an `-EINVAL` will be returned.

CPU mask filtering is supported with all three scheduler backends. The performance impact differs by backend:

- `CONFIG_SCHED_SIMPLE` — O(N) linear scan of the run queue; every context switch walks the full list looking for the first eligible thread.
- `CONFIG_SCHED_SCALABLE` — O(N) in-order walk of the red/black tree; priority ordering is preserved but the full tree may be traversed when many threads are masked off.
- `CONFIG_SCHED_MULTIQ` — scans priority buckets from highest to lowest and walks the per-bucket list; worst case is O(P·N) where P is the number of occupied priority levels.

For workloads that use CONFIG_SCHED_CPU_MASK_PIN_ONLY, each CPU maintains its own independent run queue, so the scheduler needs only examine that queue with no mask filtering overhead.

Note that k_thread_cpu_mask_clear(), k_thread_cpu_mask_enable_all(), and k_thread_cpu_mask_disable() are not permitted in CONFIG_SCHED_CPU_MASK_PIN_ONLY mode because they can produce a mask that is not exactly one bit, which violates the invariant that every thread is pinned to precisely one CPU.

SMP Boot Process A Zephyr SMP kernel begins boot identically to a uniprocessor kernel. Auxiliary CPUs begin in a disabled state in the architecture layer. All standard kernel initialization, including device initialization, happens on a single CPU before other CPUs are brought online.

Just before entering the application main() function, the kernel calls z_smp_init() to launch the SMP initialization process. This enumerates over the configured CPUs, calling into the architecture layer using arch_cpu_start() for each one. This function is passed a memory region to use as a stack on the foreign CPU (in practice it uses the area that will become that CPU's interrupt stack), the address of a local smp_init_top() callback function to run on that CPU, and a pointer to a "start flag" address which will be used as an atomic signal.

The local SMP initialization (smp_init_top()) on each CPU is then invoked by the architecture layer. Note that interrupts are still masked at this point. This routine is responsible for calling smp_timer_init() to set up any needed stat in the timer driver. On many architectures the timer is a per-CPU device and needs to be configured specially on auxiliary CPUs. Then it waits (spinning) for the atomic "start flag" to be released in the main thread, to guarantee that all SMP initialization is complete before any Zephyr application code runs, and finally calls z_swap() to transfer control to the appropriate runnable thread via the standard scheduler API.

By default the kernel brings up every available CPU during this start-up sequence. A CPU whose devicetree node carries the zephyr,deferred-start flag is skipped and left disabled so that architecture, SoC, board, or application code can start it later at run time; deferral is per CPU, so a system may bring up some secondary CPUs at boot and leave others to be started on demand. A deferred CPU is started with k_smp_cpu_start(), which performs full per-CPU initialization; k_smp_cpu_resume() is the counterpart used to bring a previously stopped CPU back online without repeating one-time initialization.

Interprocessor Interrupts When running in multiprocessor environments, it is occasionally the case that state modified on the local CPU needs to be synchronously handled on a different processor.

One example is the Zephyr k_thread_abort() API, which cannot return until the thread that had been aborted is no longer runnable. If it is currently running on another CPU, that becomes difficult to implement.

Another is low power idle. It is a firm requirement on many devices that system idle be implemented using a low-power mode with as many interrupts (including periodic timer interrupts) disabled or deferred as is possible. If a CPU is in such a state, and on another CPU a thread becomes runnable, the idle CPU has no way to "wake up" to handle the newly-runnable load.

So where possible, Zephyr SMP architectures should implement an interprocessor interrupt. The current framework is very simple: the architecture provides at least a arch_sched_broadcast_ipi() call, which when invoked will flag an interrupt on all CPUs (except the current one, though that is allowed behavior). If the architecture supports directed IPIs (see CONFIG_ARCH_HAS_DIRECTED_IPIS), then the architecture also provides a arch_sched_directed_ipi() call, which when invoked will flag an interrupt on the specified CPUs. When an interrupt is flagged on the CPUs, the z_sched_ipi() function implemented in the scheduler will get invoked on those CPUs. The expectation is that these APIs will evolve over time to encompass more functionality (e.g. cross-CPU calls), and that the scheduler-specific calls here will be implemented in terms of a more general framework.

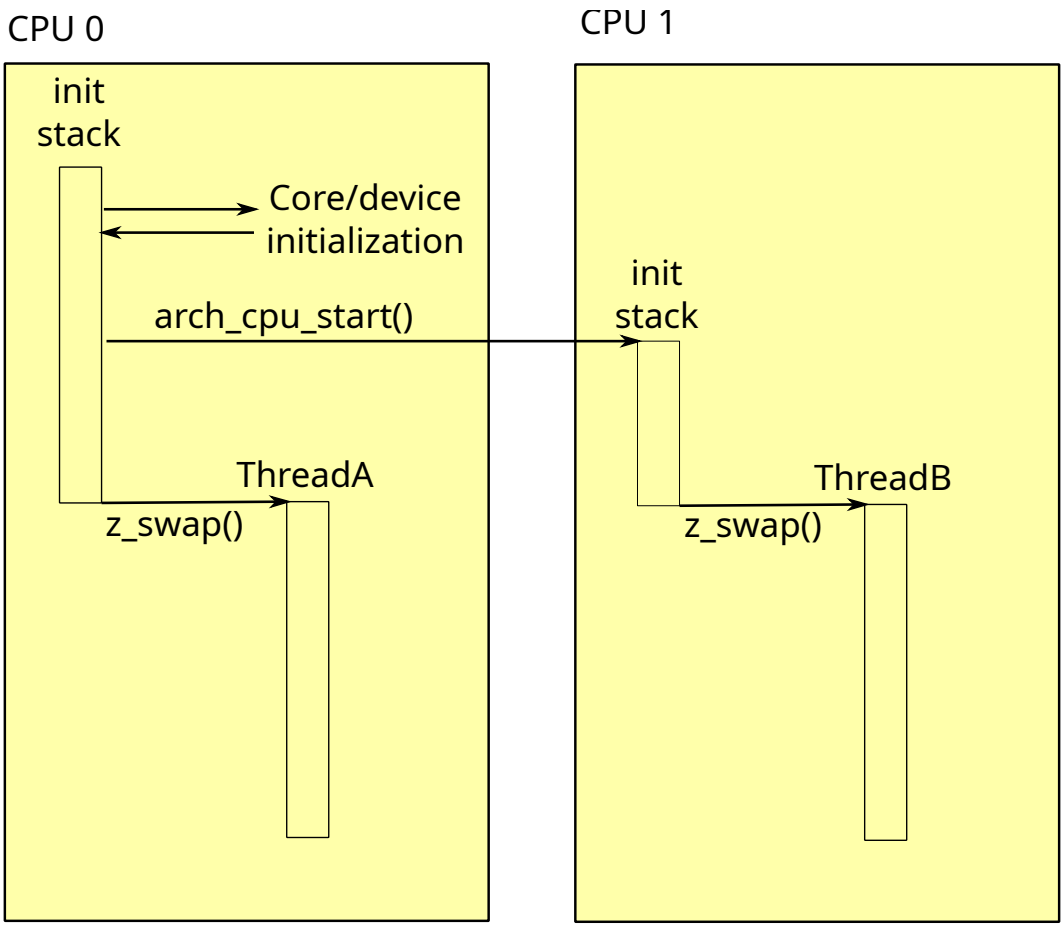


Fig. 1: Example SMP initialization process, showing a configuration with two CPUs and two app threads which begin operating simultaneously.

When directed IPIs are available, the scheduler signals only those CPUs that actually need to reschedule when a thread becomes ready, rather than broadcasting to every other CPU. This avoids disturbing CPUs whose currently running thread does not need to be preempted, reducing the overall interrupt load.

Note that not all SMP architectures will have a usable IPI mechanism (either missing, or just undocumented/unimplemented). In those cases Zephyr provides fallback behavior that is correct, but perhaps suboptimal.

Using this, `k_thread_abort()` becomes only slightly more complicated in SMP: for the case where a thread is actually running on another CPU (we can detect this atomically inside the scheduler), we broadcast an IPI and spin, waiting for the thread to either become "DEAD" or for it to re-enter the queue (in which case we terminate it the same way we would have in uniprocessor mode). Note that the "aborted" check happens on any interrupt exit, so there is no special handling needed in the IPI per se. This allows us to implement a reasonable fallback when IPI is not available: we can simply spin, waiting until the foreign CPU receives any interrupt, though this may be a much longer time!

Likewise idle wakeups are trivially implementable with an empty IPI handler. If a thread is added to an empty run queue (i.e. there may have been idle CPUs), we broadcast an IPI. A foreign CPU will then be able to see the new thread when exiting from the interrupt and will switch to it if available.

Without an IPI, however, a low power idle that requires an interrupt will not work to synchronously run new threads. The workaround in that case is more invasive: Zephyr will **not** enter the system idle handler and will instead spin in its idle loop, testing the scheduler state at high frequency (not spinning on it though, as that would involve severe lock contention) for new threads. The expectation is that power constrained SMP applications are always going to provide an IPI, and this code will only be used for testing purposes or on systems without power consumption requirements.

IPI Cascades The kernel can not control the order in which IPIs are processed by the CPUs in the system. In general, this is not an issue and a single set of IPIs is sufficient to trigger a reschedule on the N CPUs that results with them scheduling the highest N priority ready threads to execute. When CPU masking is used, there may be more than one valid set of threads (not to be confused with an optimal set of threads) that can be scheduled on the N CPUs and a single set of IPIs may be insufficient to result in any of these valid sets.

> **Note**
>
> When CPU masking is not in play, the optimal set of threads is the same as the valid set of threads. However when CPU masking is in play, there may be more than one valid set–one of which may be optimal.
>
> To better illustrate the distinction, consider a 2-CPU system with ready threads T1 and T2 at priorities 1 and 2 respectively. Let T2 be pinned to CPU0 and T1 not be pinned. If CPU0 is executing T2 and CPU1 executing T1, then this set is is both valid and optimal. However, if CPU0 is executing T1 and CPU1 is idling, then this too would be valid though not optimal.

In those cases where a single set of IPIs is not sufficient to generate a valid set, the resulting set of executing threads are expected to be close to a valid set, and subsequent IPIs can generally be expected to correct the situation soon. However, for cases where neither the approximation nor the delay are acceptable, enabling `CONFIG_SCHED_IPI_CASCADE` will allow the kernel to generate cascading IPIs until the kernel has selected a valid set of ready threads for the CPUs.

There are three types of costs/penalties associated with the IPI cascades–and for these reasons they are disabled by default. The first is a cost incurred by the CPU producing the IPI when a new thread preempts the old thread as checks must be done to compare the old thread against the threads executing on the other CPUs. The second is a cost incurred by the CPUs receiving the

IPIs as they must be processed. The third is the apparent sputtering of a thread as it "winks in" and then "winks out" due to cascades stemming from the aforementioned first cost.

IPI Work Items The kernel allows developers to execute functions on other CPUs at ISR level using one or more IPI work items. After IPI work items have been added to the specified CPUs' work queues using `k_ipi_work_add()`, the targeted CPUs will process them after receiving an IPI. Signaling an IPI is done by calling `k_ipi_work_signal()`. Waiting for an IPI work item to be completed by the targeted CPUs is done by calling `k_ipi_work_wait()`. Only a single waiter is permitted at a time.

> **Note**
>
> IPI work items will only be added to the IPI work queues of other CPUs. If adding IPI work items at thread level, the developer must ensure that the current thread does not change CPUs until after signaling the IPIs.

Sample Use The following code outlines how to use IPI work items to update status information across a set of CPUs as a result of one CPU handling an ISR.

```
struct k_ipi_work my_work;

void remote_cpu_action(struct k_ipi_work *arg)
{
    ...
}

void my_isr(void)
{
    /*
     * Wait for previous use of <my_work> to complete.
     * It assumes that <my_work> was initialized elsewhere.
     */

    uint32_t cpu_mask = <bitmask identifying CPUs to update>;

    while (k_ipi_work_wait(&my_work, K_NO_WAIT) == -EAGAIN) {
    }

    /* Add and signal the new work */

    k_ipi_work_add(&my_work, cpu_mask, remote_cpu_action);

    k_ipi_work_signal();
}
```

SMP Kernel Internals In general, Zephyr kernel code is SMP-agnostic and, like application code, will work correctly regardless of the number of CPUs available. But in a few areas there are notable changes in structure or behavior.

Per-CPU data Many elements of the core kernel data need to be implemented for each CPU in SMP mode. For example, the `_current` thread pointer obviously needs to reflect what is running locally, there are many threads running concurrently. Likewise a kernel-provided interrupt stack needs to be created and assigned for each physical CPU, as does the interrupt nesting count used to detect ISR state.

These fields are now moved into a separate struct _cpu instance within the _kernel struct, which has a cpus[] array indexed by ID. Compatibility fields are provided for legacy uniprocessor code trying to access the fields of cpus[0] using the older syntax and assembly offsets.

Note that an important requirement on the architecture layer is that the pointer to this CPU struct be available rapidly when in kernel context. The expectation is that arch_curr_cpu() will be implemented using a CPU-provided register or addressing mode that can store this value across arbitrary context switches or interrupts and make it available to any kernel-mode code.

Similarly, where on a uniprocessor system Zephyr could simply create a global "idle thread" at the lowest priority, in SMP we may need one for each CPU. This makes the internal predicate test for "_is_idle()" in the scheduler, which is a hot path performance environment, more complicated than simply testing the thread pointer for equality with a known static variable. In SMP mode, idle threads are distinguished by a separate field in the thread struct.

Switch-based context switching The traditional Zephyr context switch primitive has been z_swap(). Unfortunately, this function takes no argument specifying a thread to switch to. The expectation has always been that the scheduler has already made its preemption decision when its state was last modified and cached the resulting "next thread" pointer in a location where architecture context switch primitives can find it via a simple struct offset. That technique will not work in SMP, because the other CPU may have modified scheduler state since the current CPU last exited the scheduler (for example: it might already be running that cached thread!).

Instead, the SMP "switch to" decision needs to be made synchronously with the swap call, and as we don't want per-architecture assembly code to be handling scheduler internal state, Zephyr requires a somewhat lower-level context switch primitives for SMP systems: arch_switch() is always called with interrupts masked, and takes exactly two arguments. The first is an opaque (architecture defined) handle to the context to which it should switch, and the second is a pointer to such a handle into which it should store the handle resulting from the thread that is being switched out. The kernel then implements a portable z_swap() implementation on top of this primitive which includes the relevant scheduler logic in a location where the architecture doesn't need to understand it.

Similarly, on interrupt exit, switch-based architectures are expected to call z_get_next_switch_handle() to retrieve the next thread to run from the scheduler. The argument to z_get_next_switch_handle() is either the interrupted thread's "handle" reflecting the same opaque type used by arch_switch(), or NULL if that thread cannot be released to the scheduler just yet. The choice between a handle value or NULL depends on the way CPU interrupt mode is implemented.

Architectures with a large CPU register file would typically preserve only the caller-saved registers on the current thread's stack when interrupted in order to minimize interrupt latency, and preserve the callee-saved registers only when arch_switch() is called to minimize context switching latency. Such architectures must use NULL as the argument to z_get_next_switch_handle() to determine if there is a new thread to schedule, and follow through with their own arch_switch() or derivative if so, or directly leave interrupt mode otherwise. In the former case it is up to that switch code to store the handle resulting from the thread that is being switched out in that thread's "switch_handle" field after its context has fully been saved.

Architectures whose entry in interrupt mode already preserves the entire thread state may pass that thread's handle directly to z_get_next_switch_handle() and be done in one step.

Note that while SMP requires CONFIG_USE_SWITCH, the reverse is not true. A uniprocessor architecture built with CONFIG_SMP set to No might still decide to implement its context switching using arch_switch().

Thread Queue Ordering When a thread is preempted by another thread of higher priority in an SMP system, it is moved to the back of the queue for its priority level. In contrast, on UP systems, a preempted thread does not move to the back of its priority queue; it simply waits to regain the CPU, preserving its original ordering relative to other threads of the same priority.

3.1.2 Data Passing

These pages cover kernel objects which can be used to pass data between threads and ISRs.

The following table summarizes their high-level features.

Object	Bidirectional?	Data structure	Data item size	Data Alignment	ISRs can receive?	ISRs can send?	Overrun handling
FIFO	No	Queue	Arbitrary[1]	4 B[2]	Yes[3]	Yes	N/A
LIFO	No	Queue	Arbitrary[1]	4 B[2]	Yes[3]	Yes	N/A
Stack	No	Array	Word	Word	Yes[3]	Yes	Undefined behavior
Message queue	No	Ring buffer	Arbitrary[6]	Power of two	Yes[3]	Yes	Pend thread or return -errno
Mailbox	Yes	Queue	Arbitrary[1]	Arbitrary	No	No	N/A
Pipe	No	Ring buffer[4]	Arbitrary	Arbitrary	Yes[5]	Yes[5]	Pend thread or return -errno

Queues

A Queue in Zephyr is a kernel object that implements a traditional queue, allowing threads and ISRs to add and remove data items of any size. The queue is similar to a FIFO and serves as the underlying implementation for both *k_fifo* and *k_lifo*. For more information on usage see *k_fifo*.

Cancelling a Wait A thread that is blocked waiting to retrieve an item from a queue can be released without an item by calling `k_queue_cancel_wait()` from another thread or an ISR. The first thread pending on the queue returns from its `k_queue_get()` call with a `NULL` value, exactly as if its timeout had expired. If the queue is instead being waited on through `k_poll()`, that call returns `-EINTR` with the poll event in the cancelled state.

This mechanism is also available to the queue-based primitives through `k_fifo_cancel_wait` and `k_lifo_cancel_wait`.

Configuration Options Related configuration options:

- None

API Reference
Queue APIs

FIFOs

A *FIFO* is a kernel object that implements a traditional first in, first out (FIFO) queue, allowing threads and ISRs to add and remove data items of any size.

[1] Callers allocate space for queue overhead in the data elements themselves.
[2] Objects added with `k_fifo_alloc_put()` and `k_lifo_alloc_put()` do not have alignment constraints, but use temporary memory from the calling thread's resource pool.
[3] ISRs can receive only when passing K_NO_WAIT as the timeout argument.
[6] Data item size must be a multiple of the data alignment.
[4] Optional.
[5] ISRs can send and/or receive only when passing K_NO_WAIT as the timeout argument.

- *Concepts*
- *Implementation*
  - *Defining a FIFO*
  - *Writing to a FIFO*
  - *Reading from a FIFO*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of FIFOs can be defined (limited only by available RAM). Each FIFO is referenced by its memory address.

A FIFO has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A FIFO must be initialized before it can be used. This sets its queue to empty.

FIFO data items must be aligned as required for a pointer, as the kernel reserves the first pointer-sized field of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 (or N+8) bytes of memory. There are no alignment or reserved space requirements for data items if they are added with `k_fifo_alloc_put()`, instead additional memory is temporarily allocated from the calling thread's resource pool.

> **Note**
>
> FIFO data items are restricted to single active instance across all FIFO data queues. Any attempt to re-add a FIFO data item to a queue before it has been removed from the queue to which it was previously added will result in undefined behavior.

A data item may be **added** to a FIFO by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the FIFO's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a FIFO by a thread. If the FIFO's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty FIFO simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

> **Note**
>
> The kernel does allow an ISR to remove an item from a FIFO, however the ISR must not attempt to wait if the FIFO is empty.

If desired, **multiple data items** can be added to a FIFO in a single operation if they are chained together into a singly-linked list. This capability can be useful if multiple writers are adding sets of related data items to the FIFO, as it ensures the data items in each set are not interleaved with other data items. Adding multiple data items to a FIFO is also more efficient than adding them

one at a time, and can be used to guarantee that anyone who removes the first data item in a set will be able to remove the remaining data items without waiting.

Implementation

Defining a FIFO A FIFO is defined using a variable of type `k_fifo`. It must then be initialized by calling `k_fifo_init()`.

The following code defines and initializes an empty FIFO.

```
struct k_fifo my_fifo;

k_fifo_init(&my_fifo);
```

Alternatively, an empty FIFO can be defined and initialized at compile time by calling `K_FIFO_DEFINE`.

The following code has the same effect as the code segment above.

```
K_FIFO_DEFINE(my_fifo);
```

Writing to a FIFO A data item is added to a FIFO by calling `k_fifo_put()`.

The following code builds on the example above, and uses the FIFO to send data to one or more consumer threads.

```
struct data_item_t {
    void *fifo_reserved;   /* 1st word reserved for use by FIFO */
    ...
};

struct data_item_t tx_data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_fifo_put(&my_fifo, &tx_data);

        ...
    }
}
```

Additionally, a singly-linked list of data items can be added to a FIFO by calling `k_fifo_put_list()` or `k_fifo_put_slist()`.

Finally, a data item can be added to a FIFO with `k_fifo_alloc_put()`. With this API, there is no need to reserve space for the kernel's use in the data item, instead additional memory will be allocated from the calling thread's resource pool until the item is read.

Reading from a FIFO A data item is removed from a FIFO by calling `k_fifo_get()`.

The following code builds on the example above, and uses the FIFO to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_fifo_get(&my_fifo, K_FOREVER);

        /* process FIFO data item */
        ...
    }
}
```

Suggested Uses Use a FIFO to asynchronously transfer data items of arbitrary size in a "first in, first out" manner.

Configuration Options Related configuration options:

- None

API Reference

FIFO APIs

Related code samples

- Dining PhilosophersImplement a solution to the Dining Philosophers problem using Zephyr kernel services.

LIFOs

A *LIFO* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove data items of any size.

- *Concepts*
- *Implementation*
  - *Defining a LIFO*
  - *Writing to a LIFO*
  - *Reading from a LIFO*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of LIFOs can be defined (limited only by available RAM). Each LIFO is referenced by its memory address.

A LIFO has the following key properties:

- A **queue** of data items that have been added but not yet removed. The queue is implemented as a simple linked list.

A LIFO must be initialized before it can be used. This sets its queue to empty.

LIFO data items must be aligned as required for a pointer, as the kernel reserves the first pointer-sized field of an item for use as a pointer to the next data item in the queue. Consequently, a data item that holds N bytes of application data requires N+4 (or N+8) bytes of memory. There are no alignment or reserved space requirements for data items if they are added with `k_lifo_alloc_put()`, instead additional memory is temporarily allocated from the calling thread's resource pool.

> **Note**
>
> LIFO data items are restricted to single active instance across all LIFO data queues. Any attempt to re-add a LIFO data item to a queue before it has been removed from the queue to which it was previously added will result in undefined behavior.

A data item may be **added** to a LIFO by a thread or an ISR. The item is given directly to a waiting thread, if one exists; otherwise the item is added to the LIFO's queue. There is no limit to the number of items that may be queued.

A data item may be **removed** from a LIFO by a thread. If the LIFO's queue is empty a thread may choose to wait for a data item to be given. Any number of threads may wait on an empty LIFO simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

> **Note**
>
> The kernel does allow an ISR to remove an item from a LIFO, however the ISR must not attempt to wait if the LIFO is empty.

Implementation

Defining a LIFO A LIFO is defined using a variable of type `k_lifo`. It must then be initialized by calling `k_lifo_init()`.

The following defines and initializes an empty LIFO.

```
struct k_lifo my_lifo;

k_lifo_init(&my_lifo);
```

Alternatively, an empty LIFO can be defined and initialized at compile time by calling `K_LIFO_DEFINE`.

The following code has the same effect as the code segment above.

```
K_LIFO_DEFINE(my_lifo);
```

Writing to a LIFO A data item is added to a LIFO by calling `k_lifo_put()`.

The following code builds on the example above, and uses the LIFO to send data to one or more consumer threads.

```
struct data_item_t {
    void *LIFO_reserved;   /* 1st word reserved for use by LIFO */
    ...
};
```

(continues on next page)

(continued from previous page)

```
struct data_item_t tx data;

void producer_thread(int unused1, int unused2, int unused3)
{
    while (1) {
        /* create data item to send */
        tx_data = ...

        /* send data to consumers */
        k_lifo_put(&my_lifo, &tx_data);

        ...
    }
}
```

A data item can be added to a LIFO with `k_lifo_alloc_put()`. With this API, there is no need to reserve space for the kernel's use in the data item, instead additional memory will be allocated from the calling thread's resource pool until the item is read.

Reading from a LIFO A data item is removed from a LIFO by calling `k_lifo_get()`.

The following code builds on the example above, and uses the LIFO to obtain data items from a producer thread, which are then processed in some manner.

```
void consumer_thread(int unused1, int unused2, int unused3)
{
    struct data_item_t  *rx_data;

    while (1) {
        rx_data = k_lifo_get(&my_lifo, K_FOREVER);

        /* process LIFO data item */
        ...
    }
}
```

Suggested Uses Use a LIFO to asynchronously transfer data items of arbitrary size in a "last in, first out" manner.

Configuration Options Related configuration options:

- None.

API Reference

LIFO APIs

Related code samples

- Dining PhilosophersImplement a solution to the Dining Philosophers problem using Zephyr kernel services.

Stacks

A *stack* is a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove a limited number of integer data values.

- *Concepts*
- *Implementation*
  - *Defining a Stack*
  - *Pushing to a Stack*
  - *Popping from a Stack*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of stacks can be defined (limited only by available RAM). Each stack is referenced by its memory address.

A stack has the following key properties:

- A **queue** of integer data values that have been added but not yet removed. The queue is implemented using an array of stack_data_t values and must be aligned on a native word boundary. The stack_data_t type corresponds to the native word size i.e. 32 bits or 64 bits depending on the CPU architecture and compilation mode.
- A **maximum quantity** of data values that can be queued in the array.

A stack must be initialized before it can be used. This sets its queue to empty.

A data value can be **added** to a stack by a thread or an ISR. The value is given directly to a waiting thread, if one exists; otherwise the value is added to the LIFO's queue.

Note

If `CONFIG_NO_RUNTIME_CHECKS` is enabled, the kernel will *not* detect and prevent attempts to add a data value to a stack that has already reached its maximum quantity of queued values. Adding a data value to a stack that is already full will result in array overflow, and lead to unpredictable behavior.

A data value may be **removed** from a stack by a thread. If the stack's queue is empty a thread may choose to wait for it to be given. Any number of threads may wait on an empty stack simultaneously. When a data item is added, it is given to the highest priority thread that has waited longest.

Note

The kernel does allow an ISR to remove an item from a stack, however the ISR must not attempt to wait if the stack is empty.

Implementation

Defining a Stack A stack is defined using a variable of type `k_stack`. It must then be initialized by calling `k_stack_init()` or `k_stack_alloc_init()`. In the latter case, a buffer is not provided and it is instead allocated from the calling thread's resource pool.

The following code defines and initializes an empty stack capable of holding up to ten word-sized data values.

```
#define MAX_ITEMS 10

stack_data_t my_stack_array[MAX_ITEMS];
struct k_stack my_stack;

k_stack_init(&my_stack, my_stack_array, MAX_ITEMS);
```

Alternatively, a stack can be defined and initialized at compile time by calling `K_STACK_DEFINE`.

The following code has the same effect as the code segment above. Observe that the macro defines both the stack and its array of data values.

```
K_STACK_DEFINE(my_stack, MAX_ITEMS);
```

Pushing to a Stack A data item is added to a stack by calling `k_stack_push()`.

The following code builds on the example above, and shows how a thread can create a pool of data structures by saving their memory addresses in a stack.

```
/* define array of data structures */
struct my_buffer_type {
    int field1;
    ...
    };
struct my_buffer_type my_buffers[MAX_ITEMS];

/* save address of each data structure in a stack */
for (int i = 0; i < MAX_ITEMS; i++) {
    k_stack_push(&my_stack, (stack_data_t)&my_buffers[i]);
}
```

Popping from a Stack A data item is taken from a stack by calling `k_stack_pop()`.

The following code builds on the example above, and shows how a thread can dynamically allocate an unused data structure. When the data structure is no longer required, the thread must push its address back on the stack to allow the data structure to be reused.

```
struct my_buffer_type *new_buffer;

k_stack_pop(&buffer_stack, (stack_data_t *)&new_buffer, K_FOREVER);
new_buffer->field1 = ...
```

Suggested Uses Use a stack to store and retrieve integer data values in a "last in, first out" manner, when the maximum number of stored items is known.

Configuration Options Related configuration options:

- None.

API Reference
Stack APIs

Related code samples

- Dining PhilosophersImplement a solution to the Dining Philosophers problem using Zephyr kernel services.

Message Queues

A *message queue* is a kernel object that implements a simple message queue, allowing threads and ISRs to asynchronously send and receive fixed-size data items.

- *Concepts*
- *Implementation*
  - *Defining a Message Queue*
  - *Writing to a Message Queue*
  - *Reading from a Message Queue*
  - *Peeking into a Message Queue*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of message queues can be defined (limited only by available RAM). Each message queue is referenced by its memory address.

A message queue has the following key properties:

- A **ring buffer** of data items that have been sent but not yet received.
- A **data item size**, measured in bytes.
- A **maximum quantity** of data items that can be queued in the ring buffer.

A message queue must be initialized before it can be used. This sets its ring buffer to empty.

A data item can be **sent** to a message queue by a thread or an ISR. The data item pointed at by the sending thread is copied to a waiting thread, if one exists; otherwise the item is copied to the message queue's ring buffer, if space is available. In either case, the size of the data area being sent *must* equal the message queue's data item size.

If a thread attempts to send a data item when the ring buffer is full, the sending thread may choose to wait for space to become available. Any number of sending threads may wait simultaneously when the ring buffer is full; when space becomes available it is given to the highest priority sending thread that has waited the longest.

A data item can be **received** from a message queue by a thread. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

If a thread attempts to receive a data item when the ring buffer is empty, the receiving thread may choose to wait for a data item to be sent. Any number of receiving threads may wait simultaneously when the ring buffer is empty; when a data item becomes available it is given to the highest priority receiving thread that has waited the longest.

A thread can also **peek** at the message on the head of a message queue without removing it from the queue. The data item is copied to the area specified by the receiving thread; the size of the receiving area *must* equal the message queue's data item size.

> **Note**
>
> The kernel does allow an ISR to receive an item from a message queue, however the ISR must not attempt to wait if the message queue is empty.

> **Note**
>
> Alignment of the message queue's ring buffer is not necessary. The underlying implementation uses `memcpy()` (which is alignment-agnostic) and does not expose any internal pointers.

Implementation

Defining a Message Queue A message queue is defined using a variable of type `k_msgq`. It must then be initialized by calling `k_msgq_init()`.

The following code defines and initializes an empty message queue that is capable of holding 10 items, each of which is 12 bytes long.

```
struct data_item_type {
    uint32_t field1;
    uint32_t field2;
    uint32_t field3;
};

char my_msgq_buffer[10 * sizeof(struct data_item_type)];
struct k_msgq my_msgq;

k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(struct data_item_type), 10);
```

Alternatively, a message queue can be defined and initialized at compile time by calling one of the following macros:

- `K_MSGQ_DEFINE` – Defines a public message queue.
- `K_MSGQ_DEFINE_STATIC` – Defines a private (static scope) message queue.
- `K_MSGQ_DEFINE_TYPE` – Defines a public message queue for given item type.
- `K_MSGQ_DEFINE_STATIC_TYPE` – Defines a private message queue for given item type.

The following code has the same effect as the code segment above. Observe that the macro defines both the message queue and its buffer.

```
K_MSGQ_DEFINE(my_msgq, sizeof(struct data_item_type), 10, 1);
```

The `K_MSGQ_DEFINE_TYPE` shorthand can be used if the queue item type is known at compile time. This macro configures the queue item size and alignment automatically:

```
K_MSGQ_DEFINE_TYPE(my_msgq, struct data_item_type, 10);
```

Writing to a Message Queue A data item is added to a message queue by calling `k_msgq_put()`.

The following code builds on the example above, and uses the message queue to pass data items from a producing thread to one or more consuming threads. If the message queue fills up because the consumers can't keep up, the producing thread throws away all existing data so the newer data can be saved. Note that this api will trigger reschedule.

```
void producer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* create data item to send (e.g. measurement, timestamp, ...) */
        data = ...

        /* send data to consumers */
        while (k_msgq_put(&my_msgq, &data, K_NO_WAIT) != 0) {
            /* message queue is full: purge old data & try again */
            k_msgq_purge(&my_msgq);
        }

        /* data item was successfully added to message queue */
    }
}
```

Reading from a Message Queue A data item is taken from a message queue by calling `k_msgq_get()`.

The following code builds on the example above, and uses the message queue to process data items generated by one or more producing threads. Note that the return value of `k_msgq_get()` should be tested as `-ENOMSG` can be returned due to `k_msgq_purge()`.

```
void consumer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* get a data item */
        k_msgq_get(&my_msgq, &data, K_FOREVER);

        /* process data item */
        ...
    }
}
```

Peeking into a Message Queue A data item is read from a message queue by calling `k_msgq_peek()`.

The following code peeks into the message queue to read the data item at the head of the queue that is generated by one or more producing threads.

```
void consumer_thread(void)
{
    struct data_item_type data;

    while (1) {
        /* read a data item by peeking into the queue */
        k_msgq_peek(&my_msgq, &data);
```

(continues on next page)

(continued from previous page)

```
        /* process data item */
        ...
    }
}
```

Suggested Uses Use a message queue to transfer small data items between threads in an asynchronous manner.

> **Note**
>
> A message queue can be used to transfer large data items, if desired. However, this can increase interrupt latency as interrupts are locked while a data item is written or read. The time to write or read a data item increases linearly with its size since the item is copied in its entirety to or from the buffer in memory. For this reason, it is usually preferable to transfer large data items by exchanging a pointer to the data item, rather than the data item itself.
>
> A synchronous transfer can be achieved by using the kernel's mailbox object type.

Configuration Options Related configuration options:

- None.

API Reference
Message Queue APIs

> **Related code samples**
>
> - Message QueueImplement a basic message queue producer/consumer thread pair.

Mailboxes

A *mailbox* is a kernel object that provides enhanced message queue capabilities that go beyond the capabilities of a message queue object. A mailbox allows threads to send and receive messages of any size synchronously or asynchronously.

- *Concepts*
  - *Message Format*
  - *Message Lifecycle*
  - *Thread Compatibility*
  - *Message Flow Control*
- *Implementation*
  - *Defining a Mailbox*
  - *Message Descriptors*
  - *Sending a Message*
  - *Receiving a Message*

- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of mailboxes can be defined (limited only by available RAM). Each mailbox is referenced by its memory address.

A mailbox has the following key properties:

- A **send queue** of messages that have been sent but not yet received.
- A **receive queue** of threads that are waiting to receive a message.

A mailbox must be initialized before it can be used. This sets both of its queues to empty.

A mailbox allows threads, but not ISRs, to exchange messages. A thread that sends a message is known as the **sending thread**, while a thread that receives the message is known as the **receiving thread**. Each message may be received by only one thread (i.e. point-to-multipoint and broadcast messaging is not supported).

Messages exchanged using a mailbox are handled non-anonymously, allowing both threads participating in an exchange to know (and even specify) the identity of the other thread.

Message Format A **message descriptor** is a data structure that specifies where a message's data is located, and how the message is to be handled by the mailbox. Both the sending thread and the receiving thread supply a message descriptor when accessing a mailbox. The mailbox uses the message descriptors to perform a message exchange between compatible sending and receiving threads. The mailbox also updates certain message descriptor fields during the exchange, allowing both threads to know what has occurred.

A mailbox message contains zero or more bytes of **message data**. The size and format of the message data is application-defined, and can vary from one message to the next.

A **message buffer** is an area of memory provided by the thread that sends or receives the message data. An array or structure variable can often be used for this purpose.

A message that has neither form of message data is called an **empty message**.

> **Note**
>
> A message whose message buffer exists, but contains zero bytes of actual data, is *not* an empty message.

Message Lifecycle The life cycle of a message is straightforward. A message is created when it is given to a mailbox by the sending thread. The message is then owned by the mailbox until it is given to a receiving thread. The receiving thread may retrieve the message data when it receives the message from the mailbox, or it may perform data retrieval during a second, subsequent mailbox operation. Only when data retrieval has occurred is the message deleted by the mailbox.

Thread Compatibility A sending thread can specify the address of the thread to which the message is sent, or send it to any thread by specifying `K_ANY`. Likewise, a receiving thread can specify the address of the thread from which it wishes to receive a message, or it can receive a message from any thread by specifying `K_ANY`. A message is exchanged only when the requirements of both the sending thread and receiving thread are satisfied; such threads are said to be **compatible**.

For example, if thread A sends a message to thread B (and only thread B) it will be received by thread B if thread B tries to receive a message from thread A or if thread B tries to receive from any thread. The exchange will not occur if thread B tries to receive a message from thread C. The message can never be received by thread C, even if it tries to receive a message from thread A (or from any thread).

Message Flow Control Mailbox messages can be exchanged **synchronously** or **asynchronously**. In a synchronous exchange, the sending thread blocks until the message has been fully processed by the receiving thread. In an asynchronous exchange, the sending thread does not wait until the message has been received by another thread before continuing; this allows the sending thread to do other work (such as gather data that will be used in the next message) *before* the message is given to a receiving thread and fully processed. The technique used for a given message exchange is determined by the sending thread.

The synchronous exchange technique provides an implicit form of flow control, preventing a sending thread from generating messages faster than they can be consumed by receiving threads. The asynchronous exchange technique provides an explicit form of flow control, which allows a sending thread to determine if a previously sent message still exists before sending a subsequent message.

Implementation

Defining a Mailbox A mailbox is defined using a variable of type `k_mbox`. It must then be initialized by calling `k_mbox_init()`.

The following code defines and initializes an empty mailbox.

```
struct k_mbox my_mailbox;

k_mbox_init(&my_mailbox);
```

Alternatively, a mailbox can be defined and initialized at compile time by calling `K_MBOX_DEFINE`.

The following code has the same effect as the code segment above.

```
K_MBOX_DEFINE(my_mailbox);
```

Message Descriptors A message descriptor is a structure of type `k_mbox_msg`. Only the fields listed below should be used; any other fields are for internal mailbox use only.

info
: A 32-bit value that is exchanged by the message sender and receiver, and whose meaning is defined by the application. This exchange is bi-directional, allowing the sender to pass a value to the receiver during any message exchange, and allowing the receiver to pass a value to the sender during a synchronous message exchange.

size
: The message data size, in bytes. Set it to zero when sending an empty message, or when sending a message buffer with no actual data. When receiving a message, set it to the maximum amount of data desired, or to zero if the message data is not wanted. The mailbox updates this field with the actual number of data bytes exchanged once the message is received.

tx_data
: A pointer to the sending thread's message buffer. Set it to `NULL` when sending an empty message. Leave this field uninitialized when receiving a message.

tx_target_thread
: The address of the desired receiving thread. Set it to `K_ANY` to allow any thread to receive

the message. Leave this field uninitialized when receiving a message. The mailbox updates this field with the actual receiver's address once the message is received.

rx_source_thread
The address of the desired sending thread. Set it to `K_ANY` to receive a message sent by any thread. Leave this field uninitialized when sending a message. The mailbox updates this field with the actual sender's address when the message is put into the mailbox.

Sending a Message A thread sends a message by first creating its message data, if any.

Next, the sending thread creates a message descriptor that characterizes the message to be sent, as described in the previous section.

Finally, the sending thread calls a mailbox send API to initiate the message exchange. The message is immediately given to a compatible receiving thread, if one is currently waiting. Otherwise, the message is added to the mailbox's send queue.

Any number of messages may exist simultaneously on a send queue. The messages in the send queue are sorted according to the priority of the sending thread. Messages of equal priority are sorted so that the oldest message can be received first.

For a synchronous send operation, the operation normally completes when a receiving thread has both received the message and retrieved the message data. If the message is not received before the waiting period specified by the sending thread is reached, the message is removed from the mailbox's send queue and the send operation fails. When a send operation completes successfully the sending thread can examine the message descriptor to determine which thread received the message, how much data was exchanged, and the application-defined info value supplied by the receiving thread.

> **Note**
>
> A synchronous send operation may block the sending thread indefinitely, even when the thread specifies a maximum waiting period. The waiting period only limits how long the mailbox waits before the message is received by another thread. Once a message is received there is *no* limit to the time the receiving thread may take to retrieve the message data and unblock the sending thread.

For an asynchronous send operation, the operation always completes immediately. This allows the sending thread to continue processing regardless of whether the message is given to a receiving thread immediately or added to the send queue. The sending thread may optionally specify a semaphore that the mailbox gives when the message is deleted by the mailbox, for example, when the message has been received and its data retrieved by a receiving thread. The use of a semaphore allows the sending thread to easily implement a flow control mechanism that ensures that the mailbox holds no more than an application-specified number of messages from a sending thread (or set of sending threads) at any point in time.

> **Note**
>
> A thread that sends a message asynchronously has no way to determine which thread received the message, how much data was exchanged, or the application-defined info value supplied by the receiving thread.

Sending an Empty Message This code uses a mailbox to synchronously pass 4 byte random values to any consuming thread that wants one. The message "info" field is large enough to carry the information being exchanged, so the data portion of the message isn't used.

```
void producer_thread(void)
{
    struct k_mbox_msg send_msg;

    while (1) {

        /* generate random value to send */
        uint32_t random_value = sys_rand32_get();

        /* prepare to send empty message */
        send_msg.info = random_value;
        send_msg.size = 0;
        send_msg.tx_data = NULL;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);
    }
}
```

Sending Data Using a Message Buffer This code uses a mailbox to synchronously pass variable-sized requests from a producing thread to any consuming thread that wants it. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void producer_thread(void)
{
    char buffer[100];
    int buffer_bytes_used;

    struct k_mbox_msg send_msg;

    while (1) {

        /* generate data to send */
        ...
        buffer_bytes_used = ... ;
        memcpy(buffer, source, buffer_bytes_used);

        /* prepare to send message */
        send_msg.info = buffer_bytes_used;
        send_msg.size = buffer_bytes_used;
        send_msg.tx_data = buffer;
        send_msg.tx_target_thread = K_ANY;

        /* send message and wait until a consumer receives it */
        k_mbox_put(&my_mailbox, &send_msg, K_FOREVER);

        /* info, size, and tx_target_thread fields have been updated */

        /* verify that message data was fully received */
        if (send_msg.size < buffer_bytes_used) {
            printf("some message data dropped during transfer!");
            printf("receiver only had room for %d bytes", send_msg.info);
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Receiving a Message A thread receives a message by first creating a message descriptor that characterizes the message it wants to receive. It then calls one of the mailbox receive APIs. The mailbox searches its send queue and takes the message from the first compatible thread it finds. If no compatible thread exists, the receiving thread may choose to wait for one. If no compatible thread appears before the waiting period specified by the receiving thread is reached, the receive operation fails. Once a receive operation completes successfully the receiving thread can examine the message descriptor to determine which thread sent the message, how much data was exchanged, and the application-defined info value supplied by the sending thread.

Any number of receiving threads may wait simultaneously on a mailboxes' receive queue. The threads are sorted according to their priority; threads of equal priority are sorted so that the one that started waiting first can receive a message first.

> **Note**
>
> Receiving threads do not always receive messages in a first in, first out (FIFO) order, due to the thread compatibility constraints specified by the message descriptors. For example, if thread A waits to receive a message only from thread X and then thread B waits to receive a message from thread Y, an incoming message from thread Y to any thread will be given to thread B and thread A will continue to wait.

The receiving thread controls both the quantity of data it retrieves from an incoming message and where the data ends up. The thread may choose to take all of the data in the message, to take only the initial part of the data, or to take no data at all. Similarly, the thread may choose to have the data copied into a message buffer of its choice.

The following sections outline various approaches a receiving thread may use when retrieving message data.

Retrieving Data at Receive Time The most straightforward way for a thread to retrieve message data is to specify a message buffer when the message is received. The thread indicates both the location of the message buffer (which must not be `NULL`) and its size.

The mailbox copies the message's data to the message buffer as part of the receive operation. If the message buffer is not big enough to contain all of the message's data, any uncopied data is lost. If the message is not big enough to fill all of the buffer with data, the unused portion of the message buffer is left unchanged. In all cases the mailbox updates the receiving thread's message descriptor to indicate how many data bytes were copied (if any).

The immediate data retrieval technique is best suited for small messages where the maximum size of a message is known in advance.

The following code uses a mailbox to process variable-sized requests from any producing thread, using the immediate data retrieval technique. The message "info" field is used to exchange information about the maximum size message buffer that each thread can handle.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[100];

    int i;
    int total;
```

(continues on next page)

(continued from previous page)

```
    while (1) {
        /* prepare to receive message */
        recv_msg.info = 100;
        recv_msg.size = 100;
        recv_msg.rx_source_thread = K_ANY;

        /* get a data item, waiting as long as needed */
        k_mbox_get(&my_mailbox, &recv_msg, buffer, K_FOREVER);

        /* info, size, and rx_source_thread fields have been updated */

        /* verify that message data was fully received */
        if (recv_msg.info != recv_msg.size) {
            printf("some message data dropped during transfer!");
            printf("sender tried to send %d bytes", recv_msg.info);
        }

        /* compute sum of all message bytes (from 0 to 100 of them) */
        total = 0;
        for (i = 0; i < recv_msg.size; i++) {
            total += buffer[i];
        }
    }
}
```

Retrieving Data Later Using a Message Buffer A receiving thread may choose to defer message data retrieval at the time the message is received, so that it can retrieve the data into a message buffer at a later time. The thread does this by specifying a message buffer location of `NULL` and a size indicating the maximum amount of data it is willing to retrieve later.

The mailbox does not copy any message data as part of the receive operation. However, the mailbox still updates the receiving thread's message descriptor to indicate how many data bytes are available for retrieval.

The receiving thread must then respond as follows:

- If the message descriptor size is zero, then either the sender's message contained no data or the receiving thread did not want to receive any data. The receiving thread does not need to take any further action, since the mailbox has already completed data retrieval and deleted the message.
- If the message descriptor size is non-zero and the receiving thread still wants to retrieve the data, the thread must call `k_mbox_data_get()` and supply a message buffer large enough to hold the data. The mailbox copies the data into the message buffer and deletes the message.
- If the message descriptor size is non-zero and the receiving thread does *not* want to retrieve the data, the thread must call `k_mbox_data_get()` and specify a message buffer of `NULL`. The mailbox deletes the message without copying the data.

The subsequent data retrieval technique is suitable for applications where immediate retrieval of message data is undesirable. For example, it can be used when memory limitations make it impractical for the receiving thread to always supply a message buffer capable of holding the largest possible incoming message.

The following code uses a mailbox's deferred data retrieval mechanism to get message data from a producing thread only if the message meets certain criteria, thereby eliminating unneeded data copying. The message "info" field supplied by the sender is used to classify the message.

```
void consumer_thread(void)
{
    struct k_mbox_msg recv_msg;
    char buffer[10000];

    while (1) {
        /* prepare to receive message */
        recv_msg.size = 10000;
        recv_msg.rx_source_thread = K_ANY;

        /* get message, but not its data */
        k_mbox_get(&my_mailbox, &recv_msg, NULL, K_FOREVER);

        /* get message data for only certain types of messages */
        if (is_message_type_ok(recv_msg.info)) {
            /* retrieve message data and delete the message */
            k_mbox_data_get(&recv_msg, buffer);

            /* process data in "buffer" */
            ...
        } else {
            /* ignore message data and delete the message */
            k_mbox_data_get(&recv_msg, NULL);
        }
    }
}
```

Suggested Uses Use a mailbox to transfer data items between threads whenever the capabilities of a message queue are insufficient.

Configuration Options Related configuration options:

- `CONFIG_NUM_MBOX_ASYNC_MSGS`

API Reference
Mailbox APIs

Pipes

A *pipe* is a kernel object that allows a thread to send a byte stream to another thread. Pipes enable efficient inter-thread communication and can be used to synchronously transfer chunks of data in whole or in part.

- *Concepts*
- *Implementation*
  - *Writing to a Pipe*
  - *Reading from a Pipe*
  - *Resetting a Pipe*
- *Suggested Uses*
- *API Reference*

Concepts Any number of pipes can be defined, limited only by available RAM. Each pipe is referenced by its memory address.

A pipe has the following key property:

- A **size** that indicates the capacity of the pipe's ring buffer. Note that a size of zero defines a pipe with no ring buffer.

A pipe must be initialized before it can be used. When initialized, the pipe is empty.

Threads interact with the pipe as follows:

- **Writing**: Data is synchronously written, either in whole or in part, to a pipe by a thread. Accepted data is either copied directly to the waiting reader(s) or to the pipe's ring buffer. If the ring buffer is full or simply absent, the operation blocks until sufficient space becomes available or the specified timeout expires.
- **Reading**: Data is synchronously read, either in whole or in part, from a pipe by a thread. Accepted data is either copied from the pipe's ring buffer or directly from the waiting sender(s). If the ring buffer is empty or simply absent, the operation blocks until data becomes available or the specified timeout expires.
- **Resetting**: A thread can reset a pipe, which resets its internal state and ends all pending read and write operations with an error code.

Pipes are well-suited for scenarios like producer-consumer patterns or streaming data between threads.

Implementation A pipe is defined using a variable of type `k_pipe` and a byte buffer. The pipe must then be initialized by calling `k_pipe_init()`.

The following code defines and initializes an empty pipe with a ring buffer capable of holding 100 bytes, aligned to a 4-byte boundary:

```
uint8_t __aligned(4) my_ring_buffer[100];
struct k_pipe my_pipe;

k_pipe_init(&my_pipe, my_ring_buffer, sizeof(my_ring_buffer));
```

Alternatively, a pipe can be defined and initialized at compile time using the `K_PIPE_DEFINE` macro, which defines both the pipe and its ring buffer:

```
K_PIPE_DEFINE(my_pipe, 100, 4);
```

This has the same effect as the code above.

When no ring buffer is used, the buffer pointer argument should be NULL and the size argument should be 0.

Writing to a Pipe Data is added to a pipe by calling `k_pipe_write()`.

The following example demonstrates using a pipe to send data from a producer thread to one or more consumer threads. If the pipe's ring buffer fills up, the producer thread waits for a specified amount of time.

```
struct message_header {
    size_t num_data_bytes; /* Example field */
    ...
};

void producer_thread(void)
{
    int rc;
```

(continues on next page)

(continued from previous page)

```
    uint8_t *data;
    size_t total_size;
    size_t bytes_written;

    while (1) {
        /* Craft message to send in the pipe */
        make_message(data, &total_size);
        bytes_written = 0;

        /* Write data to the pipe, handling partial writes */
        while (bytes_written < total_size) {
            rc = k_pipe_write(&my_pipe, &data[bytes_written], total_size -␣
↪bytes_written, K_NO_WAIT);

            if (rc < 0) {
                /* Error occurred */
                ...
                break;
            } else {
                /* Partial or full write succeeded; adjust for next iteration */
                bytes_written += rc;
            }
        }

        /* Reset bytes_written for the next message */
        bytes_written = 0;
        ...
    }
}
```

Reading from a Pipe Data is retrieved from the pipe by calling `k_pipe_read()`.

The following example builds on the producer thread example above. It shows a consumer thread that processes data generated by the producer.

```
struct message_header {
    size_t num_data_bytes; /* Example field */
    ...
};

void consumer_thread(void)
{
    int rc;
    uint8_t buffer[128];
    size_t bytes_read = 0;
    struct message_header *header = (struct message_header *)buffer;

    while (1) {
        /* Step 1: Read the message header */
        bytes_read = 0;
   read_header:
        while (bytes_read < sizeof(*header)) {
            rc = k_pipe_read(&my_pipe, &buffer[bytes_read], sizeof(*header) -␣
↪bytes_read, &bytes_read, K_NO_WAIT);

            if (rc < 0) {
```

(continues on next page)

(continued from previous page)

```
                /* Error occurred */
                ...
                goto read_header;
            }

            /* Adjust for partial reads */
            bytes_read += rc;
        }

        /* Step 2: Read the message body */
        bytes_read = 0;
        while (bytes_read < header->num_data_bytes) {
            rc = k_pipe_read(&my_pipe, &buffer[sizeof(*header) + bytes_read],␣
↪header->num_data_bytes - bytes_read, K_NO_WAIT);

            if (rc < 0) {
                /* Error occurred */
                ...
                goto read_header;
            }

            /* Adjust for partial reads */
            bytes_read += rc;
        }
        /* Successfully received the complete message */
    }
}
```

Resetting a Pipe The pipe can be reset by calling `k_pipe_reset()`. Resetting a pipe resets its internal state and ends all pending operations with an error code.

The following example demonstrates resetting a pipe in response to a critical error:

```
void monitor_thread(void)
{
    while (1) {
        ...
        /* Critical error detected: reset the entire pipe to reset it. */
        k_pipe_reset(&my_pipe);
        ...
    }
}
```

Suggested Uses Pipes are useful for sending streams of data between threads. Typical applications include:

- Implementing producer-consumer patterns.
- Streaming logs or packets between threads.
- Handling variable-length message passing in real-time systems.

API Reference

Pipe APIs

3.1.3 Memory Management

See *Memory Management*.

3.1.4 Timing

These pages cover timing related services.

Kernel Timing

Zephyr provides a robust and scalable timing framework to enable reporting and tracking of timed events from hardware timing sources of arbitrary precision.

Time Units Kernel time is tracked in several units which are used for different purposes.

Real time values, typically specified in milliseconds or microseconds, are the default presentation of time to application code. They have the advantages of being universally portable and pervasively understood, though they may not match the precision of the underlying hardware perfectly.

The kernel presents a "cycle" count via the `k_cycle_get_32()` and `k_cycle_get_64()` APIs. The intent is that this counter represents the fastest cycle counter that the operating system is able to present to the user (for example, a CPU cycle counter) and that the read operation is very fast. The expectation is that very sensitive application code might use this in a polling manner to achieve maximal precision. The frequency of this counter is available from `sys_clock_hw_cycles_per_sec()`. On most platforms this is a runtime constant that evaluates to `CONFIG_SYS_CLOCK_HW_CYCLES_PER_SEC` and is fixed for the lifetime of the system. On platforms where the system timer frequency is not fixed, `sys_clock_hw_cycles_per_sec()` returns a runtime value and application code must not assume a single immutable frequency.

Runtime System Timer Frequency Some platforms need the system timer frequency to be available at runtime, either because the timer driver discovers the clock rate from hardware or because the timer clock rate can change after boot.

Platforms that can change the active system timer frequency at runtime must enable `CONFIG_SYSTEM_CLOCK_HW_CYCLES_PER_SEC_RUNTIME_UPDATE` and:

- Call `z_sys_clock_hw_cycles_per_sec_update()` after applying the clock change.
- If the system timer driver caches derived constants (e.g. cycles-per-tick) or needs to reprogram hardware when the clock changes, provide a timer-driver override of `z_sys_clock_hw_cycles_per_sec_update()`.

The default implementation of `z_sys_clock_hw_cycles_per_sec_update()` only updates the stored frequency value.

> **Note**
>
> `CONFIG_SYSTEM_CLOCK_HW_CYCLES_PER_SEC_RUNTIME_UPDATE` tracks the system timer frequency as a single **global** value. It is not compatible with per-CPU frequency scaling configurations where different CPUs could observe different system timer frequencies.
>
> When enabled, `sys_clock_hw_cycles_per_sec()` and time unit conversions follow the current runtime value.

For asynchronous timekeeping, the kernel defines a "ticks" concept. A "tick" is the internal count in which the kernel does all its internal uptime and timeout bookkeeping. Interrupts are expected to be delivered on tick boundaries to the extent practical, and no fractional ticks are tracked. The choice of tick rate is configurable via `CONFIG_SYS_CLOCK_TICKS_PER_SEC`. Defaults

on most hardware platforms (ones that support setting arbitrary interrupt timeouts) are expected to be in the range of 10 kHz, with software emulation platforms and legacy drivers using a more traditional 100 Hz value.

Conversion Zephyr provides an extensively enumerated conversion library with rounding control for all time units. Any unit of "ms" (milliseconds), "us" (microseconds), "tick", or "cyc" can be converted to any other. Control of rounding is provided, and each conversion is available in "floor" (round down to nearest output unit), "ceil" (round up) and "near" (round to nearest). Finally the output precision can be specified as either 32 or 64 bits.

For example: `k_ms_to_ticks_ceil32()` will convert a millisecond input value to the next higher number of ticks, returning a result truncated to 32 bits of precision; and `k_cyc_to_us_floor64()` will convert a measured cycle count to an elapsed number of microseconds in a full 64 bits of precision. See the reference documentation for the full enumeration of conversion routines.

On most platforms, where the various counter rates are integral multiples of each other and where the output fits within a single word, these conversions expand to a 2-4 operation sequence, requiring full precision only where actually required and requested.

Uptime The kernel tracks a system uptime count on behalf of the application. This is available at all times via `k_uptime_get()`, which provides an uptime value in milliseconds since system boot. This is expected to be the utility used by most portable application code.

The internal tracking, however, is as a 64 bit integer count of ticks. Apps with precise timing requirements (that are willing to do their own conversions to portable real time units) may access this with `k_uptime_ticks()`.

`k_uptime_delta()` can be used to get the time elapsed between a reference and the current time. The referenced time will be updated to the current uptime, to easily calculate the next elapsed time.

Timeouts The Zephyr kernel provides many APIs with a "timeout" parameter. Conceptually, this indicates the time at which an event will occur. For example:

- Kernel blocking operations like `k_sem_take()` or `k_queue_get()` may provide a timeout after which the routine will return with an error code if no data is available.
- Kernel `k_timer` objects must specify delays for their duration and period.
- The kernel `k_work_delayable` API provides a timeout parameter indicating when a work queue item will be added to the system queue.

All these values are specified using a `k_timeout_t` value. This is an opaque struct type that must be initialized using one of a family of kernel timeout macros. The most common, `K_MSEC`, defines a time in milliseconds after the current time.

What is meant by "current time" for relative timeouts depends on the context:

- When scheduling a relative timeout from within a timeout callback (e.g. from within the expiry function passed to `k_timer_init()` or the work handler passed to `k_work_init_delayable()`), "current time" is the exact time at which the currently firing timeout was originally scheduled even if the "real time" will already have advanced. This is to ensure that timers scheduled from within another timer's callback will always be calculated with a precise offset to the firing timer. It is thereby possible to fire at regular intervals without introducing systematic clock drift over time.
- When scheduling a timeout from application context, "current time" means the value returned by `k_uptime_ticks()` at the time at which the kernel receives the timeout value.

Other options for timeout initialization follow the unit conventions described above: `K_NSEC()`, `K_USEC`, `K_TICKS` and `K_CYC()` specify timeout values that will expire after specified numbers of nanoseconds, microseconds, ticks and cycles, respectively.

Precision of `k_timeout_t` values is configurable, with the default being 32 bits. Large uptime counts in non-tick units will experience complicated rollover semantics, so it is expected that timing-sensitive applications with long uptimes will be configured to use a 64 bit timeout type.

Finally, it is possible to specify timeouts as absolute times since system boot. A timeout initialized with `K_TIMEOUT_ABS_MS` indicates a timeout that will expire after the system uptime reaches the specified value. There are likewise nanosecond, microsecond, cycles and ticks variants of this API.

Timing Internals

Timeout Queue All Zephyr `k_timeout_t` events specified using the API above are managed in a single, global queue of events. The action to take on an event is specified as a callback function pointer provided by the subsystem requesting the event, along with a `_timeout` tracking struct that is expected to be embedded within subsystem-defined data structures (for example: a `wait_q` struct, or a `k_tid_t` thread struct).

Note that all variant units passed via a `k_timeout_t` are converted to ticks once on insertion into the queue. There are no multiple-conversion steps internal to the kernel, so precision is guaranteed at the tick level no matter how many events exist or how long a timeout might be.

The data structure that holds the queue is selected at build time through the `CONFIG_TIMEOUT_BACKEND` choice. Only the front end is shared between backends (the announce path, the SMP re-entry handling, and the relative versus absolute timeout rules); each backend supplies the queue itself, so an integrator can match the data structure to the workload without touching the common code.

The default, `CONFIG_TIMEOUT_BACKEND_DLIST`, stores events in a doubly linked list sorted by expiry, each holding a delta count in ticks from its predecessor. Insertion is O(N) in the number of pending timeouts: inexpensive for the handful a typical system has pending, but it scales poorly when many are outstanding. The four alternative backends, all currently experimental, trade extra memory or behaviour for faster insertion at scale:

- `CONFIG_TIMEOUT_BACKEND_MINHEAP` keeps the events in a binary min-heap keyed on absolute expiry, making insertion and removal O(log N). It requires 64-bit ticks (`CONFIG_TIMEOUT_64BIT`) and a fixed-capacity heap (`CONFIG_TIMEOUT_HEAP_MAX_ENTRIES`, whose overflow is fatal), and it does not preserve the firing order of timeouts that expire on the same tick.
- `CONFIG_TIMEOUT_BACKEND_WHEEL` is a hierarchical timer wheel with O(1) insertion and removal for the near future and a sorted overflow list beyond. It has the largest per-event and static footprint, does not preserve same-tick firing order, and wakes a tickless-idle CPU periodically because its next-timeout estimate is bounded by the wheel period (a power cost the other backends avoid).
- `CONFIG_TIMEOUT_BACKEND_BUCKET` is a single-level bucketed delta list, a simpler relative of the wheel. It gives O(1) insertion within a tunable near-future window (`CONFIG_TIMEOUT_BUCKET_LISTS`) and falls back to a sorted overflow list beyond it. It also requires 64-bit ticks, but unlike the wheel it preserves same-tick firing order and adds no idle-wakeup cost.
- `CONFIG_TIMEOUT_BACKEND_SKIPLIST` is a Pugh skip list keyed on absolute expiry. Expected insertion and removal are O(log N) with no capacity limit, and same-tick firing order is FIFO. It requires 64-bit ticks. Each event stores `CONFIG_TIMEOUT_SKIPLIST_MAX_LEVEL` forward pointers, so per-event RAM is higher than the delta list or min-heap.

The non-default backends target systems that hold many concurrent timeouts, especially ones clustered in the near future. For most applications the delta list remains the appropriate default.

Timer Drivers Kernel timing at the tick level is driven by a timer driver. That interface and the locking it runs under are described in *System Timer Drivers*.

Time Slicing An auxiliary job of the timing subsystem is to provide tick counters to the scheduler that allow implementation of time slicing of threads. A thread time-slice cannot be a timeout value, as it does not reflect a global expiration but instead a per-CPU value that needs to be tracked independently on each CPU in an SMP context.

Because there may be no other hardware available to drive timeslicing, Zephyr multiplexes the existing timer driver. This means that the value passed to `sys_clock_set_timeout()` may be clamped to a smaller value than the current next timeout when a time sliced thread is currently scheduled.

Subsystems that keep millisecond APIs In general, code like this will port just like applications code will. Millisecond values from the user may be treated any way the subsystem likes, and then converted into kernel timeouts using `K_MSEC()` at the point where they are presented to the kernel.

Obviously this comes at the cost of not being able to use new features, like the higher precision timeout constructors or absolute timeouts. But for many subsystems with simple needs, this may be acceptable.

One complexity is `K_FOREVER`. Subsystems that might have been able to accept this value to their millisecond API in the past no longer can, because it is no longer an integral type. Such code will need to use a different, integer-valued token to represent "forever". `K_NO_WAIT` has the same typesafety concern too, of course, but as it is (and has always been) simply a numerical zero, it has a natural porting path.

Subsystems using `k_timeout_t` Ideally, code that takes a "timeout" parameter specifying a time to wait should be using the kernel native abstraction where possible. But `k_timeout_t` is opaque, and needs to be converted before it can be inspected by an application.

Some conversions are simple. Code that needs to test for `K_FOREVER` can simply use the `K_TIMEOUT_EQ()` macro to test the opaque struct for equality and take special action.

The more complicated case is when the subsystem needs to take a timeout and loop, waiting for it to finish while doing some processing that may require multiple blocking operations on underlying kernel code. For example, consider this design:

```
void my_wait_for_event(struct my_subsys *obj, int32_t timeout_in_ms)
{
    while (true) {
        uint32_t start = k_uptime_get_32();

        if (is_event_complete(obj)) {
            return;
        }

        /* Wait for notification of state change */
        k_sem_take(obj->sem, timeout_in_ms);

        /* Subtract elapsed time */
        timeout_in_ms -= (k_uptime_get_32() - start);
    }
}
```

This code requires that the timeout value be inspected, which is no longer possible. For situations like this, the new API provides the internal `sys_timepoint_calc()` and `sys_timepoint_timeout()` routines that converts an arbitrary timeout to and from a timepoint value based on an uptime tick at which it will expire. So such a loop might look like:

```
void my_wait_for_event(struct my_subsys *obj, k_timeout_t timeout)
{
    /* Compute the end time from the timeout */
    k_timepoint_t end = sys_timepoint_calc(timeout);

    do {
        if (is_event_complete(obj)) {
            return;
        }

        /* Update timeout with remaining time */
        timeout = sys_timepoint_timeout(end);

        /* Wait for notification of state change */
        k_sem_take(obj->sem, timeout);
    } while (!K_TIMEOUT_EQ(timeout, K_NO_WAIT));
}
```

Note that `sys_timepoint_calc()` accepts special values `K_FOREVER` and `K_NO_WAIT`, and works identically for absolute timeouts as well as conventional ones. Conversely, `sys_timepoint_timeout()` may return `K_FOREVER` or `K_NO_WAIT` if those were used to create the timepoint, the later also being returned if the timepoint is now in the past. For simple cases, `sys_timepoint_expired()` can be used as well.

But some care is still required for subsystems that use those. Note that delta timeouts need to be interpreted relative to a "current time", and obviously that time is the time of the call to `sys_timepoint_calc()`. But the user expects that the time is the time they passed the timeout to you. Care must be taken to call this function just once, as synchronously as possible to the timeout creation in user code. It should not be used on a "stored" timeout value, and should never be called iteratively in a loop.

API Reference

System Clock APIs

Timers

A *timer* is a kernel object that measures the passage of time using the kernel's system clock. When a timer's specified time limit is reached it can perform an application-defined action, or it can simply record the expiration and wait for the application to read its status.

- *Concepts*
- *Timer Observers*
- *Implementation*
  - *Defining a Timer*
  - *Using a Timer Expiry Function*
  - *Reading Timer Status*
  - *Using Timer Status Synchronization*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of timers can be defined (limited only by available RAM). Each timer is referenced by its memory address.

A timer has the following key properties:

- A **duration** specifying the time interval before the timer expires for the first time. This is a `k_timeout_t` value that may be initialized via different units.
- A **period** specifying the time interval between all timer expirations after the first one, also a `k_timeout_t`. It must be non-negative. A period of `K_NO_WAIT` (i.e. zero) or `K_FOREVER` means that the timer is a one-shot timer that stops after a single expiration. (For example then, if a timer is started with a duration of 200 and a period of 75, it will first expire after 200 ms and then every 75 ms after that.)
- An **expiry function** that is executed each time the timer expires. The function is executed by the system clock interrupt handler. If no expiry function is required a `NULL` function can be specified.
- A **stop function** that is executed if the timer is stopped prematurely while running. The function is executed by the thread that stops the timer. If no stop function is required a `NULL` function can be specified.
- A **status** value that indicates how many times the timer has expired since the status value was last read.

A timer must be initialized before it can be used. This specifies its expiry function and stop function values, sets the timer's status to zero, and puts the timer into the **stopped** state.

A timer is **started** by specifying a duration and a period. The timer's status is reset to zero, and then the timer enters the **running** state and begins counting down towards expiry.

> **Note**
>
> The timer's duration is a **minimum** delay relative to the time the timer was started. The timer's period is a **minimum** delay relative to the last time the timer "should" have expired. This means that a periodic timer will not drift relative to the system timer, and that its periodic delay can be shorter or longer than the specified period.
>
> To ensure a minimum delay until the timer expires, restart the timer from within the **expiry function** or after the timer expired.
>
> The variability of the delays stem from system variables like interrupt handling delays and execution time of preceding timer handlers.

When a running timer expires its status is incremented and the timer executes its expiry function, if one exists; If a thread is waiting on the timer, it is unblocked. If the timer's period is zero the timer enters the stopped state; otherwise, the timer restarts with a new duration equal to the delta between the time the timer "should" have expired and its period.

A running timer can be stopped in mid-countdown, if desired. The timer's status is left unchanged, then the timer enters the stopped state and executes its stop function, if one exists. If a thread is waiting on the timer, it is unblocked. Attempting to stop a non-running timer is permitted, but has no effect on the timer since it is already stopped.

A running timer can be restarted in mid-countdown, if desired. The timer's status is reset to zero, then the timer begins counting down using the new duration and period values specified by the caller. If a thread is waiting on the timer, it continues waiting.

A timer's status can be read directly at any time to determine how many times the timer has expired since its status was last read. Reading a timer's status resets its value to zero. The amount of time remaining before the timer expires can also be read; a value of zero indicates that the timer is stopped.

A thread may read a timer's status indirectly by **synchronizing** with the timer. This blocks the thread until the timer's status is non-zero (indicating that it has expired at least once) or the timer is stopped; if the timer status is already non-zero or the timer is already stopped the thread continues without waiting. The synchronization operation returns the timer's status and resets it to zero.

> **Note**
>
> Only a single user should examine the status of any given timer, since reading the status (directly or indirectly) changes its value. Similarly, only a single thread at a time should synchronize with a given timer. ISRs are not permitted to synchronize with timers, since ISRs are not allowed to block.

Timer Observers When `CONFIG_TIMER_OBSERVER` is enabled, code can register *timer observers* that are notified of timer lifecycle events across all timers in the system. An observer is a set of optional callbacks invoked when a timer is initialized, started, stopped, or expires. This allows external modules – for example tracing, profiling, or power-management code – to react to timer activity without modifying kernel internals or the individual timers.

An observer is defined statically with `K_TIMER_OBSERVER_DEFINE`, which takes pointers to the `on_init`, `on_start`, `on_stop`, and `on_expiry` callbacks; any callback that is not needed may be passed as `NULL`. Because the expiry callback runs in interrupt context, it must be kept short and non-blocking.

Implementation

Defining a Timer A timer is defined using a variable of type `k_timer`. It must then be initialized by calling `k_timer_init()`.

The following code defines and initializes a timer.

```
struct k_timer my_timer;
extern void my_expiry_function(struct k_timer *timer_id);

k_timer_init(&my_timer, my_expiry_function, NULL);
```

Alternatively, a timer can be defined and initialized at compile time by calling `K_TIMER_DEFINE`.

The following code has the same effect as the code segment above.

```
K_TIMER_DEFINE(my_timer, my_expiry_function, NULL);
```

Using a Timer Expiry Function The following code uses a timer to perform a non-trivial action on a periodic basis. Since the required work cannot be done at the interrupt level, the timer's expiry function submits a work item to the *system workqueue*, whose thread performs the work.

```
void my_work_handler(struct k_work *work)
{
    /* do the processing that needs to be done periodically */
    ...
}

K_WORK_DEFINE(my_work, my_work_handler);

void my_timer_handler(struct k_timer *dummy)
{
```

(continues on next page)

(continued from previous page)

```
    k_work_submit(&my_work);
}

K_TIMER_DEFINE(my_timer, my_timer_handler, NULL);

...

/* start a periodic timer that expires once every second */
k_timer_start(&my_timer, K_SECONDS(1), K_SECONDS(1));
```

Reading Timer Status The following code reads a timer's status directly to determine if the timer has expired or not.

```
K_TIMER_DEFINE(my_status_timer, NULL, NULL);

...

/* start a one-shot timer that expires after 200 ms */
k_timer_start(&my_status_timer, K_MSEC(200), K_NO_WAIT);

/* do work */
...

/* check timer status */
if (k_timer_status_get(&my_status_timer) > 0) {
    /* timer has expired */
} else if (k_timer_remaining_get(&my_status_timer) == 0) {
    /* timer was stopped (by someone else) before expiring */
} else {
    /* timer is still running */
}
```

Using Timer Status Synchronization The following code performs timer status synchronization to allow a thread to do useful work while ensuring that a pair of protocol operations are separated by the specified time interval.

```
K_TIMER_DEFINE(my_sync_timer, NULL, NULL);

...

/* do first protocol operation */
...

/* start a one-shot timer that expires after 500 ms */
k_timer_start(&my_sync_timer, K_MSEC(500), K_NO_WAIT);

/* do other work */
...

/* ensure timer has expired (waiting for expiry, if necessary) */
k_timer_status_sync(&my_sync_timer);

/* do second protocol operation */
...
```

> **Note**
>
> If the thread had no other work to do it could simply sleep between the two protocol operations, without using a timer.

Suggested Uses Use a timer to initiate an asynchronous operation after a specified amount of time.

Use a timer to determine whether or not a specified amount of time has elapsed. In particular, timers should be used when higher precision and/or unit control is required than that afforded by the simpler `k_sleep()` and `k_usleep()` calls.

Use a timer to perform other work while carrying out operations involving time limits.

> **Note**
>
> If a thread needs to measure the time required to perform an operation it can read the *system clock or the hardware clock* directly, rather than using a timer.

Configuration Options Related configuration options:

- None

API Reference
Timer APIs

System Timer Drivers

This page describes the interface between the kernel's tick accounting and the driver for the hardware that produces those ticks. The kernel side of that accounting, and everything an application sees, is described in *Kernel Timing*.

Timer Drivers Kernel timing at the tick level is driven by a timer driver with a comparatively simple API.

- The driver is expected to be able to "announce" new ticks to the kernel via the `sys_clock_announce()` call, which passes an integer number of ticks that have elapsed since the last announce call (or system boot). These calls can occur at any time, but the driver is expected to attempt to ensure (to the extent practical given interrupt latency interactions) that they occur near tick boundaries (i.e. not "halfway through" a tick), and most importantly that they be correct over time and subject to minimal skew vs. other counters and real world time.
- The driver is expected to provide a `sys_clock_set_timeout()` call to the kernel which indicates how many ticks may elapse before the kernel must receive an announce call to trigger registered timeouts. It is legal to announce new ticks before that moment (though they must be correct) but delay after that will cause events to be missed. Note that the timeout value passed here is in a delta from current time, but that does not absolve the driver of the requirement to provide ticks at a steady rate over time. Naive implementations of this function are subject to bugs where the fractional tick gets "reset" incorrectly and causes clock skew.
- The driver is expected to provide a `sys_clock_elapsed()` call which provides a current indication of how many ticks have elapsed (as compared to a real world clock) since the last call to `sys_clock_announce()`, which the kernel needs to test newly arriving timeouts for expiration.

- The driver may optionally provide a `sys_clock_no_timeout()` call. The kernel invokes it in place of `sys_clock_set_timeout()` when no timeout is pending, that is when the timeout queue is empty, and `CONFIG_SYSTEM_CLOCK_SLOPPY_IDLE` is enabled.

  No tick announcement is forthcoming and the system does not care about precise uptime keeping, so the driver may do something to save resources. Normal operation resumes at the next `sys_clock_set_timeout()`; unlike `sys_clock_disable()`, this is not a teardown.

  The timer *counter* must not be stopped. `sys_clock_cycle_get_32()` and `sys_clock_cycle_get_64()` must keep counting up as if the call had never happened.

  > **Note**
  >
  > The CPU keeps running with an empty timeout queue, for instance because the ready queue still holds threads, or because an interrupt fires. Those threads and ISRs may call `k_cycle_get_32()` or `k_busy_wait()`. That limits what an implementation may do: masking the timer interrupt is safe, gating the timer's clock most likely is not. Exactly what is safe is hardware specific.

  The hook is optional. Without it, `sys_clock_set_timeout()` is asked for the longest wait it can express, `UINT32_MAX` ticks. That is numerically what `K_TICKS_FOREVER` was here, the long-standing "no deadline" signal, so a driver that has not been converted keeps working.

- The driver may optionally provide a `sys_clock_idle_enter()` call, which the power-management path uses in place of `sys_clock_set_timeout()` when the CPU is about to enter low-power idle. It receives the number of ticks until the next expected wakeup, or `SYS_CLOCK_IDLE_FOREVER` when there is none and the uptime accounting may drift.

  A driver that hands off to a low-power wakeup timer, or otherwise reconfigures itself for sleep, does so here. Told `SYS_CLOCK_IDLE_FOREVER` it may also stop its time base, which `sys_clock_no_timeout()` does not allow, because the CPU is on its way out and `sys_clock_idle_exit()` is guaranteed to run on the way back in. Only the calling CPU goes idle, so a driver whose time base is shared between CPUs must ensure only the last CPU going idle stops the clock.

  The hook is optional. Without it, `sys_clock_set_timeout()` is called with its deprecated `idle` argument set to `true`, so a driver still keying its low-power handling on that argument keeps working.

The last three entry points divide the four states a system can be in:

	nothing pending	something pending
running	`sys_clock_no_timeout()`	`sys_clock_set_timeout(ticks)`
idle	`sys_clock_idle_enter(SYS_CLOCK_IDLE_I`	`sys_clock_idle_enter(ticks)`

Without `CONFIG_SYSTEM_CLOCK_SLOPPY_IDLE` the left column never arises: the kernel keeps a synthetic deadline armed so the uptime stays exact, and a driver only ever sees the right one.

Timer Driver Locking The kernel exposes a unified timer lock via `sys_clock_lock()` and `sys_clock_unlock()`. This lock protects both the kernel's internal tick accounting (`curr_tick`, the timeout queue) and any driver-private state that must be consistent with it (e.g. a hardware cycle counter baseline).

Timer drivers that maintain internal state should acquire this lock at the start of their ISR, update their hardware state, then pass the lock key to `sys_clock_announce_locked()` which consumes it. This ensures that the driver's cycle counter baseline and the kernel's `curr_tick` are always updated under the same lock, eliminating race conditions that can arise on SMP systems (or, less commonly, on UP systems where higher-priority ISRs need consistent realtime references) when two separate locks are used.

The driver-provided callbacks `sys_clock_set_timeout()` and `sys_clock_elapsed()` are always invoked by the kernel with this lock already held.

For backward compatibility, `sys_clock_announce()` remains available and acquires the lock internally. New and migrated drivers should prefer the `sys_clock_lock()` / `sys_clock_announce_locked()` pattern.

Note that a natural implementation of this API results in a "tickless" kernel, which receives and processes timer interrupts only for registered events, relying on programmable hardware counters to provide irregular interrupts. But a traditional, "ticked" or "dumb" counter driver can be trivially implemented also:

- The driver can receive interrupts at a regular rate corresponding to the OS tick rate, calling `sys_clock_announce()` with an argument of one each time.
- The driver can ignore calls to `sys_clock_set_timeout()`, as every tick will be announced regardless of timeout status.
- The driver can return zero for every call to `sys_clock_elapsed()` as no more than one tick can be detected as having elapsed (because otherwise an interrupt would have been received).

SMP Details In general, the timer API described above does not change when run in a multiprocessor context. The kernel will internally synchronize all access appropriately, and ensure that all critical sections are small and minimal. But some notes are important to detail:

- Zephyr is agnostic about which CPU services timer interrupts. It is not illegal (though probably undesirable in some circumstances) to have every timer interrupt handled on a single processor. Existing SMP architectures implement symmetric timer drivers.
- The `sys_clock_announce()` call is expected to be globally synchronized at the driver level. The kernel does not do any per-CPU tracking, and expects that if two timer interrupts fire near simultaneously, that only one will provide the current tick count to the timing subsystem. The other may legally provide a tick count of zero if no ticks have elapsed. It should not "skip" the announce call because of timeslicing requirements (see the time slicing notes in *Kernel Timing*).
- Some SMP hardware uses a single, global timer device, others use a per-CPU counter. The complexity here (for example: ensuring counter synchronization between CPUs) is expected to be managed by the driver, not the kernel.
- The next timeout value passed back to the driver via `sys_clock_set_timeout()` is done identically for every CPU. So by default, every CPU will see simultaneous timer interrupts for every event, even though by definition only one of them should see a non-zero ticks argument to `sys_clock_announce()`. This is probably a correct default for timing sensitive applications (because it minimizes the chance that an errant ISR or interrupt lock will delay a timeout), but may be a performance problem in some cases. The current design expects that any such optimization is the responsibility of the timer driver.

Generic Tickless Core The work described above, the cycle-to-tick conversion, the announce baseline, the tick-aligned deadline computation and the counter wrap and range handling, is nearly the same in every tickless driver, and the hand-rolled variations are a recurring source of timer bugs. drivers/timer/system_timer_generic.h carries that logic once.

It is an implementation header, not a declaration header: including it *defines* `sys_clock_set_timeout()`, `sys_clock_elapsed()` and `sys_clock_cycle_get_32()` / `sys_clock_cycle_get_64()`. Any number of drivers can be built on it; each includes it once, after providing the macros and primitives below. A build compiles a single system timer driver, so those definitions land once. The driver then works in cycles only; the core owns the tick domain.

The core emits both cycle getters, the 64-bit one whatever the counter's width: the announce baseline is 64-bit, and the linker drops the getter where nothing calls it.

Whether a driver also selects `CONFIG_TIMER_HAS_64BIT_CYCLE_COUNTER` stays its own call. That option says a 64-bit read is cheap, or at least no dearer than a 32-bit one, so nothing is gained by preferring the narrower getter. With `TIMER_CORE_COUNTER_NONATOMIC` both go through the clock lock and that holds; with an atomic counter narrower than 64 bits only the 64-bit getter takes the lock, and the 32-bit one stays a lock-free read.

Two primitives are required: `timer_driver_cycle_get()`, which reads the hardware cycle counter, and one arming function, `timer_driver_set_compare()` or `timer_driver_set_reload()` depending on the backend. The driver's ISR acknowledges the hardware then calls `timer_core_announce()`, its init connects the IRQ then calls `timer_core_init()`, and on SMP its `smp_timer_init()` calls `timer_core_smp_prime()`.

Backend selection Exactly one of these states what the hardware is:

`TIMER_CORE_BACKEND_COMPARE_ORDERED`
: Absolute comparator, ordered match: the interrupt fires once the counter has reached or passed the programmed value, so a deadline already in the past fires at once. The usable range is half the counter width.

`TIMER_CORE_BACKEND_COMPARE_EXACT`
: Absolute comparator, equality match: a value written after the counter has passed it is missed for a whole counter period. The core writes the comparator through a verify loop, so the driver needs no minimum-delay floor of its own.

`TIMER_CORE_BACKEND_RELOAD`
: Relative delay: down-counters and compare-match-reset periodics. Assumed to auto-reload, so a ticked kernel free-runs from the value set at init.

Optional macros Each is defined only when the default, given below, does not fit:

`TIMER_CORE_CYCLES_PER_SEC`
: Counter rate, in Hz. Defaults to the kernel system clock rate. Set it when the counter is prescaled or runs at a fixed rate of its own. The core derives `TIMER_CORE_CYC_PER_TICK` from it, which a driver may read for its own hardware setup, typically the one-tick period it programs at init, but never defines itself.

`TIMER_CORE_CYCLES_PER_SEC_RUNTIME`
: The rate above is a variable, not a build-time constant, because it is read from the clock controller or computed at init. The core then precomputes the cycles per tick once instead of relying on the division folding. The variable must hold its final value before `timer_core_init()` is called.

`TIMER_CORE_COUNTER_WIDTH`
: Width, in bits, up to 64, of the count `timer_driver_cycle_get()` returns. Defaults to the native register width. A counter of another width states its own, and must: a genuine 64-bit counter on a 32-bit CPU would otherwise inherit a 32-bit mask and lose any span past 2^32 cycles. The core masks every delta to this width, so a narrow counter is read raw and the driver never has to widen the count in software.

`TIMER_CORE_COUNTER_NONMONOTONIC`
: The counter may momentarily read behind a value already observed, as a global timer does under QEMU SMP. The core treats a backwards read as no elapse rather than a huge jump.

`TIMER_CORE_COUNTER_NONATOMIC`
: `timer_driver_cycle_get()` is not a single atomic read but a value synthesized from state the ISR also touches. The core then reads it under the clock lock.

`TIMER_CORE_HAVE_CYCLE_GET_32`, `TIMER_CORE_HAVE_CYCLE_GET_64`
: The driver defines that entry point itself and the core does not. For a counter that needs scaling. `timer_core_cycle_get()`, available after the include, gives the full-width count in the counter's own domain to scale from.

The 32-bit one is required of a driver that sets `TIMER_CORE_CYCLES_PER_SEC`, a counter running at a rate of its own being one the kernel cannot read raw. The core then emits no 64-bit getter either, that being in the domain the driver just declared foreign. Supply one as well to have it.

`TIMER_CORE_ALARM_MAX_CYCLES`
Largest value the arming primitive can express, nothing more. Defaults to the whole span of `TIMER_CORE_COUNTER_WIDTH`, the alarm and the counter usually being one piece of hardware. Set it where the arming range is decided by something else: a compare or reload register narrower than the counter, or a separate device.

It states hardware capacity, so it carries no safety margin. The core derives its own from `TIMER_CORE_COUNTER_WIDTH`: it arms no further ahead of the last announce than half the counter's span, a quarter with `TIMER_CORE_COUNTER_NONMONOTONIC`, so a late announce still yields a delta the masking can resolve. Whichever of the two binds first wins.

`TIMER_CORE_ALARM_MIN_CYCLES`
Reload floor, in cycles, `TIMER_CORE_BACKEND_RELOAD` only. Defaults to 1.

`TIMER_CORE_ALARM_LEAD_CYCLES`
Cycles a compare must be ahead of the counter for the match to be caught, `TIMER_CORE_BACKEND_COMPARE_EXACT` only. Defaults to 1, a comparator written while the counter is still below it firing. Raise it for hardware that has to carry the write into the counter's clock domain first and misses a match programmed closer than that.

New tickless drivers should build on this header rather than reimplement the tick accounting. The header documents each macro and primitive in full.

3.1.5 Other

These pages cover other kernel services.

Atomic Services

An *atomic variable* is one that can be read and modified by threads and ISRs in an uninterruptible manner. It is a 32-bit variable on 32-bit machines and a 64-bit variable on 64-bit machines.

- *Concepts*
- *Implementation*
  - *Defining an Atomic Variable*
  - *Manipulating an Atomic Variable*
  - *Manipulating an Array of Atomic Variables*
  - *Memory Ordering*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts Any number of atomic variables can be defined (limited only by available RAM).

Using the kernel's atomic APIs to manipulate an atomic variable guarantees that the desired operation occurs correctly, even if higher priority contexts also manipulate the same variable.

The kernel also supports the atomic manipulation of a single bit in an array of atomic variables.

Implementation

Defining an Atomic Variable An atomic variable is defined using a variable of type `atomic_t`.

By default an atomic variable is initialized to zero. However, it can be given a different value using `ATOMIC_INIT`:

```
atomic_t flags = ATOMIC_INIT(0xFF);
```

Manipulating an Atomic Variable An atomic variable is manipulated using the APIs listed at the end of this section.

The following code shows how an atomic variable can be used to keep track of the number of times a function has been invoked. Since the count is incremented atomically, there is no risk that it will become corrupted in mid-increment if a thread calling the function is interrupted if by a higher priority context that also calls the routine.

```
atomic_t call_count;

int call_counting_routine(void)
{
    /* increment invocation counter */
    atomic_inc(&call_count);

    /* do rest of routine's processing */
    ...
}
```

Manipulating an Array of Atomic Variables An array of 32-bit atomic variables can be defined in the conventional manner. However, you can also define an N-bit array of atomic variables using `ATOMIC_DEFINE`.

A single bit in array of atomic variables can be manipulated using the APIs listed at the end of this section that end with `_bit()`.

The following code shows how a set of 200 flag bits can be implemented using an array of atomic variables.

```
#define NUM_FLAG_BITS 200

ATOMIC_DEFINE(flag_bits, NUM_FLAG_BITS);

/* set specified flag bit & return its previous value */
int set_flag_bit(int bit_position)
{
    return (int)atomic_set_bit(flag_bits, bit_position);
}
```

Memory Ordering For consistency and correctness, all Zephyr atomic APIs are expected to include a full memory barrier (in the sense of e.g. "serializing" instructions on x86, "DMB" on ARM, or a "sequentially consistent" operation as defined by the C++ memory model) where needed by hardware to guarantee a reliable picture across contexts. Any architecture-specific implementations are responsible for ensuring this behavior.

Suggested Uses Use an atomic variable to implement critical section processing that only requires the manipulation of a single 32-bit value.

Use multiple atomic variables to implement critical section processing on a set of flag bits in a bit array longer than 32 bits.

> **Note**
>
> Using atomic variables is typically far more efficient than using other techniques to implement critical sections such as using a mutex or locking interrupts.

Configuration Options Related configuration options:

- `CONFIG_ATOMIC_OPERATIONS_BUILTIN`
- `CONFIG_ATOMIC_OPERATIONS_ARCH`
- `CONFIG_ATOMIC_OPERATIONS_C`

> **Important**
>
> All atomic services APIs can be used by both threads and ISRs.

API Reference
Atomic Services APIs

Floating Point Services

The kernel allows threads to use floating point registers on board configurations that support these registers.

> **Note**
>
> Floating point services are currently available only for boards based on ARM Cortex-M SoCs supporting the Floating Point Extension, the Intel x86 architecture, the SPARC architecture and ARCv2 SoCs supporting the Floating Point Extension. The services provided are architecture specific.
>
> The kernel does not support the use of floating point registers by ISRs.

- *Concepts*
  - *No FP registers mode*
  - *Unshared FP registers mode*
  - *Shared FP registers mode*
- *Implementation*
  - *Performing Floating Point Arithmetic*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts The kernel can be configured to provide only the floating point services required by an application. Three modes of operation are supported, which are described below. In addition, the kernel's support for the SSE registers can be included or omitted, as desired.

A thread's participation in floating point context preservation can also be managed at run time. `k_float_enable()` enables preservation for a thread and accepts options – such as `K_FP_REGS` and `K_SSE_REGS` – that select which register sets are preserved, while `k_float_disable()` turns preservation off again. On targets that do not implement floating point context management these routines return `-ENOTSUP`, and they return `-EINVAL` if the request is otherwise invalid and cannot be performed.

No FP registers mode This mode is used when the application has no threads that use floating point registers. It is the kernel's default floating point services mode.

If a thread uses any floating point register, the kernel generates a fatal error condition and aborts the thread.

Unshared FP registers mode This mode is used when the application has only a single thread that uses floating point registers.

On x86 platforms, the kernel initializes the floating point registers so they can be used by any thread (initialization in skipped on ARM Cortex-M platforms and ARCv2 platforms). The floating point registers are left unchanged whenever a context switch occurs.

> **Note**
>
> The behavior is undefined, if two or more threads attempt to use the floating point registers, as the kernel does not attempt to detect (or prevent) multiple threads from using these registers.

Shared FP registers mode This mode is used when the application has two or more threads that use floating point registers. Depending upon the underlying CPU architecture, the kernel supports one or more of the following thread sub-classes:

- non-user: A thread that cannot use any floating point registers
- FPU user: A thread that can use the standard floating point registers
- SSE user: A thread that can use both the standard floating point registers and SSE registers

The kernel initializes and enables access to the floating point registers, so they can be used by any thread, then saves and restores these registers during context switches to ensure the computations performed by each FPU user or SSE user are not impacted by the computations performed by the other users.

> **Note**
>
> The Shared FP registers mode is the default Floating Point Services mode in ARM Cortex-M.

ARM Cortex-M architecture (with the Floating Point Extension) On the ARM Cortex-M architecture with the Floating Point Extension, the kernel treats *all* threads as FPU users when shared FP registers mode is enabled. This means that any thread is allowed to access the floating point registers. The ARM kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them.

Pretag a thread that intends to use the FP registers by using one of the techniques listed below.

- A statically-created ARM thread can be pretagged by passing the K_FP_REGS option to K_THREAD_DEFINE.
- A dynamically-created ARM thread can be pretagged by passing the K_FP_REGS option to k_thread_create().

Pretagging a thread with the K_FP_REGS option instructs the MPU-based stack protection mechanism to properly configure the size of the thread's guard region to always guarantee stack overflow detection, and enable lazy stacking for the given thread upon thread creation.

During thread context switching the ARM kernel saves the *callee-saved* floating point registers, if the switched-out thread has been using them. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been using the floating point registers, the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers.

Each thread that intends to use the floating point registers must provide an extra 72 bytes of stack space where the callee-saved FP context can be saved.

Lazy Stacking is currently enabled in Zephyr applications on ARM Cortex-M architecture, minimizing interrupt latency, when the floating point context is active.

When the MPU-based stack protection mechanism is not enabled, lazy stacking is always active in the Zephyr application. When the MPU-based stack protection is enabled, the following rules apply with respect to lazy stacking:

- Lazy stacking is activated by default on threads that are pretagged with K_FP_REGS
- Lazy stacking is activated dynamically on threads that are not pretagged with K_FP_REGS, as soon as the kernel detects that they are using the floating point registers.

If an ARM thread does not require use of the floating point registers any more, it can call k_float_disable(). This instructs the kernel not to save or restore its FP context during thread context switching.

Note

The Shared FP registers mode is the default Floating Point Services mode on ARM64. The compiler is free to optimize code using FP/SIMD registers, and library functions such as memcpy are known to make use of them.

ARM64 architecture On the ARM64 (Aarch64) architecture the kernel treats each thread as a FPU user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user.

FPU register usage by ISRs is supported although not recommended. When an ISR uses floating point or SIMD registers, then the access is trapped, the current FPU user context is saved in the thread object and the ISR is resumed with interrupts disabled so to prevent another IRQ from interrupting the ISR and potentially requesting FPU usage. Because ISR don't have a persistent register context, there are no provision for saving an ISR's FPU context either, hence the IRQ disabling.

Each thread object becomes 512 bytes larger when Shared FP registers mode is enabled.

ARCv2 architecture On the ARCv2 architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques.

- A statically-created ARC thread can be tagged by passing the K_FP_REGS option to K_THREAD_DEFINE.

- A dynamically-created ARC thread can be tagged by passing the `K_FP_REGS` to `k_thread_create()`.

If an ARC thread does not require use of the floating point registers any more, it can call `k_float_disable()`. This instructs the kernel not to save or restore its FP context during thread context switching.

During thread context switching the ARC kernel saves the *callee-saved* floating point registers, if the switched-out thread has been using them. Additionally, the *caller-saved* floating point registers are saved on the thread's stack. If the switched-in thread has been using the floating point registers, the kernel restores the *callee-saved* FP registers of the switched-in thread and the *caller-saved* FP context is restored from the thread's stack. Thus, the kernel does not save or restore the FP context of threads that are not using the FP registers. An extra 16 bytes (single floating point hardware) or 32 bytes (double floating point hardware) of stack space is required to load and store floating point registers.

RISC-V architecture On the RISC-V architecture the kernel treats each thread as an FPU user on a case-by-case basis with the FPU access allocated on demand. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the FPU registers are *not* saved when switching from an FPU user to a non-user thread (or an FPU user that doesn't touch the FPU during its scheduling slot), and then back to the original FPU user.

FPU register usage by ISRs is supported although not recommended. When an ISR uses floating point or SIMD registers, then the access is trapped, the current FPU user context is saved in the thread object and the ISR is resumed with interrupts disabled so to prevent another IRQ from interrupting the ISR and potentially requesting FPU usage. Because ISR don't have a persistent register context, there are no provision for saving an ISR's FPU context either, hence the IRQ disabling.

As an optimization, the FPU context is preemptively restored upon scheduling back an "active FPU user" thread that had its FPU context saved away due to FPU usage by another thread. Active FPU users are so designated when they make the FPU state "dirty" during their most recent scheduling slot before being scheduled out. So if a thread doesn't modify the FPU state within its scheduling slot and another thread claims the FPU for itself afterwards then that first thread will be subjected to the on-demand regime and won't have its FPU context restored until it attempts to access it again. But if that thread does modify the FPU before being scheduled out then it is likely to continue using it when scheduled back in and preemptively restoring its FPU context saves on the exception trap overhead that would occur otherwise.

Each thread object becomes 136 bytes (single-precision floating point hardware) or 264 bytes (double-precision floating point hardware) larger when Shared FP registers mode is enabled.

SPARC architecture On the SPARC architecture, the kernel treats each thread as a non-user or FPU user and the thread must be tagged by one of the following techniques:

- A statically-created thread can be tagged by passing the `K_FP_REGS` option to `K_THREAD_DEFINE`.
- A dynamically-created thread can be tagged by passing the `K_FP_REGS` to `k_thread_create()`.

During thread context switch at exit from interrupt handler, the SPARC kernel saves *all* floating point registers, if the FPU was enabled in the switched-out thread. Floating point registers are saved on the thread's stack. Floating point registers are restored when a thread context is restored iff they were saved at the context save. Saving and restoring of the floating point registers is synchronous and thus not lazy. The FPU is always disabled when an ISR is called (independent of `CONFIG_FPU_SHARING`).

Floating point disabling with `k_float_disable()` is not implemented.

When CONFIG_FPU_SHARING is used, then 136 bytes of stack space is required for each FPU user thread to load and store floating point registers. No extra stack is required if CONFIG_FPU_SHARING is not used.

x86 architecture On the x86 architecture the kernel treats each thread as a non-user, FPU user or SSE user on a case-by-case basis. A "lazy save" algorithm is used during context switching which updates the floating point registers only when it is absolutely necessary. For example, the registers are *not* saved when switching from an FPU user to a non-user thread, and then back to the original FPU user. The following table indicates the amount of additional stack space a thread must provide so the registers can be saved properly.

Thread type	FP register use	Extra stack space required
cooperative	any	0 bytes
preemptive	none	0 bytes
preemptive	FPU	108 bytes
preemptive	SSE	464 bytes

The x86 kernel automatically detects that a given thread is using the floating point registers the first time the thread accesses them. The thread is tagged as an SSE user if the kernel has been configured to support the SSE registers, or as an FPU user if the SSE registers are not supported. If this would result in a thread that is an FPU user being tagged as an SSE user, or if the application wants to avoid the exception handling overhead involved in auto-tagging threads, it is possible to pretag a thread using one of the techniques listed below.

- A statically-created x86 thread can be pretagged by passing the K_FP_REGS or K_SSE_REGS option to K_THREAD_DEFINE.
- A dynamically-created x86 thread can be pretagged by passing the K_FP_REGS or K_SSE_REGS option to k_thread_create().
- An already-created x86 thread can pretag itself once it has started by passing the K_FP_REGS or K_SSE_REGS option to k_float_enable().

If an x86 thread uses the floating point registers infrequently it can call k_float_disable() to remove its tagging as an FPU user or SSE user. This eliminates the need for the kernel to take steps to preserve the contents of the floating point registers during context switches when there is no need to do so. When the thread again needs to use the floating point registers it can re-tag itself as an FPU user or SSE user by calling k_float_enable().

Implementation

Performing Floating Point Arithmetic No special coding is required for a thread to use floating point arithmetic if the kernel is properly configured.

The following code shows how a routine can use floating point arithmetic to avoid overflow issues when computing the average of a series of integer values.

```
int average(int *values, int num_values)
{
    double sum;
    int i;

    sum = 0.0;

    for (i = 0; i < num_values; i++) {
        sum += *values;
        values++;
```

(continues on next page)

(continued from previous page)

```
    }

    return (int)((sum / num_values) + 0.5);
}
```

Suggested Uses Use the kernel floating point services when an application needs to perform floating point operations.

Configuration Options To configure unshared FP registers mode, enable the `CONFIG_FPU` configuration option and leave the `CONFIG_FPU_SHARING` configuration option disabled.

To configure shared FP registers mode, enable both the `CONFIG_FPU` configuration option and the `CONFIG_FPU_SHARING` configuration option. Also, ensure that any thread that uses the floating point registers has sufficient added stack space for saving floating point register values during context switches, as described above.

For x86, use the `CONFIG_X86_SSE` configuration option to enable support for SSEx instructions.

API Reference
Floating Point APIs

Version

Kernel version handling and APIs related to kernel version being used.

API Reference
Version APIs

Assertions

Zephyr provides several assertion facilities for catching programming errors:

- **Runtime assertions** check a condition while the code is running and, if it fails, induce a *fatal error*. The recommended API is the module-aware `ZASSERT()` macro; the older `__ASSERT()` macro is now a thin compatibility layer on top of it and is deprecated.
- **Build assertions** (`BUILD_ASSERT()`) are evaluated entirely at compile-time and always checked.

> **Note**
>
> The runtime `ZASSERT()` macro documented here is unrelated to the lowercase `zassert_*` macros (`zassert_true()`, `zassert_equal()`, ...) provided by the *Ztest* framework. The `zassert_*` macros report test failures, the `ZASSERT()` macro raises a fatal error when a programming error is detected.

Runtime Assertions

ZASSERT() The module-aware assertion API is declared in include/zephyr/sys/zassert.h. Each source file opts into an assertion *module* whose level is a compile-time constant. Because the level is known at compile time, the compiler can optimize the footprint of the assertion code based on the associated assertion level of the module the assert belongs to.

Assertion Levels Every module resolves to one of four levels:

- ZASSERT_LEVEL_OFF – assertions are compiled out entirely.
- ZASSERT_LEVEL_TERSE – assertions are checked; on failure only a fixed ASSERTION FAIL banner is reported. The location, condition, message and arguments are not compiled in.
- ZASSERT_LEVEL_NORMAL – assertions are checked; on failure only the location is reported (ASSERTION FAIL @ file:line). The condition, message and arguments are not compiled in.
- ZASSERT_LEVEL_VERBOSE – assertions are checked; on failure the stringified condition, location and optional message are reported.

CONFIG_ASSERT is the master switch. When it is disabled, every module is forced to ZASSERT_LEVEL_OFF and all ZASSERT() / ZASSERT_MODULE() usage compiles to nothing, regardless of any module's configured level.

note
ZASSERTS footprint reduction relies on compiler optimizations to prune unused conditions, arguments, and string literals at compile time. Lower optimization levels may prevent dead-code elimination, leaving assertion artifacts in the binary despite a low module assertion setting.

Selecting a Module Place ZASSERT_MODULE once at file scope, before any use of ZASSERT in the translation unit:

```
#include <zephyr/sys/zassert.h>

ZASSERT_MODULE(MYMODULE);
```

The module name is an UPPERCASE identifier. Its default level is taken from the Kconfig symbol CONFIG_ASSERT_MODULE_<module>_LEVEL (here CONFIG_ASSERT_MODULE_MYMODULE_LEVEL). A file may override the module default by passing an explicit level as a second argument, for example ZASSERT_MODULE(MYMODULE, ZASSERT_LEVEL_VERBOSE).

Once a module is selected, use ZASSERT like a conditional check with an optional printf()-like message:

```
ZASSERT(x == 3, "x was %d, expected 3", x);
```

If the condition is false and the module's level is at least ZASSERT_LEVEL_TERSE, a fatal error is raised. The location is only compiled in and printed at ZASSERT_LEVEL_NORMAL or above, and the condition, message and its arguments are only compiled in and printed at ZASSERT_LEVEL_VERBOSE.

For headers and inline functions, avoid ZASSERT_MODULE() at file scope, as the selection would leak into every file that includes the header. Use one of the following instead.

Place ZASSERT_MODULE inside the function body. The selection is then block-scoped and does not escape to the includer, and plain ZASSERT works within that function:

```
static inline void f(void *ptr)
{
        ZASSERT_MODULE(MYMODULE);

        ZASSERT(ptr != NULL, "ptr must not be NULL");
}
```

For an assertion with a fixed level, use ZASSERT_TERSE, ZASSERT_NORMAL or ZASSERT_VERBOSE. These forms select the level at the call site and declare nothing in scope:

```
ZASSERT_TERSE(ptr != NULL);
ZASSERT_NORMAL(ptr != NULL);
ZASSERT_VERBOSE(ptr != NULL, "ptr must not be NULL");
```

The ZASSERT_LEVEL_TERSE, ZASSERT_LEVEL_NORMAL and ZASSERT_LEVEL_VERBOSE defines configure module assertion levels, while ZASSERT_TERSE(), ZASSERT_NORMAL() and ZASSERT_VERBOSE() perform fixed-level assertions.

> **Note**
>
> A few rules apply to ZASSERT() and its file-scope module:
>
> - ZASSERT_MODULE() must appear before the first ZASSERT() in the translation unit, and only one module may be selected per file.
> - Using ZASSERT() with no module in scope is a compile error. Select a module first, or use the fixed-level ZASSERT_TERSE(), ZASSERT_NORMAL() or ZASSERT_VERBOSE() form.
> - CONFIG_ASSERT remains the master switch: when it is disabled the module level is forced to ZASSERT_LEVEL_OFF regardless of the configured level.

Defining a Module's Kconfig Level The CONFIG_ASSERT_MODULE_<module>_LEVEL symbol is generated from the template subsys/debug/zassert/Kconfig.template.assert. Source it from a Kconfig file, setting the module name and a human-readable description first:

```
module = MYMODULE
module-str = the MYMODULE assert module
source "subsys/debug/zassert/Kconfig.template.assert"
```

This produces a user-facing Off / Terse / Normal / Verbose choice and the derived, non-assignable integer symbol CONFIG_ASSERT_MODULE_MYMODULE_LEVEL consumed by ZASSERT_MODULE(MYMODULE). The choice defaults to Verbose and stays overridable from prj.conf.

Example The assert sample demonstrates enabling verbose assertions for a single file while the master assertion switch is enabled. A condensed version:

```
#include <zephyr/kernel.h>
#include <zephyr/sys/zassert.h>

ZASSERT_MODULE(MYMODULE);

int main(void)
{
        int x = 2;

        ZASSERT(x == 3, "x was %d, expected 3", x);

        return 0;
}
```

With CONFIG_ASSERT_MODULE_MYMODULE_LEVEL_VERBOSE=y the failing check produces:

```
ASSERTION FAIL [x == 3] @ .../src/main.c:...
x was 2, expected 3
```

Customizing the Failure Behavior The entire assertion cold path is consolidated into a small set of weak, overridable functions declared in include/zephyr/sys/zassert.h and implemented in subsys/debug/zassert/zassert.c:

- `zassert_fail()` reports a failed assertion (location, and when a message is present the message and its arguments) and then invokes `zassert_post_action()`. Overriding it is the single surface for capturing or redirecting the whole assertion output.
- `zassert_post_action()` takes the terminal action. The default implementation invokes `k_oops()` if the failing thread was running in user mode, and `k_panic()` otherwise.
- `zassert_vprint()` is the single primitive through which all assertion text flows. Override it to capture or redirect every assertion message from one place. `zassert_print()` is a variadic convenience wrapper around it, used by the legacy `__ASSERT_PRINT()` compatibility shims.

When `CONFIG_ASSERT_TEST` is enabled, the post action handler is allowed to return (rather than abort) so that tests can validate assertion behavior by installing a custom hook.

Build Assertions Zephyr provides a macro for performing build-time assertion checks. It is evaluated completely at compile-time and always checked.

BUILD_ASSERT() This has the same semantics as C's `_Static_assert` or C++'s `static_assert`. If the evaluation fails, a build error will be generated by the compiler. If the compiler supports it, the provided message will be printed to provide further context.

Unlike `__ASSERT()`, the message must be a static string or string concatenation of static strings. The macro does not support formatting or variable arguments.

For example, suppose this check fails:

```
BUILD_ASSERT(FOO == 2000, "Invalid value of FOO, expected 2000, got "␣
↪STRINGIFY(FOO));
```

With GCC, the output resembles:

```
     tests/kernel/fatal/src/main.c: In function 'test_main':
     include/zephyr/toolchain/gcc.h:28:37: error: static assertion failed:
"Invalid value of FOO, expected 2000, got 1000"
      #define BUILD_ASSERT(EXPR, MSG) _Static_assert(EXPR, "" MSG)
                                      ^~~~~~~~~~~~~~
     tests/kernel/fatal/src/main.c:370:2: note: in expansion of macro 'BUILD_ASSERT
↪'
       BUILD_ASSERT(FOO == 2000,
       ^~~~~~~~~~~~~~~~
```

Legacy __ASSERT() The `__ASSERT()` family, declared in include/zephyr/sys/__assert.h, predates `ZASSERT()` and is now a compatibility shim using the built-in `DEFAULT` assertion module. New code should prefer `ZASSERT()` with a dedicated module.

> **Note**
>
> `__ASSERT()` and the `CONFIG_ASSERT*` Kconfig options are deprecated. They continue to work through the `DEFAULT` module, but the underlying assert API may change in future releases.

The `DEFAULT` module is enabled by `CONFIG_ASSERT`. Its level is controlled by `CONFIG_ASSERT_MODULE_DEFAULT_LEVEL`, configured through the `Off` / `Terse` / `Normal` / `Verbose` choice (`CONFIG_ASSERT_MODULE_DEFAULT_LEVEL_OFF` / `CONFIG_ASSERT_MODULE_DEFAULT_LEVEL_TERSE` / `CONFIG_ASSERT_MODULE_DEFAULT_LEVEL_NORMAL`

/ CONFIG_ASSERT_MODULE_DEFAULT_LEVEL_VERBOSE). Assertions are enabled by default when running Zephyr test cases, as configured by the CONFIG_TEST option.

The deprecated legacy symbols are still honored and derive the DEFAULT module level: CONFIG_ASSERT_VERBOSE maps to Verbose, CONFIG_ASSERT_NO_COND_INFO, CONFIG_ASSERT_NO_MSG_INFO and CONFIG_ASSERT_NO_FILE_INFO map to Terse, and a CONFIG_ASSERT_LEVEL == 0 leaves it Off. When none of these are set, the DEFAULT module choice falls through to its own CONFIG_ASSERT_MODULE_DEFAULT_LEVEL_VERBOSE default, preserving the legacy behavior of verbose assertions as the default. To disable all assertions regardless of how the level was configured, set CONFIG_ASSERT=n.

API Reference

Module based assert handling

Fatal Errors

Software Errors Triggered in Source Code Zephyr provides several methods for inducing fatal error conditions through either build-time checks, conditionally compiled assertions, or deliberately invoked panic or oops conditions.

Failed runtime assertions (ZASSERT() and the legacy __ASSERT()) induce a fatal error. See *Assertions* for details on the assertion APIs and how their failure behavior is configured.

Kernel Oops A kernel oops is a software triggered fatal error invoked by k_oops(). This should be used to indicate an unrecoverable condition in application logic.

The fatal error reason code generated will be K_ERR_KERNEL_OOPS.

Kernel Panic A kernel error is a software triggered fatal error invoked by k_panic(). This should be used to indicate that the Zephyr kernel is in an unrecoverable state. Implementations of k_sys_fatal_error_handler() should not return if the kernel encounters a panic condition, as the entire system needs to be reset.

Threads running in user mode are not permitted to invoke k_panic(), and doing so will generate a kernel oops instead. Otherwise, the fatal error reason code generated will be K_ERR_KERNEL_PANIC.

Exceptions

Spurious Interrupts If the CPU receives a hardware interrupt on an interrupt line that has not had a handler installed with IRQ_CONNECT() or irq_connect_dynamic(), then the kernel will generate a fatal error with the reason code K_ERR_SPURIOUS_IRQ().

Stack Overflows In the event that a thread pushes more data onto its execution stack than its stack buffer provides, the kernel may be able to detect this situation and generate a fatal error with a reason code of K_ERR_STACK_CHK_FAIL.

If a thread is running in user mode, then stack overflows are always caught, as the thread will simply not have permission to write to adjacent memory addresses outside of the stack buffer. Because this is enforced by the memory protection hardware, there is no risk of data corruption to memory that the thread would not otherwise be able to write to.

If a thread is running in supervisor mode, or if CONFIG_USERSPACE is not enabled, depending on configuration stack overflows may or may not be caught. CONFIG_HW_STACK_PROTECTION is supported on some architectures and will catch stack overflows in supervisor mode, including when handling a system call on behalf of a user thread. Typically this is implemented via dedicated CPU features, or read-only MMU/MPU guard regions placed immediately adjacent to the stack buffer. Stack overflows caught in this way can detect the overflow, but cannot guarantee against

data corruption and should be treated as a very serious condition impacting the health of the entire system.

If a platform lacks memory management hardware support, `CONFIG_STACK_SENTINEL` is a software-only stack overflow detection feature which periodically checks if a sentinel value at the end of the stack buffer has been corrupted. It does not require hardware support, but provides no protection against data corruption. Since the checks are typically done at interrupt exit, the overflow may be detected a nontrivial amount of time after the stack actually overflowed.

Finally, Zephyr supports GCC compiler stack canaries via `CONFIG_STACK_CANARIES`. If enabled, the compiler will insert a canary value randomly generated at boot into function stack frames, checking that the canary has not been overwritten at function exit. If the check fails, the compiler invokes `__stack_chk_fail()`, whose Zephyr implementation invokes a fatal stack overflow error. An error in this case does not indicate that the entire stack buffer has overflowed, but instead that the current function stack frame has been corrupted. See the compiler documentation for more details.

By default a single canary value, generated at boot, is shared by all threads. When `CONFIG_STACK_CANARIES_TLS` is enabled, the canary is instead stored in *thread-local storage* so that each thread has its own value, making the canary location and value harder to predict at the cost of additional per-thread setup.

As a complementary hardening measure, `CONFIG_STACK_POINTER_RANDOM` applies a randomized offset to each thread's initial stack pointer when the thread is created. This is a limited form of address space layout randomization that makes the location of any given stack frame non-deterministic, hindering some classes of security attack, at the cost of consuming up to the configured number of bytes from each thread's stack region.

Other Exceptions Any other type of unhandled CPU exception will generate an error code of `K_ERR_CPU_EXCEPTION`.

Fatal Error Handling The policy for what to do when encountering a fatal error is determined by the implementation of the `k_sys_fatal_error_handler()` function. This function has a default implementation with weak linkage that calls `LOG_PANIC()` to dump all pending logging messages and then unconditionally halts the system with `k_fatal_halt()`.

Applications are free to implement their own error handling policy by overriding the implementation of `k_sys_fatal_error_handler()`. If the implementation returns, the faulting thread will be aborted and the system will otherwise continue to function. See the documentation for this function for additional details and constraints.

API Reference
Fatal error APIs

Thread Local Storage (TLS)

Thread Local Storage (TLS) allows variables to be allocated on a per-thread basis. These variables are stored in the thread stack which means every thread has its own copy of these variables.

Zephyr currently requires toolchain support for TLS.

Configuration To enable thread local storage in Zephyr, `CONFIG_THREAD_LOCAL_STORAGE` needs to be enabled. Note that this option may not be available if the architecture or the SoC does not have the hidden option `CONFIG_ARCH_HAS_THREAD_LOCAL_STORAGE` enabled, which means the architecture or the SoC does not have the necessary code to support thread local storage and/or the toolchain does not support TLS.

`CONFIG_ERRNO_IN_TLS` can be enabled together with `CONFIG_ERRNO` to let the variable `errno` be a thread local variable. This allows user threads to access the value of `errno` without making a system call.

Declaring and Using Thread Local Variables The macro `Z_THREAD_LOCAL` can be used to declare thread local variables.

For example, to declare a thread local variable in header files:

```
extern Z_THREAD_LOCAL int i;
```

And to declare the actual variable in source files:

```
Z_THREAD_LOCAL int i;
```

Keyword `static` can also be used to limit the variable within a source file:

```
static Z_THREAD_LOCAL int j;
```

Using the thread local variable is the same as using other variable, for example:

```
void testing(void) {
    i = 10;
}
```

3.2 Device Driver Model

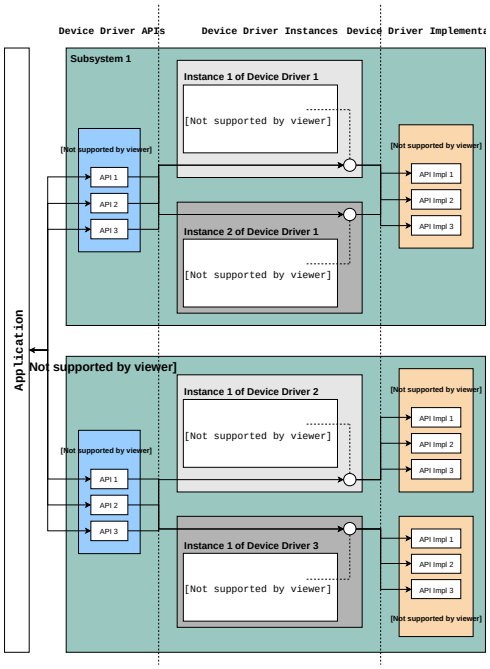
3.2.1 Introduction

The Zephyr kernel supports a variety of device drivers. Whether a driver is available depends on the board and the driver.

The Zephyr device model provides a consistent device model for configuring the drivers that are part of a system. The device model is responsible for initializing all the drivers configured into the system.

Each type of driver (e.g. UART, SPI, I2C) is supported by a generic type API.

In this model the driver fills in the pointer to the structure containing the function pointers to its API functions during driver initialization. These structures are placed into the RAM section in initialization level order.



3.2.2 Standard Drivers

Device drivers which are present on all supported board configurations are listed below.

- **Interrupt controller**: This device driver is used by the kernel's interrupt management subsystem.
- **Timer**: This device driver is used by the kernel's system clock and hardware clock subsystem.
- **Serial communication**: This device driver is used by the kernel's system console subsystem.
- **Entropy**: This device driver provides a source of entropy numbers for the random number generator subsystem.

> **Important**
>
> Use the *random API functions* for random values. *Entropy functions* should not be directly used as a random number generator source as some hardware implementations are designed to be an entropy seed source for random number generators and will not provide cryptographically secure random number streams.

3.2.3 Synchronous Calls

Zephyr provides a set of device drivers for multiple boards. Each driver should support an interrupt-based implementation, rather than polling, unless the specific hardware does not provide any interrupt.

High-level calls accessed through device-specific APIs, such as `i2c.h` or `spi.h`, are usually intended as synchronous. Thus, these calls should be blocking.

3.2.4 Driver APIs

The following APIs for device drivers are provided by `device.h`. The APIs are intended for use in device drivers only and should not be used in applications.

`DEVICE_DEFINE()`
: Create device object and related data structures including setting it up for boot-time initialization.

`DEVICE_NAME_GET()`
: Converts a device identifier to the global identifier for a device object.

`DEVICE_GET()`
: Obtain a pointer to a device object by name.

`DEVICE_DECLARE()`
: Declare a device object. Use this when you need a forward reference to a device that has not yet been defined.

`DEVICE_API()`
: Wrap a driver API declaration to assign it to its respective linker section. Use of this macro is mandatory for any driver implementing an upstream driver class, as `DEVICE_API_GET` and `DEVICE_API_IS` rely on API instances being placed in their class linker section to validate the device's API class at runtime.

3.2.5 Driver Data Structures

The device initialization macros populate some data structures at build time which are split into read-only and runtime-mutable parts. At a high level we have:

```
struct device {
      const char *name;
      const void *config;
      const void *api;
      void * const data;
};
```

The `config` member is for read-only configuration data set at build time. For example, base memory mapped IO addresses, IRQ line numbers, or other fixed physical characteristics of the device. This is the `config` pointer passed to `DEVICE_DEFINE()` and related macros.

The `data` struct is kept in RAM, and is used by the driver for per-instance runtime housekeeping. For example, it may contain reference counts, semaphores, scratch buffers, etc.

The `api` struct maps generic subsystem APIs to the device-specific implementations in the driver. It is typically read-only and populated at build time. The next section describes this in more detail.

3.2.6 Subsystems and API Structures

Most drivers will be implementing a device-independent subsystem API. Applications can simply program to that generic API, and application code is not specific to any particular driver implementation.

Driver API instances must be declared with `DEVICE_API()` so that they are assigned to their respective API linker section. This is required for `DEVICE_API_GET` and `DEVICE_API_IS` to validate at runtime that a device's API belongs to the expected class.

A subsystem API definition typically looks like this:

```
typedef int (*subsystem_do_this_t)(const struct device *dev, int foo, int bar);
typedef void (*subsystem_do_that_t)(const struct device *dev, void *baz);

__subsystem struct subsystem_driver_api {
      subsystem_do_this_t do_this;
      subsystem_do_that_t do_that;
};

static inline int subsystem_do_this(const struct device *dev, int foo, int bar)
{
      return DEVICE_API_GET(subsystem, dev)->do_this(dev, foo, bar);
}

static inline void subsystem_do_that(const struct device *dev, void *baz)
{
      DEVICE_API_GET(subsystem, dev)->do_that(dev, baz);
}
```

A driver implementing a particular subsystem will define the real implementation of these APIs, and populate an instance of subsystem_driver_api structure using the `DEVICE_API()` wrapper:

```
static int my_driver_do_this(const struct device *dev, int foo, int bar)
{
      ...
}

static void my_driver_do_that(const struct device *dev, void *baz)
{
      ...
```

(continues on next page)

(continued from previous page)

```
}

static DEVICE_API(subsystem, my_driver_api_funcs) = {
      .do_this = my_driver_do_this,
      .do_that = my_driver_do_that,
};
```

The driver would then pass `my_driver_api_funcs` as the `api` argument to `DEVICE_DEFINE()`.

> **Note**
>
> Since pointers to the API functions are referenced in the `api` struct, they will always be included in the binary even if unused; `gc-sections` linker option will always see at least one reference to them. Providing for link-time size optimizations with driver APIs in most cases requires that the optional feature be controlled by a Kconfig option.

3.2.7 API Class Inheritance

A subsystem API can extend another subsystem API, forming a parent-child relationship. This allows a device implementing the child API to also be recognized as implementing the parent API. For example, an I3C controller extends the I2C API, so it can be used wherever an I2C device is expected.

To define a child API, embed the parent API struct as the **first member** of the child struct and declare the relationship with `DEVICE_API_EXTENDS`:

```
__subsystem struct child_driver_api {
      struct subsystem_driver_api parent_api;
      child_do_extra_t do_extra;
};

DEVICE_API_EXTENDS(child, subsystem, parent_api);
```

A driver implementing the child API populates both the parent and child methods:

```
static DEVICE_API(child, my_child_api) = {
      .parent_api = {
              .do_this = my_child_do_this,
              .do_that = my_child_do_that,
      },
      .do_extra = my_child_do_extra,
};
```

With this in place, `DEVICE_API_IS` returns true for both the child and parent classes, and `DEVICE_API_GET` can retrieve the API as either type:

```
/* Both return true for a child device */
DEVICE_API_IS(subsystem, dev);
DEVICE_API_IS(child, dev);

/* Access through parent API */
DEVICE_API_GET(subsystem, dev)->do_this(dev, foo, bar);
```

Multi-level inheritance is supported (e.g. grandchild extends child extends parent). Sibling relationships also work correctly: two different child APIs extending the same parent are distinguished by `DEVICE_API_IS`.

3.2.8 Device-Specific API Extensions

Some devices can be cast as an instance of a driver subsystem such as GPIO, but provide additional functionality that cannot be exposed through the standard API. These devices combine subsystem operations with device-specific APIs, described in a device-specific header.

A device-specific API definition typically looks like this:

```
#include <zephyr/drivers/subsystem.h>

/* When extensions need not be invoked from user mode threads */
int specific_do_that(const struct device *dev, int foo);

/* When extensions must be invokable from user mode threads */
__syscall int specific_from_user(const struct device *dev, int bar);

/* Only needed when extensions include syscalls */
#include <zephyr/syscalls/specific.h>
```

A driver implementing extensions to the subsystem will define the real implementation of both the subsystem API and the specific APIs:

```
static int generic_do_this(const struct device *dev, void *arg)
{
   ...
}

static struct generic_api api {
   ...
   .do_this = generic_do_this,
   ...
};

/* supervisor-only API is globally visible */
int specific_do_that(const struct device *dev, int foo)
{
   ...
}

/* syscall API passes through a translation */
int z_impl_specific_from_user(const struct device *dev, int bar)
{
   ...
}

#ifdef CONFIG_USERSPACE

#include <zephyr/internal/syscall_handler.h>

int z_vrfy_specific_from_user(const struct device *dev, int bar)
{
    K_OOPS(K_SYSCALL_SPECIFIC_DRIVER(dev, K_OBJ_DRIVER_GENERIC, &api));
    return z_impl_specific_do_that(dev, bar)
}

#include <zephyr/syscalls/specific_from_user_mrsh.c>

#endif /* CONFIG_USERSPACE */
```

Applications use the device through both the subsystem and specific APIs.

3.2.9 Single Driver, Multiple Instances

Some drivers may be instantiated multiple times in a given system. For example there can be multiple GPIO banks, or multiple UARTS. Each instance of the driver will have a different `config` struct and `data` struct.

Configuring interrupts for multiple drivers instances is a special case. If each instance needs to configure a different interrupt line, this can be accomplished through the use of per-instance configuration functions, since the parameters to `IRQ_CONNECT()` need to be resolvable at build time.

For example, let's say we need to configure two instances of `my_driver`, each with a different interrupt line. In `drivers/subsystem/subsystem_my_driver.h`:

```
typedef void (*my_driver_config_irq_t)(const struct device *dev);

struct my_driver_config {
      DEVICE_MMIO_ROM;
      my_driver_config_irq_t config_func;
};
```

In the implementation of the common init function:

```
void my_driver_isr(const struct device *dev)
{
      /* Handle interrupt */
      ...
}

int my_driver_init(const struct device *dev)
{
      const struct my_driver_config *config = dev->config;

      DEVICE_MMIO_MAP(dev, K_MEM_CACHE_NONE);

      /* Do other initialization stuff */
      ...

      config->config_func(dev);

      return 0;
}
```

Then when the particular instance is declared:

```
#if CONFIG_MY_DRIVER_0

DEVICE_DECLARE(my_driver_0);

static void my_driver_config_irq_0(const struct device *dev)
{
      IRQ_CONNECT(MY_DRIVER_0_IRQ, MY_DRIVER_0_PRI, my_driver_isr,
                  DEVICE_GET(my_driver_0), MY_DRIVER_0_FLAGS);
}

const static struct my_driver_config my_driver_config_0 = {
      DEVICE_MMIO_ROM_INIT(DT_DRV_INST(0)),
      .config_func = my_driver_config_irq_0
}
```

(continues on next page)

(continued from previous page)

```
static struct my_data_0;

DEVICE_DEFINE(my_driver_0, MY_DRIVER_0_NAME, my_driver_init,
              NULL, &my_data_0, &my_driver_config_0,
              POST_KERNEL, MY_DRIVER_0_PRIORITY, &my_api_funcs);

#endif /* CONFIG_MY_DRIVER_0 */
```

Note the use of `DEVICE_DECLARE()` to avoid a circular dependency on providing the IRQ handler argument and the definition of the device itself.

3.2.10 Initialization Levels

Drivers may depend on other drivers being initialized first, or require the use of kernel services. `DEVICE_DEFINE()` and related APIs allow the user to specify at what time during the boot sequence the init function will be executed. Any driver will specify one of three initialization levels:

PRE_KERNEL_1
: Used for devices that have no dependencies, such as those that rely solely on hardware present in the processor/SOC. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. The interrupt subsystem will be configured however so it's OK to set up interrupts. Init functions at this level run on the interrupt stack.

PRE_KERNEL_2
: Used for devices that rely on the initialization of devices initialized as part of the `PRE_KERNEL_1` level. These devices cannot use any kernel services during configuration, since the kernel services are not yet available. Init functions at this level run on the interrupt stack.

POST_KERNEL
: Used for devices that require kernel services during configuration. Init functions at this level run in context of the kernel main task.

Within each initialization level you may specify a priority level, relative to other devices in the same initialization level. The priority level is specified as an integer value in the range 0 to 999; lower values indicate earlier initialization. The priority level must be a decimal integer literal without leading zeroes or sign (e.g. 32), or an equivalent symbolic name (e.g. `\#define MY_INIT_PRIO 32`); symbolic expressions are *not* permitted (e.g. `CONFIG_KERNEL_INIT_PRIORITY_DEFAULT + 5`).

Drivers and other system utilities can determine whether startup is still in pre-kernel states by using the `k_is_pre_kernel()` function.

3.2.11 Deferred initialization

Initialization of devices can also be deferred to a later time. In this case, the device is not automatically initialized by Zephyr at boot time. Instead, the device is initialized when the application calls `device_init()`. To defer a device driver initialization, add the property `zephyr,deferred-init` to the associated device node in the DTS file. For example:

```
/ {
        a-driver@40000000 {
                reg = <0x40000000 0x1000>;
                zephyr,deferred-init;
        };
};
```

3.2.12 System Drivers

In some cases you may just need to run a function at boot. For such cases, the `SYS_INIT` can be used. This macro does not take any config or runtime data structures and there isn't a way to later get a device pointer by name. The same device policies for initialization level and priority apply.

3.2.13 Inspecting the initialization sequence

Device drivers declared with `DEVICE_DEFINE` (or any variations of it) and `SYS_INIT` are processed at boot time and the corresponding initialization functions are called sequentially according to their specified level and priority.

Sometimes it's useful to inspect the final sequence of initialization function call as produced by the linker. To do that, use the `initlevels` CMake target, for example `west build -t initlevels`.

3.2.14 Error handling

In general, it's best to use `__ASSERT()` macros instead of propagating return values unless the failure is expected to occur during the normal course of operation (such as a storage device full). Bad parameters, programming errors, consistency checks, pathological/unrecoverable failures, etc., should be handled by assertions.

When it is appropriate to return error conditions for the caller to check, 0 should be returned on success and a POSIX `errno.h` code returned on failure. See https://github.com/zephyrproject-rtos/zephyr/wiki/Naming-Conventions#return-codes for details about this.

3.2.15 Memory Mapping

On some systems, the linear address of peripheral memory-mapped I/O (MMIO) regions cannot be known at build time:

- The I/O ranges must be probed at runtime from the bus, such as with PCI express
- A memory management unit (MMU) is active, and the physical address of the MMIO range must be mapped into the page tables at some virtual memory location determined by the kernel.

These systems must maintain storage for the MMIO range within RAM and establish the mapping within the driver's init function. Other systems do not care about this and can use MMIO physical addresses directly from DTS and do not need any RAM-based storage for it.

For drivers that may need to deal with this situation, a set of APIs under the DEVICE_MMIO scope are defined, along with a mapping function `device_map()`.

Device Model Drivers with one MMIO region

The simplest case is for drivers which need to maintain one MMIO region. These drivers will need to use the `DEVICE_MMIO_ROM` and `DEVICE_MMIO_RAM` macros in the definitions for their `config_info` and `driver_data` structures, with initialization of the `config_info` from DTS using `DEVICE_MMIO_ROM_INIT`. A call to `DEVICE_MMIO_MAP()` is made within the init function:

```
struct my_driver_config {
   DEVICE_MMIO_ROM; /* Must be first */
   ...
}

struct my_driver_dev_data {
   DEVICE_MMIO_RAM; /* Must be first */
   ...
```

(continues on next page)

(continued from previous page)

```
}

const static struct my_driver_config my_driver_config_0 = {
   DEVICE_MMIO_ROM_INIT(DT_DRV_INST(...)),
   ...
}

int my_driver_init(const struct device *dev)
{
   ...
   DEVICE_MMIO_MAP(dev, K_MEM_CACHE_NONE);
   ...
}

int my_driver_some_function(const struct device *dev)
{
   ...
   /* Write some data to the MMIO region */
   sys_write32(0xDEADBEEF, DEVICE_MMIO_GET(dev));
   ...
}
```

The particular expansion of these macros depends on configuration. On a device with no MMU or PCI-e, `DEVICE_MMIO_MAP` and `DEVICE_MMIO_RAM` expand to nothing.

Device Model Drivers with multiple MMIO regions

Some drivers may have multiple MMIO regions. In addition, some drivers may already be implementing a form of inheritance which requires some other data to be placed first in the `config_info` and `driver_data` structures.

This can be managed with the `DEVICE_MMIO_NAMED` variant macros. These require that `DEV_CFG()` and `DEV_DATA()` macros be defined to obtain a properly typed pointer to the driver's config_info or dev_data structs. For example:

```
struct my_driver_config {
   ...
     DEVICE_MMIO_NAMED_ROM(corge);
     DEVICE_MMIO_NAMED_ROM(grault);
   ...
}

struct my_driver_dev_data {
        ...
     DEVICE_MMIO_NAMED_RAM(corge);
     DEVICE_MMIO_NAMED_RAM(grault);
     ...
}

#define DEV_CFG(_dev) \
   ((const struct my_driver_config *)((_dev)->config))

#define DEV_DATA(_dev) \
   ((struct my_driver_dev_data *)((_dev)->data))

const static struct my_driver_config my_driver_config_0 = {
   ...
```

(continues on next page)

(continued from previous page)

```
   DEVICE_MMIO_NAMED_ROM_INIT(corge, DT_DRV_INST(...)),
   DEVICE_MMIO_NAMED_ROM_INIT(grault, DT_DRV_INST(...)),
   ...
}

int my_driver_init(const struct device *dev)
{
   ...
   DEVICE_MMIO_NAMED_MAP(dev, corge, K_MEM_CACHE_NONE);
   DEVICE_MMIO_NAMED_MAP(dev, grault, K_MEM_CACHE_NONE);
   ...
}

int my_driver_some_function(const struct device *dev)
{
   ...
   /* Write some data to the MMIO regions */
   sys_write32(0xDEADBEEF, DEVICE_MMIO_GET(dev, grault));
   sys_write32(0xF0CCAC1A, DEVICE_MMIO_GET(dev, corge));
   ...
}
```

Device Model Drivers with multiple MMIO regions in the same DT node

Some drivers may have multiple MMIO regions defined into the same DT device node using the `reg-names` property to differentiate them, for example:

```
/dts-v1/;

/ {
        a-driver@40000000 {
                reg = <0x40000000 0x1000>,
                      <0x40001000 0x1000>;
                reg-names = "corge", "grault";
        };
};
```

This can be managed as seen in the previous section but this time using the `DEVICE_MMIO_NAMED_ROM_INIT_BY_NAME` macro instead. So the only difference would be in the driver config struct:

```
const static struct my_driver_config my_driver_config_0 = {
   ...
   DEVICE_MMIO_NAMED_ROM_INIT_BY_NAME(corge, DT_DRV_INST(...)),
   DEVICE_MMIO_NAMED_ROM_INIT_BY_NAME(grault, DT_DRV_INST(...)),
   ...
}
```

Drivers that do not use Zephyr Device Model

Some drivers or driver-like code may not use Zephyr's device model, and alternative storage must be arranged for the MMIO data. An example of this are timer drivers, or interrupt controller code.

This can be managed with the `DEVICE_MMIO_TOPLEVEL` set of macros, for example:

```
DEVICE_MMIO_TOPLEVEL_STATIC(my_regs, DT_DRV_INST(..));

void some_init_code(...)
{
   ...
   DEVICE_MMIO_TOPLEVEL_MAP(my_regs, K_MEM_CACHE_NONE);
   ...
}

void some_function(...)
   ...
   sys_write32(DEVICE_MMIO_TOPLEVEL_GET(my_regs), 0xDEADBEEF);
   ...
}
```

Drivers that do not use DTS

Some drivers may not obtain the MMIO physical address from DTS, such as is the case with PCI-E. In this case the `device_map()` function may be used directly:

```
void some_init_code(...)
{
   ...
   struct pcie_bar mbar;
   bool bar_found = pcie_get_mbar(bdf, index, &mbar);

   device_map(DEVICE_MMIO_RAM_PTR(dev), mbar.phys_addr, mbar.size,␣
↪K_MEM_CACHE_NONE);
   ...
}
```

For these cases, DEVICE_MMIO_ROM directives may be omitted.

3.2.16 API Reference

Device Model

3.3 User Mode

Zephyr offers the capability to run threads at a reduced privilege level which we call user mode. The current implementation is designed for devices with MPU hardware.

For details on creating threads that run in user mode, please see *Lifecycle*.

3.3.1 Overview

Threat Model

User mode threads are considered to be untrusted by Zephyr and are therefore isolated from other user mode threads and from the kernel. A flawed or malicious user mode thread cannot leak or modify the private data/resources of another thread or the kernel, and cannot interfere with or control another user mode thread or the kernel.

Example use-cases of Zephyr's user mode features:

- The kernel can protect against many unintentional programming errors which could otherwise silently or spectacularly corrupt the system.

- The kernel can sandbox complex data parsers such as interpreters, network protocols, and filesystems such that malicious third-party code or data cannot compromise the kernel or other threads.
- The kernel can support the notion of multiple logical "applications", each with their own group of threads and private data structures, which are isolated from each other if one crashes or is otherwise compromised.

Design Goals For threads running in a non-privileged CPU state (hereafter referred to as 'user mode') we aim to protect against the following:

- We prevent access to memory not specifically granted, or incorrect access to memory that has an incompatible policy, such as attempting to write to a read-only area.
  - Access to thread stack buffers will be controlled with a policy which partially depends on the underlying memory protection hardware.
    - A user thread will by default have read/write access to its own stack buffer.
    - A user thread will never by default have access to user thread stacks that are not members of the same memory domain.
    - A user thread will never by default have access to thread stacks owned by a supervisor thread, or thread stacks used to handle system call privilege elevations, interrupts, or CPU exceptions.
    - A user thread may have read/write access to the stacks of other user threads in the same memory domain, depending on hardware.
      - On MPU systems, threads may only access their own stack buffer.
      - On MMU systems, threads may access any user thread stack in the same memory domain. Portable code should not assume this.
  - By default, program text and read-only data are accessible to all threads on read-only basis, kernel-wide. This policy may be adjusted.
  - User threads by default are not granted default access to any memory except what is noted above.
- We prevent use of device drivers or kernel objects not specifically granted, with the permission granularity on a per object or per driver instance basis.
- We validate kernel or driver API calls with incorrect parameters that would otherwise cause a crash or corruption of data structures private to the kernel. This includes:
  - Using the wrong kernel object type.
  - Using parameters outside of proper bounds or with nonsensical values.
  - Passing memory buffers that the calling thread does not have sufficient access to read or write, depending on the semantics of the API.
  - Use of kernel objects that are not in a proper initialization state.
- We ensure the detection and safe handling of user mode stack overflows.
- We prevent invoking system calls to functions excluded by the kernel configuration.
- We prevent disabling of or tampering with kernel-defined and hardware-enforced memory protections.
- We prevent re-entry from user to supervisor mode except through the kernel-defined system calls and interrupt handlers.
- We prevent the introduction of new executable code by user mode threads, except to the extent to which this is supported by kernel system calls.

We are specifically not protecting against the following attacks:

- The kernel itself, and any threads that are executing in supervisor mode, are assumed to be trusted.
- The toolchain and any supplemental programs used by the build system are assumed to be trusted.
- The kernel build is assumed to be trusted. There is considerable build-time logic for creating the tables of valid kernel objects, defining system calls, and configuring interrupts. The .elf binary files that are worked with during this process are all assumed to be trusted code.
- We can't protect against mistakes made in memory domain configuration done in kernel mode that exposes private kernel data structures to a user thread. RAM for kernel objects should always be configured as supervisor-only.
- It is possible to make top-level declarations of user mode threads and assign them permissions to kernel objects. In general, all C and header files that are part of the kernel build producing zephyr.elf are assumed to be trusted.
- We do not protect against denial of service attacks through thread CPU starvation. Zephyr has no thread priority aging and a user thread of a particular priority can starve all threads of lower priority, and also other threads of the same priority if time-slicing is not enabled.
- There are build-time defined limits on how many threads can be active simultaneously, after which creation of new user threads will fail.
- Stack overflows for threads running in supervisor mode may be caught, but the integrity of the system cannot be guaranteed.

High-level Policy Details

Broadly speaking, we accomplish these thread-level memory protection goals through the following mechanisms:

- Any user thread will only have access to a subset of memory: typically its stack, program text, read-only data, and any partitions configured in the *Memory Protection Design* it belongs to. Access to any other RAM must be done on the thread's behalf through system calls, or specifically granted by a supervisor thread using the memory domain APIs. Newly created threads inherit the memory domain configuration of the parent. Threads may communicate with each other by having shared membership of the same memory domains, or via kernel objects such as semaphores and pipes.
- User threads cannot directly access memory belonging to kernel objects. Although pointers to kernel objects are used to reference them, actual manipulation of kernel objects is done through system call interfaces. Device drivers and threads stacks are also considered kernel objects. This ensures that any data inside a kernel object that is private to the kernel cannot be tampered with.
- User threads by default have no permission to access any kernel object or driver other than their own thread object. Such access must be granted by another thread that is either in supervisor mode or has permission on both the receiving thread object and the kernel object being granted access to. The creation of new threads has an option to automatically inherit permissions of all kernel objects granted to the parent, except the parent thread itself.
- For performance and footprint reasons Zephyr normally does little or no parameter error checking for kernel object or device driver APIs. Access from user mode through system calls involves an extra layer of handler functions, which are expected to rigorously validate access permissions and type of the object, check the validity of other parameters through bounds checking or other means, and verify proper read/write access to any memory buffers involved.
- Thread stacks are defined in such a way that exceeding the specified stack space will generate a hardware fault. The way this is done specifically varies per architecture.

Constraints

All kernel objects, thread stacks, and device driver instances must be defined at build time if they are to be used from user mode. Dynamic use-cases for kernel objects will need to go through predefined pools of available objects.

There are some constraints if additional application binary data is loaded for execution after the kernel starts:

- Loaded object code will not be able to define any kernel objects that will be recognized by the kernel. This code will instead need to use APIs for requesting kernel objects from pools.
- Similarly, since the loaded object code will not be part of the kernel build process, this code will not be able to install interrupt handlers, instantiate device drivers, or define system calls, regardless of what mode it runs in.
- Loaded object code that does not come from a verified source should always be entered with the CPU already in user mode.

3.3.2 Memory Protection Design

Zephyr's memory protection design is geared towards microcontrollers with MPU (Memory Protection Unit) hardware. We do support some architectures, such as x86, which have a paged MMU (Memory Management Unit), but in that case the MMU is used like an MPU with an identity page table.

All of the discussion below will be using MPU terminology; systems with MMUs can be considered to have an MPU with an unlimited number of programmable regions.

There are a few different levels on how memory access is configured when Zephyr memory protection features are enabled, which we will describe here:

Boot Time Memory Configuration

This is the configuration of the MPU after the kernel has started up. It should contain the following:

- Any configuration of memory regions which need to have special caching or write-back policies for basic hardware and driver function. Note that most MPUs have the concept of a default memory access policy map, which can be enabled as a "background" mapping for any area of memory that doesn't have an MPU region configuring it. It is strongly recommended to use this to maximize the number of available MPU regions for the end user. On ARMv7-M/ARMv8-M this is called the System Address Map, other CPUs may have similar capabilities. See *Memory Attributes* for information on how to annotate the system map in the device tree.
- A read-only, executable region or regions for program text and ro-data, that is accessible to user mode. This could be further sub-divided into a read-only region for ro-data, and a read-only, executable region for text, but this will require an additional MPU region. This is required so that threads running in user mode can read ro-data and fetch instructions.
- Depending on configuration, user-accessible read-write regions to support extra features like GCOV, HEP, etc.

Assuming there is a background map which allows supervisor mode to access any memory it needs, and regions are defined which grant user mode access to text/ro-data, this is sufficient for the boot time configuration.

Hardware Stack Overflow

`CONFIG_HW_STACK_PROTECTION` is an optional feature which detects stack buffer overflows when the system is running in supervisor mode. This catches issues when the entire stack buffer has

overflowed, and not individual stack frames, use compiler-assisted CONFIG_STACK_CANARIES for that.

Like any crash in supervisor mode, no guarantees can be made about the overall health of the system after a supervisor mode stack overflow, and any instances of this should be treated as a serious error. However it's still very useful to know when these overflows happen, as without robust detection logic the system will either crash in mysterious ways or behave in an undefined manner when the stack buffer overflows.

Some systems implement this feature by creating at runtime a 'guard' MPU region which is set to be read-only and is at either the beginning or immediately preceding the supervisor mode stack buffer. If the stack overflows an exception will be generated.

This feature is optional and is not required to catch stack overflows in user mode; disabling this may free 1-2 MPU regions depending on the MPU design.

Other systems may have dedicated CPU support for catching stack overflows and no extra MPU regions will be required.

Thread Stack

Any thread running in user mode will need access to its own stack buffer. On context switch into a user mode thread, a dedicated MPU region or MMU page table entries will be programmed with the bounds of the stack buffer. A thread exceeding its stack buffer will start pushing data onto memory it doesn't have access to and a memory access violation exception will be generated.

Note that user threads have access to the stacks of other user threads in the same memory domain. This is the minimum required for architectures to support memory domains. Architecture can further restrict access to stacks so each user thread only has access to its own stack if such architecture advertises this capability via CONFIG_ARCH_MEM_DOMAIN_SUPPORTS_ISOLATED_STACKS. This behavior is enabled by default if supported and can be selectively disabled via CONFIG_MEM_DOMAIN_ISOLATED_STACKS if architecture supports both operating modes. However, some architectures may decide to enable this all the time, and thus this option cannot be disabled. Regardless of these kconfigs, user threads cannot access the stacks of other user threads outside of their memory domains.

Thread Resource Pools

A small subset of kernel APIs, invoked as system calls, require heap memory allocations. This memory is used only by the kernel and is not accessible directly by user mode. In order to use these system calls, invoking threads must assign themselves to a resource pool, which is a k_heap object. Memory is drawn from a thread's resource pool using z_thread_malloc() and freed with k_free().

The APIs which use resource pools are as follows, with any alternatives noted for users who do not want heap allocations within their application:

- k_stack_alloc_init() sets up a k_stack with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare k_stacks that are automatically initialized at boot with K_STACK_DEFINE(), or to initialize the k_stack in supervisor mode with k_stack_init().
- k_msgq_alloc_init() sets up a k_msgq object with its storage buffer allocated out of a resource pool instead of a buffer provided by the user. An alternative is to declare a k_msgq that is automatically initialized at boot with K_MSGQ_DEFINE(), or to initialize the k_msgq in supervisor mode with k_msgq_init().
- k_poll() when invoked from user mode, needs to make a kernel-side copy of the provided events array while waiting for an event. This copy is freed when k_poll() returns for any reason.

- `k_queue_alloc_prepend()` and `k_queue_alloc_append()` allocate a container structure to place the data in, since the internal bookkeeping information that defines the queue cannot be placed in the memory provided by the user.
- `k_object_alloc()` allows for entire kernel objects to be dynamically allocated at runtime and a usable pointer to them returned to the caller.

The relevant API is `k_thread_heap_assign()` which assigns a k_heap to draw these allocations from for the target thread.

If the system heap is enabled, then the system heap may be used with `k_thread_system_pool_assign()`, but it is preferable for different logical applications running on the system to have their own pools.

Memory Domains

The kernel ensures that any user thread will have access to its own stack buffer, plus program text and read-only data. The memory domain APIs are the way to grant access to additional blocks of memory to a user thread.

Conceptually, a memory domain is a collection of some number of memory partitions. The maximum number of memory partitions in a domain is limited by the number of available MPU regions. This is why it is important to minimize the number of boot-time MPU regions.

Memory domains are *not* intended to control access to memory from supervisor mode. In some cases this may be unavoidable; for example some architectures do not allow for the definition of regions which are read-only to user mode but read-write to supervisor mode. A great deal of care must be taken when working with such regions to not unintentionally cause the kernel to crash when accessing such a region. Any attempt to use memory domain APIs to control supervisor mode access is at best undefined behavior; supervisor mode access policy is only intended to be controlled by boot-time memory regions.

Memory domain APIs are only available to supervisor mode. The only control user mode has over memory domains is that any user thread's child threads will automatically become members of the parent's domain.

All threads are members of a memory domain, including supervisor threads (even though this has no implications on their memory access). There is a default domain `k_mem_domain_default` which will be assigned to threads if they have not been specifically assigned to a domain, or inherited a memory domain membership from their parent thread. The main thread starts as a member of the default domain.

Memory Partitions Each memory partition consists of a memory address, a size, and access attributes. It is intended that memory partitions are used to control access to system memory. Defining memory partitions are subject to the following constraints:

- The partition must represent a memory region that can be programmed by the underlying memory management hardware, and needs to conform to any underlying hardware constraints. For example, many MPU-based systems require that partitions be sized to some power of two, and aligned to their own size. For MMU-based systems, the partition must be aligned to a page and the size some multiple of the page size.
- Partitions within the same memory domain may not overlap each other. There is no notion of precedence among partitions within a memory domain. Partitions within a memory domain are assumed to have a higher precedence than any boot-time memory regions, however whether a memory domain partition can overlap a boot-time memory region is architecture specific.
- The same partition may be specified in multiple memory domains. For example there may be a shared memory area that multiple domains grant access to.
- Care must be taken in determining what memory to expose in a partition. It is not appropriate to provide direct user mode access to any memory containing private kernel data.

- Memory domain partitions are intended to control access to system RAM. Configuration of memory partitions which do not correspond to RAM may not be supported by the architecture; this is true for MMU-based systems.

There are two ways to define memory partitions: either manually or automatically.

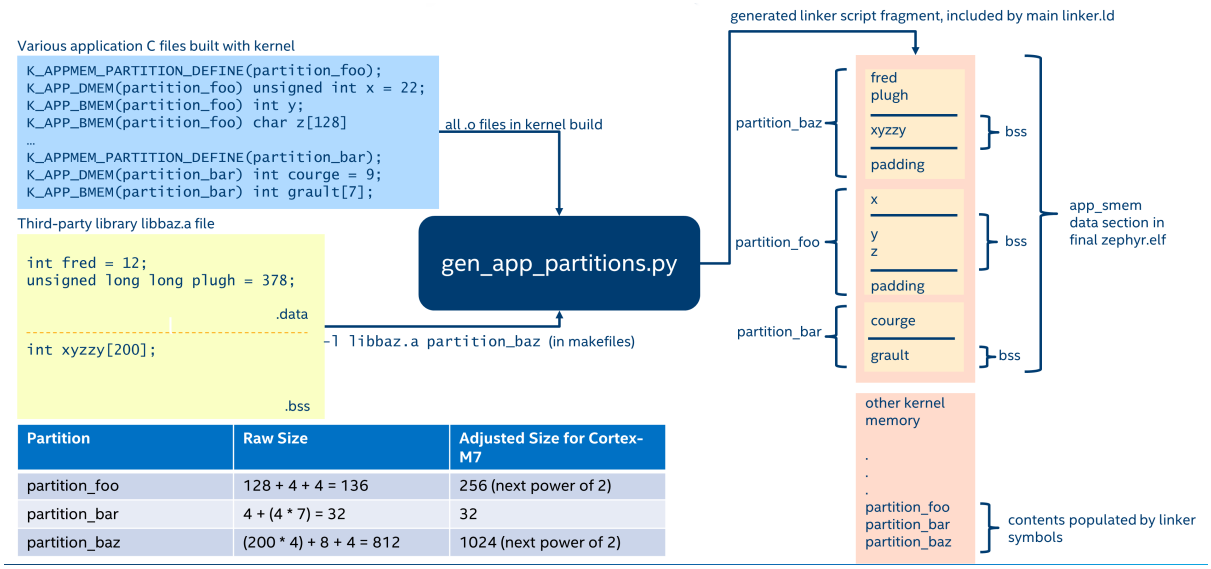
Manual Memory Partitions The following code declares a global array `buf`, and then declares a read-write partition for it which may be added to a domain:

```
uint8_t __aligned(32) buf[32];

K_MEM_PARTITION_DEFINE(my_partition, buf, sizeof(buf),
                       K_MEM_PARTITION_P_RW_U_RW);
```

This does not scale particularly well when we are trying to contain multiple objects spread out across several C files into a single partition.

Automatic Memory Partitions Automatic memory partitions are created by the build system. All globals which need to be placed inside a partition are tagged with their destination partition. The build system will then coalesce all of these into a single contiguous block of memory, zero any BSS variables at boot, and define a memory partition of appropriate base address and size which contains all the tagged data.



Partition	Raw Size	Adjusted Size for Cortex-M7
partition_foo	128 + 4 + 4 = 136	256 (next power of 2)
partition_bar	4 + (4 * 7) = 32	32
partition_baz	(200 * 4) + 8 + 4 = 812	1024 (next power of 2)

Fig. 2: Automatic Memory Domain build flow

Automatic memory partitions are only configured as read-write regions. They are defined with `K_APPMEM_PARTITION_DEFINE()`. Global variables are then routed to this partition using `K_APP_DMEM()` for initialized data and `K_APP_BMEM()` for BSS.

```
#include <zephyr/app_memory/app_memdomain.h>

/* Declare a k_mem_partition "my_partition" that is read-write to
 * user mode. Note that we do not specify a base address or size.
 */
K_APPMEM_PARTITION_DEFINE(my_partition);

/* The global variable var1 will be inside the bounds of my_partition
 * and be initialized with 37 at boot.
 */
```

(continues on next page)

(continued from previous page)

```
K_APP_DMEM(my_partition) int var1 = 37;

/* The global variable var2 will be inside the bounds of my_partition
 * and be zeroed at boot size K_APP_BMEM() was used, indicating a BSS
 * variable.
 */
K_APP_BMEM(my_partition) int var2;
```

The build system will ensure that the base address of `my_partition` will be properly aligned, and the total size of the region conforms to the memory management hardware requirements, adding padding if necessary.

If multiple partitions are being created, a variadic preprocessor macro can be used as provided in `app_macro_support.h`:

```
FOR_EACH(K_APPMEM_PARTITION_DEFINE, part0, part1, part2);
```

Automatic Partitions for Static Library Globals The build-time logic for setting up automatic memory partitions is in `scripts/build/gen_app_partitions.py`. If a static library is linked into Zephyr, it is possible to route all the globals in that library to a specific memory partition with the `--library` argument.

For example, if the Newlib C library is enabled, the Newlib globals all need to be placed in `z_libc_partition`. The invocation of the script in the top-level `CMakeLists.txt` adds the following:

```
gen_app_partitions.py ... --library libc.a z_libc_partition ..
```

For pre-compiled libraries there is no support for expressing this in the project-level configuration or build files; the toplevel `CMakeLists.txt` must be edited.

For Zephyr libraries created using `zephyr_library` or `zephyr_library_named` the `zephyr_library_app_memory` function can be used to specify the memory partition where all globals in the library should be placed.

Pre-defined Memory Partitions There are a few memory partitions which are pre-defined by the system:

- `z_malloc_partition` - This partition contains the system-wide pool of memory used by libc malloc(). Due to possible starvation issues, it is not recommended to draw heap memory from a global pool, instead it is better to define various sys_heap objects and assign them to specific memory domains.
- `z_libc_partition` - Contains globals required by the C library and runtime. Required when using either the Minimal C library or the Newlib C Library. Required when `CONFIG_STACK_CANARIES` is enabled.

Library-specific partitions are listed in include/zephyr/app_memory/partitions.h. For example, to use the MBEDTLS library from user mode, the `k_mbedtls_partition` must be added to the domain.

Memory Domain Usage

Create a Memory Domain A memory domain is defined using a variable of type `k_mem_domain`. It must then be initialized by calling `k_mem_domain_init()`.

The following code defines and initializes an empty memory domain.

```
struct k_mem_domain app0_domain;

k_mem_domain_init(&app0_domain, 0, NULL);
```

Add Memory Partitions into a Memory Domain There are two ways to add memory partitions into a memory domain.

This first code sample shows how to add memory partitions while creating a memory domain.

```
/* the start address of the MPU region needs to align with its size */
uint8_t __aligned(32) app0_buf[32];
uint8_t __aligned(32) app1_buf[32];

K_MEM_PARTITION_DEFINE(app0_part0, app0_buf, sizeof(app0_buf),
                       K_MEM_PARTITION_P_RW_U_RW);

K_MEM_PARTITION_DEFINE(app0_part1, app1_buf, sizeof(app1_buf),
                       K_MEM_PARTITION_P_RW_U_RO);

struct k_mem_partition *app0_parts[] = {
    app0_part0,
    app0_part1
};

k_mem_domain_init(&app0_domain, ARRAY_SIZE(app0_parts), app0_parts);
```

This second code sample shows how to add memory partitions into an initialized memory domain one by one.

```
/* the start address of the MPU region needs to align with its size */
uint8_t __aligned(32) app0_buf[32];
uint8_t __aligned(32) app1_buf[32];

K_MEM_PARTITION_DEFINE(app0_part0, app0_buf, sizeof(app0_buf),
                       K_MEM_PARTITION_P_RW_U_RW);

K_MEM_PARTITION_DEFINE(app0_part1, app1_buf, sizeof(app1_buf),
                       K_MEM_PARTITION_P_RW_U_RO);

k_mem_domain_add_partition(&app0_domain, &app0_part0);
k_mem_domain_add_partition(&app0_domain, &app0_part1);
```

> **Note**
>
> The maximum number of memory partitions is limited by the maximum number of MPU regions or the maximum number of MMU tables.

Memory Domain Assignment Any thread may join a memory domain, and any memory domain may have multiple threads assigned to it. Threads are assigned to memory domains with an API call:

```
k_mem_domain_add_thread(&app0_domain, app_thread_id);
```

If the thread was already a member of some other domain (including the default domain), it will be removed from it in favor of the new one.

In addition, if a thread is a member of a memory domain, and it creates a child thread, that thread will belong to the domain as well.

Remove a Memory Partition from a Memory Domain The following code shows how to remove a memory partition from a memory domain.

```
k_mem_domain_remove_partition(&app0_domain, &app0_part1);
```

The k_mem_domain_remove_partition() API finds the memory partition that matches the given parameter and removes that partition from the memory domain.

Available Partition Attributes When defining a partition, we need to set access permission attributes to the partition. Since the access control of memory partitions relies on either an MPU or MMU, the available partition attributes would be architecture dependent.

The complete list of available partition attributes for a specific architecture is found in the architecture-specific include file `include/zephyr/arch/<arch name>/arch.h`, (for example, include/zephyr/arch/arm/arch.h.) Some examples of partition attributes are:

```
/* Denote partition is privileged read/write, unprivileged read/write */
K_MEM_PARTITION_P_RW_U_RW
/* Denote partition is privileged read/write, unprivileged read-only */
K_MEM_PARTITION_P_RW_U_RO
```

In almost all cases `K_MEM_PARTITION_P_RW_U_RW` is the right choice.

Configuration Options

Related configuration options:

- `CONFIG_MAX_DOMAIN_PARTITIONS`

API Reference

The following memory domain APIs are provided by include/zephyr/kernel.h:

Memory domain APIs

3.3.3 Kernel Objects

A kernel object can be one of three classes of data:

- A core kernel object, such as a semaphore, thread, pipe, etc.
- A thread stack, which is an array of `z_thread_stack_element` and declared with `K_THREAD_STACK_DEFINE()`
- A device driver instance (const struct device) that belongs to one of a defined set of subsystems

The set of known kernel objects and driver subsystems is defined in include/zephyr/sys/kobject.h as `k_objects`.

Kernel objects are completely opaque to user threads. User threads work with addresses to kernel objects when making API calls, but may never dereference these addresses, doing so will cause a memory protection fault. All kernel objects must be placed in memory that is not accessible by user threads.

Since user threads may not directly manipulate kernel objects, all use of them must go through system calls. In order to perform a system call on a kernel object, checks are performed by system call handler functions that the kernel object address is valid and that the calling thread has sufficient permissions to work with it.

Permission on an object also has the semantics of a reference to an object. This is significant for certain object APIs which do temporary allocations, or objects which themselves have been allocated from a runtime memory pool.

If an object loses all references, two events may happen:

- If the object has an associated cleanup function, the cleanup function may be called to release any runtime-allocated buffers the object was using.
- If the object itself was dynamically allocated, the memory for the object will be freed.

Object Placement

Kernel objects that are only used by supervisor threads have no restrictions and can be located anywhere in the binary, or even declared on stacks. However, to prevent accidental or intentional corruption by user threads, they must not be located in any memory that user threads have direct access to.

In order for a static kernel object to be usable by a user thread via system call APIs, several conditions must be met on how the kernel object is declared:

- The object must be declared as a top-level global at build time, such that it appears in the ELF symbol table. It is permitted to declare kernel objects with static scope. The post-build script *scripts/build/gen_kobject_list.py* scans the generated ELF file to find kernel objects and places their memory addresses in a special table of kernel object metadata. Kernel objects may be members of arrays or embedded within other data structures.
- Kernel objects must be located in memory reserved for the kernel. They must not be located in any memory partitions that are user-accessible.
- Any memory reserved for a kernel object must be used exclusively for that object. Kernel objects may not be members of a union data type.

Kernel objects that are found but do not meet the above conditions will not be included in the generated table that is used to validate kernel object pointers passed in from user mode.

The debug output of the *scripts/build/gen_kobject_list.py* script may be useful when debugging why some object was unexpectedly not being tracked. This information will be printed if the script is run with the `--verbose` flag, or if the build system is invoked with verbose output.

Dynamic Objects

Kernel objects may also be allocated at runtime if `CONFIG_DYNAMIC_OBJECTS` is enabled. In this case, the `k_object_alloc()` API may be used to instantiate an object from the calling thread's resource pool. Such allocations may be freed in two ways:

- Supervisor threads may call `k_object_free()` to force a dynamic object to be released.
- If an object's references drop to zero (which happens when no threads have permissions on it) the object will be automatically freed. User threads may drop their own permission on an object with `k_object_release()`, and their permissions are automatically cleared when a thread terminates. Supervisor threads may additionally revoke references for another thread using `k_object_access_revoke()`.

Because permissions are also used for reference counting, it is important for supervisor threads to acquire permissions on objects they are using even though the access control aspects of the permission system are not enforced.

Implementation Details The *scripts/build/gen_kobject_list.py* script is a post-build step which finds all the valid kernel object instances in the binary. It accomplishes this by parsing the DWARF debug information present in the generated ELF file for the kernel.

Any instances of structs or arrays corresponding to kernel objects that meet the object placement criteria will have their memory addresses placed in a special perfect hash table of kernel objects

generated by the 'gperf' tool. When a system call is made and the kernel is presented with a memory address of what may or may not be a valid kernel object, the address can be validated with a constant-time lookup in this table.

Drivers are a special case. All drivers are instances of `device`, but it is important to know what subsystem a driver belongs to so that incorrect operations, such as calling a UART API on a sensor driver object, can be prevented. When a device struct is found, its API pointer is examined to determine what subsystem the driver belongs to.

The table itself maps kernel object memory addresses to instances of `z_object`, which has all the metadata for that object. This includes:

- A bitfield indicating permissions on that object. All threads have a numerical ID assigned to them at build time, used to index the permission bitfield for an object to see if that thread has permission on it. The size of this bitfield is controlled by the `CONFIG_MAX_THREAD_BYTES` option and the build system will generate an error if this value is too low.
- A type field indicating what kind of object this is, which is some instance of `k_objects`.
- A set of flags for that object. This is currently used to track initialization state and whether an object is public or not.
- An extra data field. The semantics of this field vary by object type, see the definition of `z_object_data`.

Dynamic objects allocated at runtime are tracked in a runtime red/black tree which is used in parallel to the gperf table when validating object pointers.

Supervisor Thread Access Permission

Supervisor threads can access any kernel object. However, permissions for supervisor threads are still tracked for two reasons:

- If a supervisor thread calls `k_thread_user_mode_enter()`, the thread will then run in user mode with any permissions it had been granted (in many cases, by itself) when it was a supervisor thread.
- If a supervisor thread creates a user thread with the `K_INHERIT_PERMS` option, the child thread will be granted the same permissions as the parent thread, except the parent thread object.

User Thread Access Permission

By default, when a user thread is created, it will only have access permissions on its own thread object. Other kernel objects by default are not usable. Access to them needs to be explicitly or implicitly granted. There are several ways to do this.

- If a thread is created with the `K_INHERIT_PERMS`, that thread will inherit all the permissions of the parent thread, except the parent thread object.
- A thread that has permission on an object, or is running in supervisor mode, may grant permission on that object to another thread via the `k_object_access_grant()` API. The convenience pseudo-function `k_thread_access_grant()` may also be used, which accepts an arbitrary number of pointers to kernel objects and calls `k_object_access_grant()` on each of them. The thread being granted permission, or the object whose access is being granted, do not need to be in an initialized state. If the caller is from user mode, the caller must have permissions on both the kernel object and the target thread object.
- Supervisor threads may declare a particular kernel object to be a public object, usable by all current and future threads with the `k_object_access_all_grant()` API. You must assume that any untrusted or exploited code will then be able to access the object. Use this API with caution!

- If a thread was declared statically with K_THREAD_DEFINE(), then the K_THREAD_ACCESS_GRANT() may be used to grant that thread access to a set of kernel objects at boot time.

Once a thread has been granted access to an object, such access may be removed with the k_object_access_revoke() API. This API is not available to user threads, however user threads may use k_object_release() to relinquish their own permissions on an object.

Supervisor threads may use k_object_access_revoke_others() to revoke access from all threads except the caller. This is useful to reset the permissions of an object in cases where access was previously given to an untrusted user thread, which may have in turn given it to others. This API also reverts a public access state which may have been set previously via k_object_access_all_grant().

API calls from supervisor mode to set permissions on kernel objects that are not being tracked by the kernel will be no-ops. Doing the same from user mode will result in a fatal error for the calling thread.

Objects allocated with k_object_alloc() implicitly grant permission on the allocated object to the calling thread.

Initialization State

Most operations on kernel objects will fail if the object is considered to be in an uninitialized state. The appropriate init function for the object must be performed first.

Some objects will be implicitly initialized at boot:

- Kernel objects that were declared with static initialization macros (such as K_SEM_DEFINE for semaphores) will be in an initialized state at build time.
- Device driver objects are considered initialized after their init function is run by the kernel early in the boot process.

If a kernel object is initialized with a private static initializer, the object must have k_object_init() called on it at some point by a supervisor thread, otherwise the kernel will consider the object uninitialized if accessed by a user thread. This is very uncommon, typically only for kernel objects that are embedded within some larger struct and initialized statically.

```
struct foo {
    struct k_sem sem;
    ...
};

struct foo my_foo = {
    .sem = Z_SEM_INITIALIZER(my_foo.sem, 0, 1),
    ...
};

...
k_object_init(&my_foo.sem);
...
```

Creating New Kernel Object Types

When implementing new kernel features or driver subsystems, it may be necessary to define some new kernel object types. There are different steps needed for creating core kernel objects and new driver subsystems.

Creating New Core Kernel Objects

- In scripts/build/gen_kobject_list.py, add the name of the struct to the kobjects list.

Instances of the new struct should now be tracked.

Creating New Driver Subsystem Kernel Objects All driver instances are `device`. They are differentiated by what API struct they are set to.

- In `scripts/build/gen_kobject_list.py`, add the name of the API struct for the new subsystem to the `subsystems` list.

Driver instances of the new subsystem should now be tracked.

Configuration Options

Related configuration options:

- `CONFIG_USERSPACE`
- `CONFIG_MAX_THREAD_BYTES`

API Reference

User Mode APIs

3.3.4 System Calls

User threads run with a reduced set of privileges than supervisor threads: certain CPU instructions may not be used, and they have access to only a limited part of the memory map. System calls (may) allow user threads to perform operations not directly available to them.

When defining system calls, it is very important to ensure that access to the API's private data is done exclusively through system call interfaces. Private kernel data should never be made available to user mode threads directly. For example, the `k_queue` APIs were intentionally not made available as they store bookkeeping information about the queue directly in the queue buffers which are visible from user mode.

APIs that allow the user to register callback functions that run in supervisor mode should never be exposed as system calls. Reserve these for supervisor-mode access only.

This section describes how to declare new system calls and discusses a few implementation details relevant to them.

Components

All system calls have the following components:

- A **C prototype** prefixed with `__syscall` for the API. It will be declared in some header under `include/` or in another `SYSCALL_INCLUDE_DIRS` directory. This prototype is never implemented manually, instead it gets created by the *scripts/build/gen_syscalls.py* script. What gets generated is an inline function which either calls the implementation function directly (if called from supervisor mode) or goes through privilege elevation and validation steps (if called from user mode).
- An **implementation function**, which is the real implementation of the system call. The implementation function may assume that all parameters passed in have been validated if it was invoked from user mode.
- A **verification function**, which wraps the implementation function and does validation of all the arguments passed in.
- An **unmarshalling function**, which is an automatically generated handler that must be included by user source code.

C Prototype

The C prototype represents how the API is invoked from either user or supervisor mode. For example, to initialize a semaphore:

```
__syscall void k_sem_init(struct k_sem *sem, unsigned int initial_count,
                          unsigned int limit);
```

The `__syscall` attribute is very special. To the C compiler, it simply expands to 'static inline'. However to the post-build *scripts/build/parse_syscalls.py* script, it indicates that this API is a system call. The *scripts/build/parse_syscalls.py* script does some parsing of the function prototype, to determine the data types of its return value and arguments, and has some limitations:

- Array arguments must be passed in as pointers, not arrays. For example, `int foo[]` or `int foo[12]` is not allowed, but should instead be expressed as `int *foo`.
- Function pointers horribly confuse the limited parser. The workaround is to typedef them first, and then express in the argument list in terms of that typedef.
- `__syscall` must be the first thing in the prototype.

The preprocessor is intentionally not used when determining the set of system calls to generate. However, any generated system calls that don't actually have a verification function defined (because the related feature is not enabled in the kernel configuration) will instead point to a special verification for unimplemented system calls. Data type definitions for APIs should not have conditional visibility to the compiler.

Any header file that declares system calls must include a special generated header at the very bottom of the header file. This header follows the naming convention `syscalls/<name of header file>`. For example, at the bottom of include/zephyr/drivers/sensor.h:

```
#include <zephyr/syscalls/sensor.h>
```

C prototype functions must be declared in one of the directories listed in the CMake variable `SYSCALL_INCLUDE_DIRS`. This list always contains `APPLICATION_SOURCE_DIR` when `CONFIG_APPLICATION_DEFINED_SYSCALL` is set, or `${ZEPHYR_BASE}/subsys/testsuite/ztest/include` when `CONFIG_ZTEST` is set. Additional paths can be added to the list through the CMake command line or in CMake code that is run before `find_package(Zephyr ...)` is run. `${ZEPHYR_BASE}/include` is always scanned for potential syscall prototypes.

Note that not all syscalls will be included in the final binaries. CMake functions `zephyr_syscall_header` and `zephyr_syscall_header_ifdef` are used to specify which header files contain syscall prototypes where those syscalls must be present in the final binaries. Note that header files inside directories listed in CMake variable `SYSCALL_INCLUDE_DIRS` will always have their syscalls present in final binaries. To force all syscalls to be included in the final binaries, turn on `CONFIG_EMIT_ALL_SYSCALLS`.

Invocation Context Source code that uses system call APIs can be made more efficient if it is known that all the code inside a particular C file runs exclusively in user mode, or exclusively in supervisor mode. The system will look for the definition of macros `__ZEPHYR_SUPERVISOR__` or `__ZEPHYR_USER__`, typically these will be added to the compiler flags in the build system for the related files.

- If `CONFIG_USERSPACE` is not enabled, all APIs just directly call the implementation function.
- Otherwise, the default case is to make a runtime check to see if the processor is currently running in user mode, and either make the system call or directly call the implementation function as appropriate.
- If `__ZEPHYR_SUPERVISOR__` is defined, then it is assumed that all the code runs in supervisor mode and all APIs just directly call the implementation function. If the code was actually running in user mode, there will be a CPU exception as soon as it tries to do something it isn't allowed to do.

- If `__ZEPHYR_USER__` is defined, then it is assumed that all the code runs in user mode and system calls are unconditionally made.

Implementation Details Declaring an API with `__syscall` causes some code to be generated in C and header files by the *scripts/build/gen_syscalls.py* script, all of which can be found in the project out directory under `include/generated/`:

- The system call is added to the enumerated type of system call IDs, which is expressed in `include/generated/zephyr/syscall_list.h`. It is the name of the API in uppercase, prefixed with `K_SYSCALL_`.
- An entry for the system call is created in the dispatch table `_k_syscall_table`, expressed in `include/generated/zephyr/syscall_dispatch.c`
  - This table only contains syscalls where their corresponding prototypes are declared in header files when `CONFIG_EMIT_ALL_SYSCALLS` is enabled:
    - Indicated by CMake functions `zephyr_syscall_header` and `zephyr_syscall_header_ifdef`, or
    - Under directories specified in CMake variable `SYSCALL_INCLUDE_DIRS`.
- A weak verification function is declared, which is just an alias of the 'unimplemented system call' verifier. This is necessary since the real verification function may or may not be built depending on the kernel configuration. For example, if a user thread makes a sensor subsystem API call, but the sensor subsystem is not enabled, the weak verifier will be invoked instead.
- An unmarshalling function is defined in `include/generated/zephyr/syscalls/<name>_mrsh.c`

The body of the API is created in the generated system header. Using the example of `k_sem_init()`, this API is declared in include/zephyr/kernel.h. At the bottom of include/zephyr/kernel.h is:

```
#include <zephyr/syscalls/kernel.h>
```

Inside this header is the body of `k_sem_init()`:

```
static inline void k_sem_init(struct k_sem * sem, unsigned int initial_count,␣
↪unsigned int limit)
{
#ifdef CONFIG_USERSPACE
        if (z_syscall_trap()) {
                arch_syscall_invoke3(*(uintptr_t *)&sem, *(uintptr_t␣
↪*)&initial_count, *(uintptr_t *)&limit, K_SYSCALL_K_SEM_INIT);
                return;
        }
        compiler_barrier();
#endif
        z_impl_k_sem_init(sem, initial_count, limit);
}
```

This generates an inline function that takes three arguments with void return value. Depending on context it will either directly call the implementation function or go through a system call elevation. A prototype for the implementation function is also automatically generated.

The final layer is the invocation of the system call itself. All architectures implementing system calls must implement the seven inline functions `_arch_syscall_invoke0()` through `_arch_syscall_invoke6()`. These functions marshal arguments into designated CPU registers and perform the necessary privilege elevation. Parameters of API inline function, before being passed as arguments to system call, are C casted to `uintptr_t` which matches size of register. Exception to above is passing 64-bit parameters on 32-bit systems, in which case 64-bit parameters

are split into lower and higher part and passed as two consecutive arguments. There is always a `uintptr_t` type return value, which may be neglected if not needed.

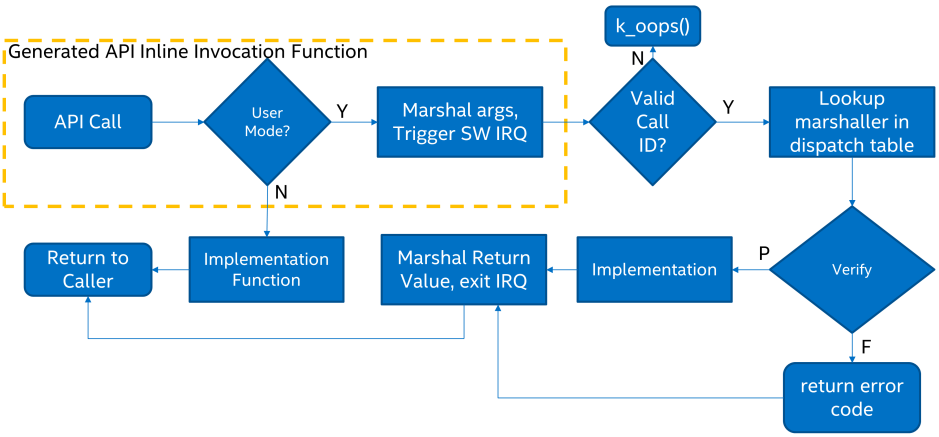


Fig. 3: System Call execution flow

Some system calls may have more than six arguments, but number of arguments passed via registers is limited to six for all architectures. Additional arguments will need to be passed in an array in the source memory space, which needs to be treated as untrusted memory in the verification function. This code (packing, unpacking and validation) is generated automatically as needed in the stub above and in the unmarshalling function.

System calls return `uintptr_t` type value that is C casted, by wrapper, to a return type of API prototype declaration. This means that 64-bit value may not be directly returned, from a system call to its wrapper, on 32-bit systems. To solve the problem the automatically generated wrapper function defines 64-bit intermediate variable, which is considered **untrusted** buffer, on its stack and passes pointer to that variable to the system call, as a final argument. Upon return from the system call the value written to that buffer will be returned by the wrapper function. The problem does not exist on 64-bit systems which are able to return 64-bit values directly.

Implementation Function

The implementation function is what actually does the work for the API. Zephyr normally does little to no error checking of arguments, or does this kind of checking with assertions. When writing the implementation function, validation of any parameters is optional and should be done with assertions.

All implementation functions must follow the naming convention, which is the name of the API prefixed with `z_impl_`. Implementation functions may be declared in the same header as the API as a static inline function or declared in some C file. There is no prototype needed for implementation functions, these are automatically generated.

Verification Function

The verification function runs on the kernel side when a user thread makes a system call. When the user thread makes a software interrupt to elevate to supervisor mode, the common system call entry point uses the system call ID provided by the user to look up the appropriate unmarshalling function for that system call and jump into it. This in turn calls the verification function.

Verification and unmarshalling functions only run when system call APIs are invoked from user mode. If an API is invoked from supervisor mode, the implementation is simply called and there is no software trap.

The purpose of the verification function is to validate all the arguments passed in. This includes:

- Any kernel object pointers provided. For example, the semaphore APIs must ensure that the semaphore object passed in is a valid semaphore and that the calling thread has permission on it.
- Any memory buffers passed in from user mode. Checks must be made that the calling thread has read or write permissions on the provided buffer.
- Any other arguments that have a limited range of valid values.

Verification functions involve a great deal of boilerplate code which has been made simpler by some macros in include/zephyr/internal/syscall_handler.h. Verification functions should be declared using these macros.

Argument Validation Several macros exist to validate arguments:

- `K_SYSCALL_OBJ()` Checks a memory address to assert that it is a valid kernel object of the expected type, that the calling thread has permissions on it, and that the object is initialized.
- `K_SYSCALL_OBJ_INIT()` is the same as `K_SYSCALL_OBJ()`, except that the provided object may be uninitialized. This is useful for verifiers of object init functions.
- `K_SYSCALL_OBJ_NEVER_INIT()` is the same as `K_SYSCALL_OBJ()`, except that the provided object must be uninitialized. This is not used very often, currently only for `k_thread_create()`.
- `K_SYSCALL_MEMORY_READ()` validates a memory buffer of a particular size. The calling thread must have read permissions on the entire buffer.
- `K_SYSCALL_MEMORY_WRITE()` is the same as `K_SYSCALL_MEMORY_READ()` but the calling thread must additionally have write permissions.
- `K_SYSCALL_MEMORY_ARRAY_READ()` validates an array whose total size is expressed as separate arguments for the number of elements and the element size. This macro correctly accounts for multiplication overflow when computing the total size. The calling thread must have read permissions on the total size.
- `K_SYSCALL_MEMORY_ARRAY_WRITE()` is the same as `K_SYSCALL_MEMORY_ARRAY_READ()` but the calling thread must additionally have write permissions.
- `K_SYSCALL_VERIFY_MSG()` does a runtime check of some boolean expression which must evaluate to true otherwise the check will fail. A variant `K_SYSCALL_VERIFY` exists which does not take a message parameter, instead printing the expression tested if it fails. The latter should only be used for the most obvious of tests.
- `K_SYSCALL_DRIVER_OP()` checks at runtime if a driver instance is capable of performing a particular operation. While this macro can be used by itself, it's mostly a building block for macros that are automatically generated for every driver subsystem. For instance, to validate the GPIO driver, one could use the `K_SYSCALL_DRIVER_GPIO()` macro.
- `K_SYSCALL_SPECIFIC_DRIVER()` is a runtime check to verify that a provided pointer is a valid instance of a specific device driver, that the calling thread has permissions on it, and that the driver has been initialized. It does this by checking the API structure pointer that is stored within the driver instance and ensuring that it matches the provided value, which should be the address of the specific driver's API structure.

If any check fails, the macros will return a nonzero value. The macro `K_OOPS()` can be used to induce a kernel oops which will kill the calling thread. This is done instead of returning some error condition to keep the APIs the same when calling from supervisor mode.

Verifier Definition All system calls are dispatched to a verifier function with a prefixed `z_vrfy_` name based on the system call. They have exactly the same return type and argument types as the wrapped system call. Their job is to execute the system call (generally by calling the implementation function) after having validated all arguments.

The verifier is itself invoked by an automatically generated unmarshaller function which takes care of unpacking the register arguments from the architecture layer and casting them to the correct type. This is defined in a header file that must be included from user code, generally somewhere after the definition of the verifier in a translation unit (so that it can be inlined).

For example:

```
static int z_vrfy_k_sem_take(struct k_sem *sem, int32_t timeout)
{
    K_OOPS(K_SYSCALL_OBJ(sem, K_OBJ_SEM));
    return z_impl_k_sem_take(sem, timeout);
}
#include <zephyr/syscalls/k_sem_take_mrsh.c>
```

Verification Memory Access Policies Parameters passed to system calls by reference require special handling, because the value of these parameters can be changed at any time by any user thread that has access to the memory that parameter points to. If the kernel makes any logical decisions based on the contents of this memory, this can open up the kernel to attacks even if checking is done. This is a class of exploits known as TOCTOU (Time Of Check to Time Of Use).

The proper procedure to mitigate these attacks is to make a copies in the verification function, and only perform parameter checks on the copies, which user threads will never have access to. The implementation functions get passed the copy and not the original data sent by the user. The `k_usermode_to_copy()` and `k_usermode_from_copy()` APIs exist for this purpose.

C strings passed from user mode need similar care, since their length is not known in advance and the terminating `NUL` must be located without reading past memory the caller can access. The `k_usermode_string_copy()` and `k_usermode_string_alloc_copy()` helpers safely validate and copy a user-supplied string into kernel-controlled memory before it is used by the implementation function.

There is one exception in place, with respect to large data buffers which are only used to provide a memory area that is either only written to, or whose contents are never used for any validation or control flow. Further discussion of this later in this section.

As a first example, consider a parameter which is used as an output parameter for some integral value:

```
int z_vrfy_some_syscall(int *out_param)
{
    int local_out_param;
    int ret;

    ret = z_impl_some_syscall(&local_out_param);
    K_OOPS(k_usermode_to_copy(out_param, &local_out_param, sizeof(*out_param)));
    return ret;
}
```

Here we have allocated `local_out_param` on the stack, passed its address to the implementation function, and then used `k_usermode_to_copy()` to fill in the memory passed in by the caller.

It might be tempting to do something more concise:

```
int z_vrfy_some_syscall(int *out_param)
{
    K_OOPS(K_SYSCALL_MEMORY_WRITE(out_param, sizeof(*out_param)));
    return z_impl_some_syscall(out_param);
}
```

However, this is unsafe if the implementation ever does any reads to this memory as part of its logic. For example, it could be used to store some counter value, and this could be meddled with

by user threads that have access to its memory. It is by far safest for small integral values to do the copying as shown in the first example.

Some parameters may be input/output. For instance, it's not uncommon to see APIs which pass in a pointer to some `size_t` which is a maximum allowable size, which is then updated by the implementation to reflect the actual number of bytes processed. This too should use a stack copy:

```
int z_vrfy_in_out_syscall(size_t *size_ptr)
{
    size_t size;
    int ret;

    K_OOPS(k_usermode_from_copy(&size, size_ptr, sizeof(size));
    ret = z_impl_in_out_syscall(&size);
    K_OOPS(k_usermode_to_copy(size_ptr, &size, sizeof(size)));
    return ret;
}
```

Many system calls pass in structures or even linked data structures. All should be copied. Typically this is done by allocating copies on the stack:

```
struct bar {
    ...
};

struct foo {
    ...
    struct bar *bar_left;
    struct bar *bar_right;
};

int z_vrfy_must_alloc(struct foo *foo)
{
    int ret;
    struct foo foo_copy;
    struct bar bar_right_copy;
    struct bar bar_left_copy;

    K_OOPS(k_usermode_from_copy(&foo_copy, foo, sizeof(*foo)));
    K_OOPS(k_usermode_from_copy(&bar_right_copy, foo_copy.bar_right,
                                sizeof(struct bar)));
    foo_copy.bar_right = &bar_right_copy;
    K_OOPS(k_usermode_from_copy(&bar_left_copy, foo_copy.bar_left,
                                sizeof(struct bar)));
    foo_copy.bar_left = &bar_left_copy;

    return z_impl_must_alloc(&foo_copy);
}
```

In some cases the amount of data isn't known at compile time or may be too large to allocate on the stack. In this scenario, it may be necessary to draw memory from the caller's resource pool via `z_thread_malloc()`. This should always be considered last resort. Functional safety programming guidelines heavily discourage usage of heap and the fact that a resource pool is used must be clearly documented. Any issues with allocation must be reported, to a caller, with returning the `-ENOMEM` . The `K_OOPS()` should never be used to verify if resource allocation has been successful.

```
struct bar {
    ...
```

(continues on next page)

(continued from previous page)

```
};

struct foo {
    size_t count;
    struct bar *bar_list; /* array of struct bar of size count */
};

int z_vrfy_must_alloc(struct foo *foo)
{
    int ret;
    struct foo foo_copy;
    struct bar *bar_list_copy;
    size_t bar_list_bytes;

    /* Safely copy foo into foo_copy */
    K_OOPS(k_usermode_from_copy(&foo_copy, foo, sizeof(*foo)));

    /* Bounds check the count member, in the copy we made */
    if (foo_copy.count > 32) {
        return -EINVAL;
    }

    /* Allocate RAM for the bar_list, replace the pointer in
     * foo_copy */
    bar_list_bytes = foo_copy.count * sizeof(struct_bar);
    bar_list_copy = z_thread_malloc(bar_list_bytes);
    if (bar_list_copy == NULL) {
        return -ENOMEM;
    }
    K_OOPS(k_usermode_from_copy(bar_list_copy, foo_copy.bar_list,
                                bar_list_bytes));
    foo_copy.bar_list = bar_list_copy;

    ret = z_impl_must_alloc(&foo_copy);

    /* All done with the memory, free it and return */
    k_free(foo_copy.bar_list_copy);
    return ret;
}
```

Finally, we must consider large data buffers. These represent areas of user memory which either have data copied out of, or copied into. It is permitted to pass these pointers to the implementation function directly. The caller's access to the buffer still must be validated with `K_SYSCALL_MEMORY` APIs. The following constraints need to be met:

- If the buffer is used by the implementation function to write data, such as data captured from some MMIO region, the implementation function must only write this data, and never read it.
- If the buffer is used by the implementation function to read data, such as a block of memory to write to some hardware destination, this data must be read without any processing. No conditional logic can be implemented due to the data buffer's contents. If such logic is required a copy must be made.
- The buffer must only be used synchronously with the call. The implementation must not ever save the buffer address and use it asynchronously, such as when an interrupt fires.

```
int z_vrfy_get_data_from_kernel(void *buf, size_t size)
{
    K_OOPS(K_SYSCALL_MEMORY_WRITE(buf, size));
    return z_impl_get_data_from_kernel(buf, size);
}
```

Verification Return Value Policies When verifying system calls, it's important to note which kinds of verification failures should propagate a return value to the caller, and which should simply invoke `K_OOPS()` which kills the calling thread. The current conventions are as follows:

1. For system calls that are defined but not compiled, invocations of these missing system calls are routed to `handler_no_syscall()` which invokes `K_OOPS()`.
2. Any invalid access to memory found by the set of `K_SYSCALL_MEMORY` APIs, `k_usermode_from_copy()`, `k_usermode_to_copy()` should trigger a `K_OOPS`. This happens when the caller doesn't have appropriate permissions on the memory buffer or some size calculation overflowed.
3. Most system calls take kernel object pointers as an argument, checked either with one of the `K_SYSCALL_OBJ` functions, `K_SYSCALL_DRIVER_nnnnn`, or manually using `k_object_validate()`. These can fail for a variety of reasons: missing driver API, bad kernel object pointer, wrong kernel object type, or improper initialization state. These issues should always invoke `K_OOPS()`.
4. Any error resulting from a failed memory heap allocation, often from invoking `z_thread_malloc()`, should propagate `-ENOMEM` to the caller.
5. General parameter checks should be done in the implementation function, in most cases using `CHECKIF()`.
   - The behavior of `CHECKIF()` depends on the kernel configuration, but if user mode is enabled, `CONFIG_RUNTIME_ERROR_CHECKS` is enforced, which guarantees that these checks will be made and a return value propagated.
6. It is totally forbidden for any kind of kernel mode callback function to be registered from user mode. APIs which simply install callbacks shall not be exposed as system calls. Some driver subsystem APIs may take optional function callback pointers. User mode verification functions for these APIs must enforce that these are NULL and should invoke `K_OOPS()` if not.
7. Some parameter checks are enforced only from user mode. These should be checked in the verification function and propagate a return value to the caller if possible.

There are some known exceptions to these policies currently in Zephyr:

- `k_thread_join()` and `k_thread_abort()` are no-ops if the thread object isn't initialized. This is because for threads, the initialization bit pulls double-duty to indicate whether a thread is running, cleared upon exit. See #23030.
- `k_thread_create()` invokes `K_OOPS()` for parameter checks, due to a great deal of existing code ignoring the return value. This will also be addressed by #23030.
- `k_thread_abort()` invokes `K_OOPS()` if an essential thread is aborted, as the function has no return value.
- Various system calls related to logging invoke `K_OOPS()` when bad parameters are passed in as they do not propagate errors.

Configuration Options

Related configuration options:

- `CONFIG_USERSPACE`

- CONFIG_EMIT_ALL_SYSCALLS

APIs

Helper macros for creating system call verification functions are provided in include/zephyr/internal/syscall_handler.h:

- K_SYSCALL_OBJ()
- K_SYSCALL_OBJ_INIT()
- K_SYSCALL_OBJ_NEVER_INIT()
- K_OOPS()
- K_SYSCALL_MEMORY_READ()
- K_SYSCALL_MEMORY_WRITE()
- K_SYSCALL_MEMORY_ARRAY_READ()
- K_SYSCALL_MEMORY_ARRAY_WRITE()
- K_SYSCALL_VERIFY_MSG()
- K_SYSCALL_VERIFY

Functions for invoking system calls are defined in include/zephyr/syscall.h:

- _arch_syscall_invoke0()
- _arch_syscall_invoke1()
- _arch_syscall_invoke2()
- _arch_syscall_invoke3()
- _arch_syscall_invoke4()
- _arch_syscall_invoke5()
- _arch_syscall_invoke6()

3.3.5 MPU Stack Objects

Thread Stack Creation

Thread stacks are declared statically with K_THREAD_STACK_DEFINE().

For architectures which utilize memory protection unit (MPU) hardware, stacks are physically contiguous allocations. This contiguous allocation has implications for the placement of stacks in memory, as well as the implementation of other features such as stack protection and userspace. The implications for placement are directly attributed to the alignment requirements for MPU regions. This is discussed in the memory placement section below.

Stack Guards

Stack protection mechanisms require hardware support that can restrict access to memory. Memory protection units can provide this kind of support. The MPU provides a fixed number of regions. Each region contains information about the start, end, size, and access attributes to be enforced on that particular region.

Stack guards are implemented by using a single MPU region and setting the attributes for that region to not allow write access. If invalid accesses occur, a fault ensues. The stack guard is defined at the bottom (the lowest address) of the stack.

Memory Placement

During stack creation, a set of constraints are enforced on the allocation of memory. These constraints include determining the alignment of the stack and the correct sizing of the stack. During linking of the binary, these constraints are used to place the stacks properly.

The main source of the memory constraints is the MPU design for the SoC. The MPU design may require specific constraints on the region definition. These can include alignment of beginning and end addresses, sizes of allocations, or even interactions between overlapping regions.

Some MPUs require that each region be aligned to a power of two. These SoCs will have `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT` defined. This means that a 1500 byte stack should be aligned to a 2kB boundary and the stack size should also be adjusted to 2kB to ensure that nothing else is placed in the remainder of the region. SoCs which include the unmodified ARM v7m MPU will have these constraints.

Some ARM MPUs use start and end addresses to define MPU regions and both the start and end addresses require 32 byte alignment. An example of this kind of MPU is found in the NXP FRDM K64F.

MPUs may have a region priority mechanisms that use the highest priority region that covers the memory access to determine the enforcement policy. Others may logically OR regions to determine enforcement policy.

Size and alignment constraints may result in stack allocations being larger than the requested size. Region priority mechanisms may result in some added complexity when implementing stack guards.

3.3.6 MPU Backed Userspace

The MPU backed userspace implementation requires the creation of a secondary set of stacks. These stacks exist in a 1:1 relationship with each thread stack defined in the system. The privileged stacks are created as a part of the build process.

A post-build script *scripts/build/gen_kobject_list.py* scans the generated ELF file and finds all of the thread stack objects. A set of privileged stacks, a lookup table, and a set of helper functions are created and added to the image.

During the process of dropping a thread to user mode, the privileged stack information is filled in and later used by the swap and system call infrastructure to configure the MPU regions properly for the thread stack and guard (if applicable).

During system calls, the user mode thread's access to the system call and the passed-in parameters are all validated. The user mode thread is then elevated to privileged mode, the stack is switched to use the privileged stack, and the call is made to the specified kernel API. On return from the kernel API, the thread is set back to user mode and the stack is restored to the user stack.

3.4 Memory Management

The following contains various topics regarding memory management.

3.4.1 Memory Heaps

Zephyr provides a collection of utilities that allow threads to dynamically allocate memory.

Synchronized Heap Allocator

Creating a Heap The simplest way to define a heap is statically, with the `K_HEAP_DEFINE` macro. This creates a static `k_heap` variable with a given name that manages a memory region of the specified size.

Heaps can also be created to manage arbitrary regions of application-controlled memory using `k_heap_init()`.

Allocating Memory Memory can be allocated from a heap using `k_heap_alloc()`, passing it the address of the heap object and the number of bytes desired. This functions similarly to standard C `malloc()`, returning a NULL pointer on an allocation failure.

The heap supports blocking operation, allowing threads to go to sleep until memory is available. The final argument is a `k_timeout_t` timeout value indicating how long the thread may sleep before returning, or else one of the constant timeout values `K_NO_WAIT` or `K_FOREVER`.

When a block with a specific alignment is required, `k_heap_aligned_alloc()` allocates a block whose start address is a multiple of a caller-supplied power-of-two alignment. `k_heap_calloc()` allocates and zero-initializes an array.

The size of an existing allocation can be changed with `k_heap_realloc()`, which returns a block of the requested size while preserving the contents up to the smaller of the old and new sizes. As with the standard C `realloc()`, the returned pointer may differ from the original.

Releasing Memory Memory allocated with `k_heap_alloc()` must be released using `k_heap_free()`. Similar to standard C `free()`, the pointer provided must be either a `NULL` value or a pointer previously returned by `k_heap_alloc()` for the same heap. Freeing a `NULL` value is defined to have no effect.

Enumerating Heaps All heaps defined statically with `K_HEAP_DEFINE` are placed in a dedicated linker section, which allows them to be enumerated at run time. `k_heap_array_get()` returns the address of the array of statically defined heaps and the number of entries it contains. This is primarily useful for instrumentation and diagnostics that need to inspect every heap in the system.

Low Level Heap Allocator

The underlying implementation of the `k_heap` abstraction is provided a data structure named `sys_heap`. This implements exactly the same allocation semantics, but provides no kernel synchronization tools. It is available for applications that want to manage their own blocks of memory in contexts (for example, userspace) where synchronization is unavailable or more complicated. Unlike `k_heap`, all calls to any `sys_heap` functions on a single heap must be serialized by the caller. Simultaneous use from separate threads is disallowed.

Implementation Internally, the `sys_heap` memory block is partitioned into "chunks" of 8 bytes. All allocations are made out of a contiguous region of chunks. The first chunk of every allocation or unused block is prefixed by a chunk header that stores the length of the chunk, the length of the next lower ("left") chunk in physical memory, a bit indicating whether the chunk is in use, and chunk-indexed link pointers to the previous and next chunk in a "free list" to which unused chunks are added.

The heap code takes reasonable care to avoid fragmentation. Free block lists are stored in "buckets" by their size, each bucket storing blocks within one power of two (i.e. a bucket for blocks of 3-4 chunks, another for 5-8, 9-16, etc...) this allows new allocations to be made from the smallest/most-fragmented blocks available. Also, as allocations are freed and added to the heap, they are automatically combined with adjacent free blocks to prevent fragmentation.

All metadata is stored at the beginning of the contiguous block of heap memory, including the variable-length list of bucket list heads (which depend on heap size). The only external memory required is the `sys_heap` structure itself.

The `sys_heap` functions are unsynchronized. Care must be taken by any users to prevent concurrent access. Only one context may be inside one of the API functions at a time.

The heap code takes care to present high performance and reliable latency. All sys_heap API functions are guaranteed to complete within constant time. On typical architectures, they will all complete within 1-200 cycles. One complexity is that the search of the minimum bucket size for an allocation (the set of free blocks that "might fit") has a compile-time upper bound of iterations to prevent unbounded list searches, at the expense of some fragmentation resistance. This CONFIG_SYS_HEAP_ALLOC_LOOPS value may be chosen by the user at build time, and defaults to a value of 3.

Multi-Heap Wrapper Utility

The sys_heap utility requires that all managed memory be in a single contiguous block. It is common for complicated microcontroller applications to have more complicated memory setups that they still want to manage dynamically as a "heap". For example, the memory might exist as separate discontiguous regions, different areas may have different cache, performance or power behavior, peripheral devices may only be able to perform DMA to certain regions, etc...

For those situations, Zephyr provides a sys_multi_heap utility. Effectively this is a simple wrapper around a set of one or more sys_heap objects. It should be initialized after its child heaps via sys_multi_heap_init(), after which each heap can be added to the managed set via sys_multi_heap_add_heap(). No destruction utility is provided; just as for sys_heap, applications that want to destroy a multi heap should simply ensure all allocated blocks are freed (or at least will never be used again) and repurpose the underlying memory for another usage.

It has a single pair of allocation entry points, sys_multi_heap_alloc() and sys_multi_heap_aligned_alloc(). These behave identically to the sys_heap functions with similar names, except that they also accept an opaque "configuration" parameter. This pointer is uninspected by the multi heap code itself; instead it is passed to a callback function provided at initialization time. This application-provided callback is responsible for doing the underlying allocation from one of the managed heaps, and may use the configuration parameter in any way it likes to make that decision.

For modifying the size of an allocated buffer (whether shrinking or enlarging it), you can use the sys_multi_heap_realloc() and sys_multi_heap_aligned_realloc() APIs. If the buffer cannot be enlarged on the heap where it currently resides, any of the eligible heaps specified by the configuration parameter may be used.

When unused, a multi heap may be freed via sys_multi_heap_free(). The application does not need to pass a configuration parameter. Memory allocated from any of the managed sys_heap objects may be freed with in the same way.

System Heap

The *system heap* is a predefined memory allocator that allows threads to dynamically allocate memory from a common memory region in a malloc()-like manner.

Only a single system heap is defined. Unlike other heaps or memory pools, the system heap cannot be directly referenced using its memory address.

The size of the system heap is configurable to arbitrary sizes, subject to space availability.

A thread can dynamically allocate a chunk of heap memory by calling k_malloc(). The address of the allocated chunk is guaranteed to be aligned on a multiple of pointer sizes. If a suitable chunk of heap memory cannot be found NULL is returned.

When the thread is finished with a chunk of heap memory it can release the chunk back to the system heap by calling k_free().

Defining the Heap Memory Pool The size of the heap memory pool is specified using the CONFIG_HEAP_MEM_POOL_SIZE configuration option.

By default, the heap memory pool size is zero bytes. This value instructs the kernel not to define the heap memory pool object. The maximum size is limited by the amount of available memory in the system. The project build will fail in the link stage if the size specified can not be supported.

In addition, each subsystem (board, driver, library, etc) can set a custom requirement by defining a Kconfig option with the prefix `HEAP_MEM_POOL_ADD_SIZE_` (this value is in bytes). If multiple subsystems specify custom values, the sum of these will be used as the minimum requirement. If the application tries to set a value that's less than the minimum value, this will be ignored and the minimum value will be used instead.

To force a smaller than minimum value to be used, the application may enable the `CONFIG_HEAP_MEM_POOL_IGNORE_MIN` option. This can be useful when optimizing the heap size and the minimum requirement can be more accurately determined for a specific application.

Allocating Memory A chunk of heap memory is allocated by calling `k_malloc()`.

The following code allocates a 200 byte chunk of heap memory, then fills it with zeros. A warning is issued if a suitable chunk is not obtained.

```
char *mem_ptr;

mem_ptr = k_malloc(200);
if (mem_ptr != NULL)) {
    memset(mem_ptr, 0, 200);
    ...
} else {
    printf("Memory not allocated");
}
```

Releasing Memory A chunk of heap memory is released by calling `k_free()`.

The following code allocates a 75 byte chunk of memory, then releases it once it is no longer needed.

```
char *mem_ptr;

mem_ptr = k_malloc(75);
... /* use memory block */
k_free(mem_ptr);
```

Suggested Uses Use the heap memory pool to dynamically allocate memory in a `malloc()`-like manner.

Configuration Options Related configuration options:

- `CONFIG_HEAP_MEM_POOL_SIZE`

API Reference

Heap APIs

Low Level Heap Allocator

Multi-Heap Wrapper

Heap listener

Heap Listener APIs

3.4.2 Shared Multi Heap

The shared multi-heap memory pool manager uses the multi-heap allocator to manage a set of reserved memory regions with different capabilities / attributes (cacheable, non-cacheable, etc...).

All the different regions can be added at run-time to the shared multi-heap pool providing an opaque "attribute" value (an integer or enum value) that can be used by drivers or applications to request memory with certain capabilities.

This framework is commonly used as follow:

1. At boot time some platform code initialize the shared multi-heap framework using `shared_multi_heap_pool_init()` and add the memory regions to the pool with `shared_multi_heap_add()`, possibly gathering the needed information for the regions from the DT.
2. Each memory region encoded in a `shared_multi_heap_region` structure. This structure is also carrying an opaque and user-defined integer value that is used to define the region capabilities (for example: cacheability, cpu affinity, etc...)

```
// Init the shared multi-heap pool
shared_multi_heap_pool_init()

// Fill the struct with the data for cacheable memory
struct shared_multi_heap_region cacheable_r0 = {
     .addr = addr_r0,
     .size = size_r0,
     .attr = SMH_REG_ATTR_CACHEABLE,
};

// Add the region to the pool
shared_multi_heap_add(&cacheable_r0, NULL);

// Add another cacheable region
struct shared_multi_heap_region cacheable_r1 = {
     .addr = addr_r1,
     .size = size_r1,
     .attr = SMH_REG_ATTR_CACHEABLE,
};

shared_multi_heap_add(&cacheable_r1, NULL);

// Add a non-cacheable region
struct shared_multi_heap_region non_cacheable_r2 = {
     .addr = addr_r2,
     .size = size_r2,
     .attr = SMH_REG_ATTR_NON_CACHEABLE,
};

shared_multi_heap_add(&non_cacheable_r2, NULL);
```

3. When a driver or application needs some dynamic memory with a certain capability, it can use `shared_multi_heap_alloc()` (or the aligned version) to request the memory by using the opaque parameter to select the correct set of attributes for the needed memory. The framework will take care of selecting the correct heap (thus memory region) to carve memory from, based on the opaque parameter and the runtime state of the heaps (available memory, heap state, etc...)

```
// Allocate 4K from cacheable memory
shared_multi_heap_alloc(SMH_REG_ATTR_CACHEABLE, 0x1000);

// Allocate 4K from non-cacheable memory
shared_multi_heap_alloc(SMH_REG_ATTR_NON_CACHEABLE, 0x1000);
```

Adding new attributes

The API does not enforce any attributes, but at least it defines the two most common ones: `SMH_REG_ATTR_CACHEABLE` and `SMH_REG_ATTR_NON_CACHEABLE`.

Shared multi-heap interface

3.4.3 Memory Slabs

A *memory slab* is a kernel object that allows memory blocks to be dynamically allocated from a designated memory region. All memory blocks in a memory slab have a single fixed size, allowing them to be allocated and released efficiently and avoiding memory fragmentation concerns.

- *Concepts*
  - *Internal Operation*
- *Implementation*
  - *Defining a Memory Slab*
  - *Allocating a Memory Block*
  - *Releasing a Memory Block*
  - *Querying Slab Usage*
- *Suggested Uses*
- *Configuration Options*
- *API Reference*

Concepts

Any number of memory slabs can be defined (limited only by available RAM). Each memory slab is referenced by its memory address.

A memory slab has the following key properties:

- The **block size** of each block, measured in bytes. It must be at least 4N bytes long on 32-bit platforms and at least 8N bytes long on 64-bit platforms, where N is greater than 0.
- The **number of blocks** available for allocation. It must be greater than zero.
- A **buffer** that provides the memory for the memory slab's blocks. It must be at least "block size" times "number of blocks" bytes long.

The memory slab's buffer must be aligned to an N-byte boundary, where N is a power of 2. N must be at least 4 on 32-bit platforms and at least 8 on 64-bit platforms. To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the block size must also be a multiple of N.

A memory slab must be initialized before it can be used. This marks all of its blocks as unused.

A thread that needs to use a memory block simply allocates it from a memory slab. When the thread finishes with a memory block, it must release the block back to the memory slab so the block can be reused.

If all the blocks are currently in use, a thread can optionally wait for one to become available. Any number of threads may wait on an empty memory slab simultaneously; when a memory block becomes available, it is given to the highest-priority thread that has waited the longest.

More than one memory slab can be defined, if needed. This allows for a memory slab with smaller blocks and others with larger-sized blocks. Alternatively, a memory pool object may be used.

Internal Operation A memory slab's buffer is an array of fixed-size blocks, with no wasted space between the blocks.

The memory slab keeps track of unallocated blocks using a linked list. 32-bit platforms use the first 4 bytes of each unused block to provide the necessary linkage, while 64-bit platforms use the first 8 bytes of each unused block.

Implementation

Defining a Memory Slab A memory slab is defined using a variable of type `k_mem_slab`. It must then be initialized by calling `k_mem_slab_init()`.

The following code defines and initializes a memory slab that has 6 blocks that are 400 bytes long, each of which is aligned to an 8-byte boundary.

```
struct k_mem_slab my_slab;
char __aligned(8) my_slab_buffer[6 * 400];

k_mem_slab_init(&my_slab, my_slab_buffer, 400, 6);
```

Alternatively, a memory slab can be defined and initialized at compile time by calling `K_MEM_SLAB_DEFINE`.

The following code has the same effect as the code segment above. Observe that the macro defines both the memory slab and its buffer.

```
K_MEM_SLAB_DEFINE(my_slab, 400, 6, 8);
```

Similarly, you can define a memory slab in private scope:

```
K_MEM_SLAB_DEFINE_STATIC(my_slab, 400, 6, 8);
```

Allocating a Memory Block A memory block is allocated by calling `k_mem_slab_alloc()`.

The following code builds on the example above, and waits up to 100 milliseconds for a memory block to become available, then fills it with zeroes. A warning is printed if a suitable block is not obtained.

```
char *block_ptr;

if (k_mem_slab_alloc(&my_slab, (void **)&block_ptr, K_MSEC(100)) == 0) {
    memset(block_ptr, 0, 400);
    ...
} else {
    printf("Memory allocation time-out");
}
```

Releasing a Memory Block A memory block is released by calling `k_mem_slab_free()`.

The following code builds on the example above, and allocates a memory block, then releases it once it is no longer needed.

```
char *block_ptr;

k_mem_slab_alloc(&my_slab, (void **)&block_ptr, K_FOREVER);
... /* use memory block pointed at by block_ptr */
k_mem_slab_free(&my_slab, (void *)block_ptr);
```

Querying Slab Usage The current utilization of a memory slab can be queried at run time. `k_mem_slab_num_used_get()` returns the number of blocks currently allocated, and `k_mem_slab_num_free_get()` returns the number of blocks still available. When `CONFIG_MEM_SLAB_TRACE_MAX_UTILIZATION` is enabled, `k_mem_slab_max_used_get()` reports the peak number of blocks that have been allocated simultaneously, and `k_mem_slab_runtime_stats_get()` returns these figures together in a `sys_memory_stats` structure.

Suggested Uses

Use a memory slab to allocate and free memory in fixed-size blocks.

Use memory slab blocks when sending large amounts of data from one thread to another, to avoid unnecessary copying of the data.

Configuration Options

Related configuration options:

- `CONFIG_MEM_SLAB_TRACE_MAX_UTILIZATION`

API Reference

Memory Slab APIs

3.4.4 Memory Blocks Allocator

The Memory Blocks Allocator allows memory blocks to be dynamically allocated from a designated memory region, where:

- All memory blocks have a single fixed size.
- Multiple blocks can be allocated or freed at the same time.
- A group of blocks allocated together may not be contiguous. This is useful for operations such as scatter-gather DMA transfers.
- Bookkeeping of allocated blocks is done outside of the associated buffer (unlike memory slab). This allows the buffer to reside in memory regions where these can be powered down to conserve energy.

- *Concepts*
  - *Internal Operation*
- *Memory Blocks Allocator*
- *Multi Memory Blocks Allocator Group*

- *Usage*
  - *Defining a Memory Blocks Allocator*
  - *Allocating Memory Blocks*
  - *Releasing a Memory Block*
  - *Using Multi Memory Blocks Allocator Group*
- *API Reference*

Concepts

Any number of Memory Blocks Allocator can be defined (limited only by available RAM). Each allocator is referenced by its memory address.

A memory blocks allocator has the following key properties:

- The **block size** of each block, measured in bytes. It must be at least 4N bytes long, where N is greater than 0.
- The **number of blocks** available for allocation. It must be greater than zero.
- A **buffer** that provides the backing store from which the memory blocks allocator will allocate blocks. It must be at least "block size" times "number of blocks" bytes long.
- A **blocks bitmap** to keep track of which block has been allocated.

The buffer must be aligned to an N-byte boundary, where N is a power of 2 larger than 2 (i.e. 4, 8, 16, ...). To ensure that all memory blocks in the buffer are similarly aligned to this boundary, the block size must also be a multiple of N.

Due to the use of internal bookkeeping structures and their creation, each memory blocks allocator must be declared and defined at compile time.

Internal Operation Each buffer associated with an allocator is an array of fixed-size blocks, with no wasted space between the blocks.

The memory blocks allocator keeps track of unallocated blocks using a bitmap.

Memory Blocks Allocator

Internally, the memory blocks allocator uses a bitmap to keep track of which blocks have been allocated. Each allocator, utilizing the `sys_bitarray` interface, gets memory blocks one by one from the backing buffer up to the requested number of blocks. All the metadata about an allocator is stored outside of the backing buffer. This allows the memory region of the backing buffer to be powered down to conserve energy, as the allocator code never touches the content of the buffer.

Multi Memory Blocks Allocator Group

The Multi Memory Blocks Allocator Group utility functions are used to conveniently manage a group of allocators. A custom function may be written to choose which allocator managed by this group should be used for block allocation.

An allocator group should be initialized at runtime via `sys_multi_mem_blocks_init()`. Each allocator can then be added via `sys_multi_mem_blocks_add_allocator()`.

To allocate memory blocks from group, `sys_multi_mem_blocks_alloc()` is called with an opaque "configuration" parameter. This parameter is passed directly to the allocator choice function so that an appropriate allocator can be chosen. After an allocator is chosen, memory blocks are allocated via `sys_mem_blocks_alloc()`.

Allocated memory blocks can be freed via `sys_multi_mem_blocks_free()`. The caller does not need to pass a configuration parameter. The allocator code matches the passed in memory addresses to find the correct allocator and then memory blocks are freed via `sys_mem_blocks_free()`.

Usage

Defining a Memory Blocks Allocator A memory blocks allocator is defined using a variable of type `sys_mem_blocks_t`. It needs to be defined and initialized at compile time by calling `SYS_MEM_BLOCKS_DEFINE`.

The following code defines and initializes a memory blocks allocator which has 4 blocks that are 64 bytes long, each of which is aligned to a 4-byte boundary:

```
SYS_MEM_BLOCKS_DEFINE(allocator, 64, 4, 4);
```

Similarly, you can define a memory blocks allocator in private scope:

```
SYS_MEM_BLOCKS_DEFINE_STATIC(static_allocator, 64, 4, 4);
```

A pre-defined buffer can also be provided to the allocator where the buffer can be placed separately. Note that the buffer **must** have its alignment specified when it is defined.

```
uint8_t __aligned(4) backing_buffer[64 * 4];
SYS_MEM_BLOCKS_DEFINE_WITH_EXT_BUF(allocator, 64, 4, backing_buffer);
```

Allocating Memory Blocks Memory blocks can be allocated by calling `sys_mem_blocks_alloc()`.

```
int ret;
uintptr_t blocks[2];

ret = sys_mem_blocks_alloc(allocator, 2, blocks);
```

If `ret == 0`, the array `blocks` will contain an array of memory addresses pointing to the allocated blocks.

Releasing a Memory Block Memory blocks are released by calling `sys_mem_blocks_free()`.

The following code builds on the example above which allocates 2 memory blocks, then releases them once they are no longer needed.

```
int ret;
uintptr_t blocks[2];

ret = sys_mem_blocks_alloc(allocator, 2, blocks);
... /* perform some operations on the allocated memory blocks */
ret = sys_mem_blocks_free(allocator, 2, blocks);
```

Using Multi Memory Blocks Allocator Group The following code demonstrates how to initialize an allocator group:

```
sys_mem_blocks_t *choice_fn(struct sys_multi_mem_blocks *group, void *cfg)
{
    ... /* choose which allocator in the group to use based on cfg */
}
```

(continues on next page)

(continued from previous page)

```
SYS_MEM_BLOCKS_DEFINE(allocator0, 64, 4, 4);
SYS_MEM_BLOCKS_DEFINE(allocator1, 64, 4, 4);

static sys_multi_mem_blocks_t alloc_group;

sys_multi_mem_blocks_init(&alloc_group, choice_fn);
sys_multi_mem_blocks_add_allocator(&alloc_group, &allocator0);
sys_multi_mem_blocks_add_allocator(&alloc_group, &allocator1);
```

To allocate and free memory blocks from the group:

```
int ret;
uintptr_t blocks[1];
size_t blk_size;

ret = sys_multi_mem_blocks_alloc(&alloc_group, UINT_TO_POINTER(0),
                                 1, blocks, &blk_size);

ret = sys_multi_mem_blocks_free(&alloc_group, 1, blocks);
```

API Reference

Memory Blocks APIs

3.4.5 Demand Paging

Demand paging provides a mechanism where data is only brought into physical memory as required by current execution context. The physical memory is conceptually divided in page-sized page frames as regions to hold data.

The Zephyr kernel image itself is always resident in physical memory and is never a candidate for eviction. Demand paging applies only to:

- anonymous memory mappings created at runtime via `k_mem_map()`, and
- memory placed in explicit on-demand linker sections via the `__ondemand_func` / `__ondemand_rodata` attributes when `CONFIG_LINKER_USE_ONDEMAND_SECTION` is enabled.

This is the same model used by every major operating system: the dispatch path for an exception or interrupt is never on a pageable page, so a fault during fault-handling is impossible by construction. Code or data added to an `__ondemand_*` section is the contributor's explicit opt-in to make that region pageable, and carries the responsibility of ensuring it is not reached from the page-fault handler's own execution path.

Note

Earlier versions of Zephyr also supported a selective-pinning scheme based on `__pinned_*` linker attributes that kept only the tagged subset of the kernel image resident and demand-paged the rest. That model was found to be both unsafe and/or very invasive: a CPU exception dispatch could target a thread's privileged stack on an evictable page, escalating to a double fault on x86 or a nested abort on ARM64 if that page had been evicted; and the contract that every byte on the fault-handler-reachable surface (scheduler, drivers, libc, locking primitives) be exhaustively tagged was impractical to establish once and unmaintainable thereafter. The `__pinned_*` attribute family, the corresponding Kconfig options (`LINKER_USE_PINNED_SECTION` / `LINKER_GENERIC_SECTIONS_PRESENT_AT_BOOT`) and the `K_*_PINNED_STACK_*` stack macros were removed in Zephyr 4.4. See GitHub #108773 for the full analysis.

- When the processor tries to access data and the data page exists in one of the page frames, the execution continues without any interruptions.
- When the processor tries to access the data page that does not exist in any page frames, a page fault occurs. The paging code then brings in the corresponding data page from backing store into physical memory if there is a free page frame. If there is no more free page frames, the eviction algorithm is invoked to select a data page to be paged out, thus freeing up a page frame for new data to be paged in. If this data page has been modified after it is first paged in, the data will be written back into the backing store. If no modifications is done or after written back into backing store, the data page is now considered paged out and the corresponding page frame is now free. The paging code then invokes the backing store to page in the data page corresponding to the location of the requested data. The backing store copies that data page into the free page frame. Now the data page is in physical memory and execution can continue.

There are functions where paging in and out can be invoked manually using `k_mem_page_in()` and `k_mem_page_out()`. `k_mem_page_in()` can be used to page in data pages in anticipation that they are required in the near future. This is used to minimize number of page faults as these data pages are already in physical memory, and thus minimizing latency. `k_mem_page_out()` can be used to page out data pages where they are not going to be accessed for a considerable amount of time. This frees up page frames so that the next page in can be executed faster as the paging code does not need to invoke the eviction algorithm.

A data region can also be **pinned** in physical memory using `k_mem_pin()`. This pages the region in if necessary and marks its page frames so that the eviction algorithm will never select them, guaranteeing that the region remains resident and that accesses to it never fault. This is a stronger form of `k_mem_page_in()` and is appropriate for latency- or safety-critical data that must always be available. A pinned region is later released with `k_mem_unpin()`, which marks the page frames as evictable again; unpinning does not itself evict the region, so it may be followed by `k_mem_page_out()` if immediate eviction is desired.

Terminology

Data Page
A data page is a page-sized region of data. It may exist in a page frame, or be paged out to some backing store. Its location can always be looked up in the CPU's page tables (or equivalent) by virtual address. The data type will always be `void *` or in some cases `uint8_t *` when doing pointer arithmetic.

Page Frame
A page frame is a page-sized physical memory region in RAM. It is a container where a data page may be placed. It is always referred to by physical address. Zephyr has a convention of using `uintptr_t` for physical addresses. For every page frame, a `struct k_mem_page_frame` is instantiated to store metadata. Flags for each page frame:

- `K_MEM_PAGE_FRAME_FREE` indicates a page frame is unused and on the list of free page frames. When this flag is set, none of the other flags are meaningful and they must not be modified.
- `K_MEM_PAGE_FRAME_PINNED` indicates a page frame is pinned in memory and should never be paged out.
- `K_MEM_PAGE_FRAME_RESERVED` indicates a physical page reserved by hardware and should not be used at all.
- `K_MEM_PAGE_FRAME_MAPPED` is set when a physical page is mapped to virtual memory address.
- `K_MEM_PAGE_FRAME_BUSY` indicates a page frame is currently involved in a page-in/out operation.
- `K_MEM_PAGE_FRAME_BACKED` indicates a page frame has a clean copy in the backing store.

K_MEM_SCRATCH_PAGE

The virtual address of a special page provided to the backing store to: * Copy a data page from `k_MEM_SCRATCH_PAGE` to the specified location; or, * Copy a data page from the provided location to `K_MEM_SCRATCH_PAGE`. This is used as an intermediate page for page in/out operations. This scratch needs to be mapped read/write for backing store code to access. However the data page itself may only be mapped as read-only in virtual address space. If this page is provided as-is to backing store, the data page must be re-mapped as read/write which has security implications as the data page is no longer read-only to other parts of the application.

Paging Statistics

Paging statistics can be obtained via various function calls when `CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM_NUM_BINS` is enabled:

- Overall statistics via `k_mem_paging_stats_get()`
- Per-thread statistics via `k_mem_paging_thread_stats_get()` if `CONFIG_DEMAND_PAGING_THREAD_STATS` is enabled
- Execution time histogram can be obtained when `CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM` is enabled, and `CONFIG_DEMAND_PAGING_TIMING_HISTOGRAM_NUM_BINS` is defined. Note that the timing is highly dependent on the architecture, SoC or board. It is highly recommended that `k_mem_paging_eviction_histogram_bounds[]` and `k_mem_paging_backing_store_histogram_bounds[]` be defined for a particular application.
  - Execution time histogram of eviction algorithm via `k_mem_paging_histogram_eviction_get()`
  - Execution time histogram of backing store doing page-in via `k_mem_paging_histogram_backing_store_page_in_get()`
  - Execution time histogram of backing store doing page-out via `k_mem_paging_histogram_backing_store_page_out_get()`

Eviction Algorithm

The eviction algorithm is used to determine which data page and its corresponding page frame can be paged out to free up a page frame for the next page in operation. There are four functions which are called from the kernel paging code:

- `k_mem_paging_eviction_init()` is called to initialize the eviction algorithm. This is called at `POST_KERNEL`.
- `k_mem_paging_eviction_add()` is called each time a data page becomes eligible for future eviction.
- `k_mem_paging_eviction_remove()` is called when a data page is no longer eligible for eviction. This may happen if the given data page becomes pinned, gets unmapped or is about to be evicted.
- `k_mem_paging_eviction_select()` is called to select a data page to evict. A function argument `dirty` is written to signal the caller whether the selected data page has been modified since it is first paged in. If the `dirty` bit is returned as set, the paging code signals to the backing store to write the data page back into storage (thus updating its content). The function returns a pointer to the page frame corresponding to the selected data page.

There is one additional function which is called by the architecture's memory management code to flag data pages when they trigger an access fault: `k_mem_paging_eviction_accessed()`. This is used by the LRU algorithm to requeue "used" pages.

Two eviction algorithms are currently available:

- An NRU (Not-Recently-Used) eviction algorithm has been implemented as a sample. This is a very simple algorithm which ranks data pages on whether they have been accessed and modified. The selection is based on this ranking.
- An LRU (Least-Recently-Used) eviction algorithm is also available. It is based on a sorted queue of data pages. The LRU code is more complex compared to the NRU code but also considerably more efficient. This is recommended for production use.

To implement a new eviction algorithm, `k_mem_paging_eviction_init()` and `k_mem_paging_eviction_select()` must be implemented. If `CONFIG_EVICTION_TRACKING` is enabled for an algorithm, these additional functions must also be implemented, `k_mem_paging_eviction_add()`, `k_mem_paging_eviction_remove()`, `k_mem_paging_eviction_accessed()`.

Backing Store

Backing store is responsible for paging in/out data page between their corresponding page frames and storage. These are the functions which must be implemented:

- `k_mem_paging_backing_store_init()` is called to initialized the backing store at `POST_KERNEL`.
- `k_mem_paging_backing_store_location_get()` is called to reserve a backing store location so a data page can be paged out. This `location` token is passed to `k_mem_paging_backing_store_page_out()` to perform actual page out operation.
- `k_mem_paging_backing_store_location_free()` is called to free a backing store location (the `location` token) which can then be used for subsequent page out operation.
- `k_mem_paging_backing_store_location_query()` is called to obtain the `location` token corresponding to storage content to be virtually mapped and paged-in on demand. Most useful with `CONFIG_DEMAND_MAPPING`.
- `k_mem_paging_backing_store_page_in()` copies a data page from the backing store location associated with the provided `location` token to the page pointed by `K_MEM_SCRATCH_PAGE`.
- `k_mem_paging_backing_store_page_out()` copies a data page from `K_MEM_SCRATCH_PAGE` to the backing store location associated with the provided `location` token.
- `k_mem_paging_backing_store_page_finalize()` is invoked after `k_mem_paging_backing_store_page_in()` so that the page frame struct may be updated for internal accounting. This can be a no-op.

To implement a new backing store, the functions mentioned above must be implemented. `k_mem_paging_backing_store_page_finalize()` can be an empty function if so desired.

API Reference

Demand Paging APIs

Related code samples

- Demand pagingLeverage demand paging to deal with code bigger than available RAM.

Eviction Algorithm APIs

Eviction Algorithm APIs

Backing Store APIs

Backing Store APIs

3.4.6 Virtual Memory

Virtual memory (VM) in Zephyr provides developers with the ability to fine tune access to memory. To utilize virtual memory, the platform must support Memory Management Unit (MMU) and it must be enabled in the build. Due to the target of Zephyr mainly being embedded systems, virtual memory support in Zephyr differs a bit from that in traditional operating systems:

Mapping of Kernel Image
Default is to do 1:1 mapping for the kernel image (including code and data) between physical and virtual memory address spaces, if demand paging is not enabled. Deviation from this requires careful manipulation of linker script.

Secondary Storage
Basic virtual memory support does not utilize secondary storage to extend usable memory. The maximum usable memory is the same as the physical memory.

- *Demand Paging* enables utilizing secondary storage as a backing store for virtual memory, thus allowing larger usable memory than the available physical memory. Note that demand paging needs to be explicitly enabled.
- Although the virtual memory space can be larger than physical memory space, without enabling demand paging, all virtually mapped memory must be backed by physical memory.

Kconfigs

Required These are the Kconfigs that need to be enabled or defined for kernel to support virtual memory.

- `CONFIG_MMU`: must be enabled for virtual memory support in kernel.
- `CONFIG_MMU_PAGE_SIZE`: size of a memory page. Default is 4KB.
- `CONFIG_KERNEL_VM_BASE`: base address of virtual address space.
- `CONFIG_KERNEL_VM_SIZE`: size of virtual address space. Default is 8MB.
- `CONFIG_KERNEL_VM_OFFSET`: kernel image starts at this offset from `CONFIG_KERNEL_VM_BASE`.

Optional

- `CONFIG_KERNEL_DIRECT_MAP`: permits 1:1 mappings between virtual and physical addresses, instead of kernel choosing addresses within the virtual address space. This is useful for mapping device MMIO regions for more precise access control.

Memory Map Overview

This is an overview of the memory map of the virtual memory address space. Note that the `Z_*` macros, which are used in code, may have different meanings depending on architecture and Kconfigs, which will be explained below.

```
+--------------+ <- K_MEM_VIRT_RAM_START
| Undefined VM | <- architecture specific reserved area
+--------------+ <- K_MEM_KERNEL_VIRT_START
| Mapping for  |
| main kernel  |
| image        |
|              |
|              |
+--------------+ <- K_MEM_VM_FREE_START
|              |
| Unused,      |
```

(continues on next page)

(continued from previous page)

```
| Available VM |
|              |
|..............| <- grows downward as more mappings are made
| Mapping      |
+--------------+
| Mapping      |
+--------------+
| ...          |
+--------------+
| Mapping      |
+--------------+ <- memory mappings start here
| Reserved     | <- special purpose virtual page(s) of size K_MEM_VM_RESERVED
+--------------+ <- K_MEM_VIRT_RAM_END
```

- `K_MEM_VIRT_RAM_START` is the beginning of the virtual memory address space. This needs to be page aligned. Currently, it is the same as `CONFIG_KERNEL_VM_BASE`.
- `K_MEM_VIRT_RAM_SIZE` is the size of the virtual memory address space. This needs to be page aligned. Currently, it is the same as `CONFIG_KERNEL_VM_SIZE`.
- `K_MEM_VIRT_RAM_END` is simply (`K_MEM_VIRT_RAM_START` + `K_MEM_VIRT_RAM_SIZE`).
- `K_MEM_KERNEL_VIRT_START` is the same as `z_mapped_start` specified in the linker script. This is the virtual address of the beginning of the kernel image at boot time.
- `K_MEM_KERNEL_VIRT_END` is the same as `z_mapped_end` specified in the linker script. This is the virtual address of the end of the kernel image at boot time.
- `K_MEM_VM_FREE_START` is the beginning of the virtual address space where addresses can be allocated for memory mapping. This depends on whether `CONFIG_ARCH_MAPS_ALL_RAM` is enabled.
  - If it is enabled, which means all physical memory are mapped in virtual memory address space, and it is the same as (`DT_CHOSEN_SRAM_ADDR` + `DT_CHOSEN_SRAM_SIZE`).
  - If it is disabled, `K_MEM_VM_FREE_START` is the same `K_MEM_KERNEL_VIRT_END` which is the end of the kernel image.
- `K_MEM_VM_RESERVED` is an area reserved to support kernel functions. For example, some addresses are reserved to support demand paging.

Virtual Memory Mappings

Setting up Mappings at Boot In general, most supported architectures set up the memory mappings at boot as following:

- `.text` section is read-only and executable. It is accessible in both kernel and user modes.
- `.rodata` section is read-only and non-executable. It is accessible in both kernel and user modes.
- Other kernel sections, such as `.data`, `.bss` and `.noinit`, are read-write and non-executable. They are only accessible in kernel mode.
  - Stacks for user mode threads are automatically granted read-write access to their corresponding user mode threads during thread creation.
  - Global variables, by default, are not accessible to user mode threads. Refer to *Memory Domains and Partitions* on how to use global variables in user mode threads, and on how to share data between user mode threads.

Caching modes for these mappings are architecture specific. They can be none, write-back, or write-through.

Note that SoCs have their own additional mappings required to boot where these mappings are defined under their own SoC configurations. These mappings usually include device MMIO regions needed to setup the hardware.

Mapping Anonymous Memory The unused physical memory can be mapped in virtual address space on demand. This is conceptually similar to memory allocation from heap, but these mappings must be aligned on page size and have finer access control.

- `k_mem_map()` can be used to map unused physical memory:
  - The requested size must be multiple of page size.
  - The address returned is inside the virtual address space between `K_MEM_VM_FREE_START` and `K_MEM_VIRT_RAM_END`.
  - The mapped region is not guaranteed to be physically contiguous in memory.
  - Guard pages immediately before and after the mapped virtual region are automatically allocated to catch access issue due to buffer underrun or overrun.
- The mapped region can be unmapped (i.e. freed) via `k_mem_unmap()`:
  - Caution must be exercised to give the pass the same region size to both `k_mem_map()` and `k_mem_unmap()`. The unmapping function does not check if it is a valid mapped region before unmapping.

API Reference

Kernel Memory Management

3.5 Data Structures

Zephyr provides a library of common general purpose data structures used within the kernel, but useful by application code in general. These include list and balanced tree structures for storing ordered data, and a ring buffer for managing "byte stream" data in a clean way.

Note that in general, the collections are implemented as "intrusive" data structures. The "node" data is the only struct used by the library code, and it does not store a pointer or other metadata to indicate what user data is "owned" by that node. Instead, the expectation is that the node will be itself embedded within a user-defined struct. Macros are provided to retrieve a user struct address from the embedded node pointer in a clean way. The purpose behind this design is to allow the collections to be used in contexts where dynamic allocation is disallowed (i.e. there is no need to allocate node objects because the memory is provided by the user).

Note also that these libraries are generally unsynchronized; access to them is not threadsafe by default. These are data structures, not synchronization primitives. The expectation is that any locking needed will be provided by the user. Some of the provided data structures are thread safe in specific usage scenarios (see *Single Producer Single Consumer Lock Free Queue* and *Multi Producer Single Consumer Lock Free Queue*).

3.5.1 Single-linked List

Zephyr provides a `sys_slist_t` type for storing simple singly-linked list data (i.e. data where each list element stores a pointer to the next element, but not the previous one). This supports constant-time access to the first (head) and last (tail) elements of the list, insertion before the head and after the tail of the list and constant time removal of the head. Removal of subsequent nodes requires access to the "previous" pointer and thus can only be performed in linear time by searching the list.

The sys_slist_t struct may be instantiated by the user in any accessible memory. It should be initialized with either sys_slist_init() or by static assignment from SYS_SLIST_STATIC_INIT before use. Its interior fields are opaque and should not be accessed by user code.

The end nodes of a list may be retrieved with sys_slist_peek_head() and sys_slist_peek_tail(), which will return NULL if the list is empty, otherwise a pointer to a sys_snode_t struct.

The sys_snode_t struct represents the data to be inserted. In general, it is expected to be allocated/controlled by the user, usually embedded within a struct which is to be added to the list. The container struct pointer may be retrieved from a list node using SYS_SLIST_CONTAINER, passing it the struct name of the containing struct and the field name of the node. Internally, the sys_snode_t struct contains only a next pointer, which may be accessed with sys_slist_peek_next().

Lists may be modified by adding a single node at the head or tail with sys_slist_prepend() and sys_slist_append(). They may also have a node added to an interior point with sys_slist_insert(), which inserts a new node after an existing one. Similarly sys_slist_remove() will remove a node given a pointer to its predecessor. These operations are all constant time.

Convenience routines exist for more complicated modifications to a list. sys_slist_merge_slist() will append an entire list to an existing one. sys_slist_append_list() will append a bounded subset of an existing list in constant time. And sys_slist_find_and_remove() will search a list (in linear time) for a given node and remove it if present.

Finally the slist implementation provides a set of "for each" macros that allows for iterating over a list in a natural way without needing to manually traverse the next pointers. SYS_SLIST_FOR_EACH_NODE will enumerate every node in a list given a local variable to store the node pointer. SYS_SLIST_FOR_EACH_NODE_SAFE behaves similarly, but has a more complicated implementation that requires an extra scratch variable for storage and allows the user to delete the iterated node during the iteration. Each of those macros also exists in a "container" variant (SYS_SLIST_FOR_EACH_CONTAINER and SYS_SLIST_FOR_EACH_CONTAINER_SAFE) which assigns a local variable of a type that matches the user's container struct and not the node struct, performing the required offsets internally. And SYS_SLIST_ITERATE_FROM_NODE exists to allow for enumerating a node and all its successors only, without inspecting the earlier part of the list.

Single-linked List Internals

The slist code is designed to be minimal and conventional. Internally, a sys_slist_t struct is nothing more than a pair of "head" and "tail" pointer fields. And a sys_snode_t stores only a single "next" pointer.

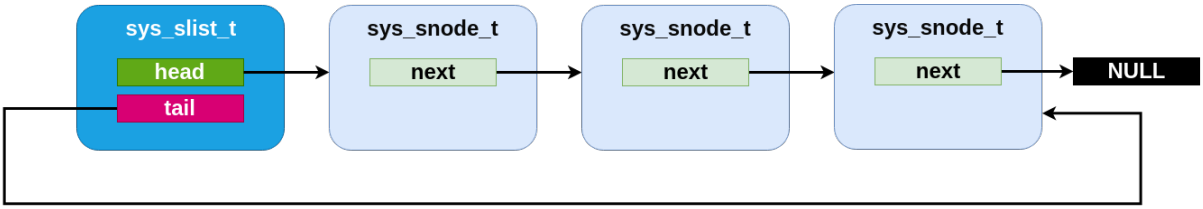


Fig. 4: An slist containing three elements.

The specific implementation of the list code, however, is done with an internal "Z_GENLIST" template API which allows for extracting those fields from arbitrary structures and emits an arbitrarily named set of functions. This allows for implementing more complicated single-linked list variants using the same basic primitives. The genlist implementer is responsible for a custom implementation of the primitive operations only: an "init" step for each struct, and a "get" and "set" primitives for each of head, tail and next pointers on their relevant structs. These inline functions are passed as parameters to the genlist macro expansion.

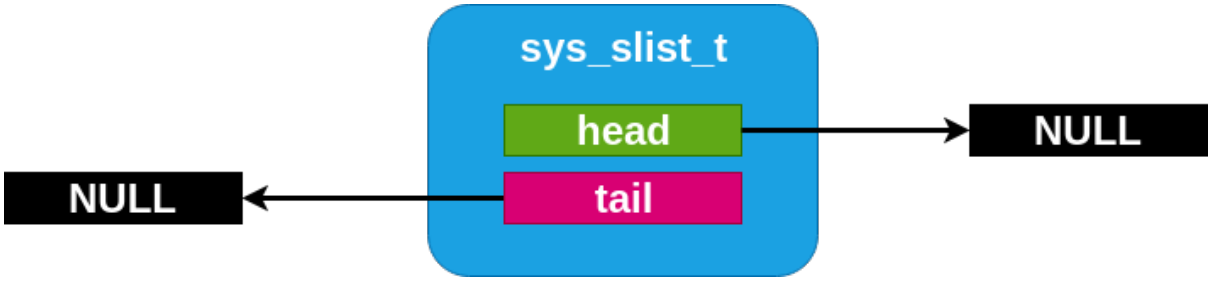


Fig. 5: An empty slist

Only one such variant, sflist, exists in Zephyr at the moment.

Flagged List

The `sys_sflist_t` is implemented using the described genlist template API. With the exception of symbol naming ("sflist" instead of "slist") and the additional API described next, it operates in all ways identically to the slist API.

It adds the ability to associate exactly two bits of user defined "flags" with each list node. These can be accessed and modified with `sys_sfnode_flags_get()` and `sys_sfnode_flags_set()`. Internally, the flags are stored unioned with the bottom bits of the next pointer and incur no SRAM storage overhead when compared with the simpler slist code.

Single-linked List API Reference

Single-linked list

Flagged List API Reference

Flagged Single-linked list

3.5.2 Double-linked List

Similar to the single-linked list in many respects, Zephyr includes a double-linked implementation. This provides the same algorithmic behavior for all the existing slist operations, but also allows for constant-time removal and insertion (at all points: before or after the head, tail or any internal node). To do this, the list stores two pointers per node, and thus has somewhat higher runtime code and memory space needs.

A `sys_dlist_t` struct may be instantiated by the user in any accessible memory. It must be initialized with `sys_dlist_init()` or `SYS_DLIST_STATIC_INIT` before use. The `sys_dnode_t` struct is expected to be provided by the user for any nodes added to the list (typically embedded within the struct to be tracked, as described above). It must be initialized in zeroed/bss memory or with `sys_dnode_init()` before use.

Primitive operations may retrieve the head/tail of a list and the next/prev pointers of a node with `sys_dlist_peek_head()`, `sys_dlist_peek_tail()`, `sys_dlist_peek_next()` and `sys_dlist_peek_prev()`. These can all return NULL where appropriate (i.e. for empty lists, or nodes at the endpoints of the list).

A dlist can be modified in constant time by removing a node with `sys_dlist_remove()`, by adding a node to the head or tail of a list with `sys_dlist_prepend()` and `sys_dlist_append()`, or by inserting a node before an existing node with `sys_dlist_insert()`.

As for slist, each node in a dlist can be processed in a natural code block style using `SYS_DLIST_FOR_EACH_NODE`. This macro also exists in a "FROM_NODE" form which allows for iterating from a known starting point, a "SAFE" variant that allows for removing the node being

inspected within the code block, a "CONTAINER" style that provides the pointer to a containing struct instead of the raw node, and a "CONTAINER_SAFE" variant that provides both properties.

Convenience utilities provided by dlist include `sys_dlist_insert_at()`, which inserts a node that linearly searches through a list to find the right insertion point, which is provided by the user as a C callback function pointer, and `sys_dnode_is_linked()`, which will affirmatively return whether or not a node is currently linked into a dlist or not (via an implementation that has zero overhead vs. the normal list processing).

Double-linked List Internals

Internally, the dlist implementation is minimal: the `sys_dlist_t` struct contains "head" and "tail" pointer fields, the `sys_dnode_t` contains "prev" and "next" pointers, and no other data is stored. But in practice the two structs are internally identical, and the list struct is inserted as a node into the list itself. This allows for a very clean symmetry of operations:

- An empty list has backpointers to itself in the list struct, which can be trivially detected.
- The head and tail of the list can be detected by comparing the prev/next pointers of a node vs. the list struct address.
- An insertion or deletion never needs to check for the special case of inserting at the head or tail. There are never any NULL pointers within the list to be avoided. Exactly the same operations are run, without tests or branches, for all list modification primitives.

Effectively, a dlist of N nodes can be thought of as a "ring" of "N+1" nodes, where one node represents the list tracking struct.

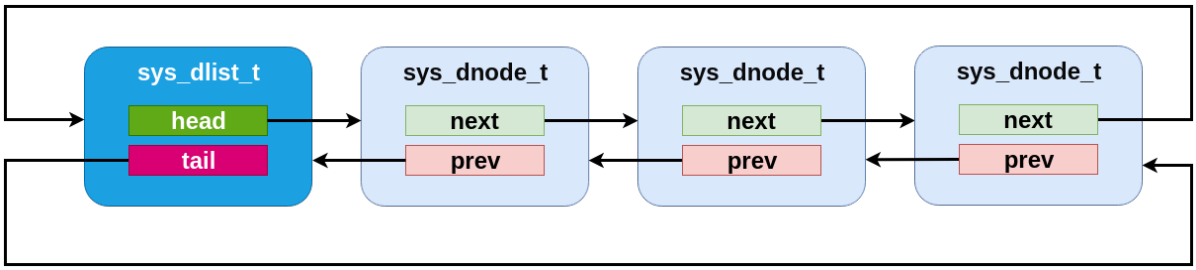


Fig. 6: A dlist containing three elements. Note that the list struct appears as a fourth "element" in the list.

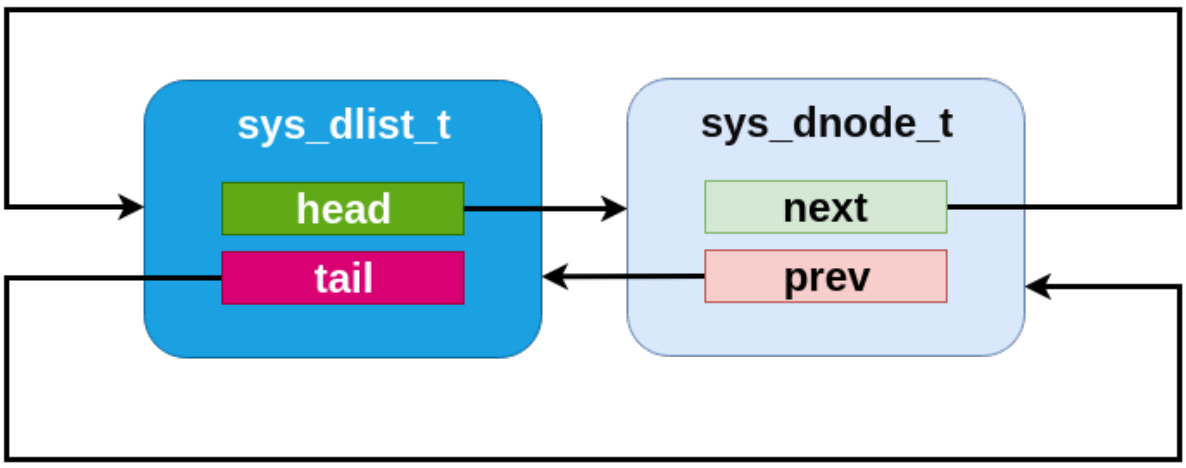


Fig. 7: An dlist containing just one element.

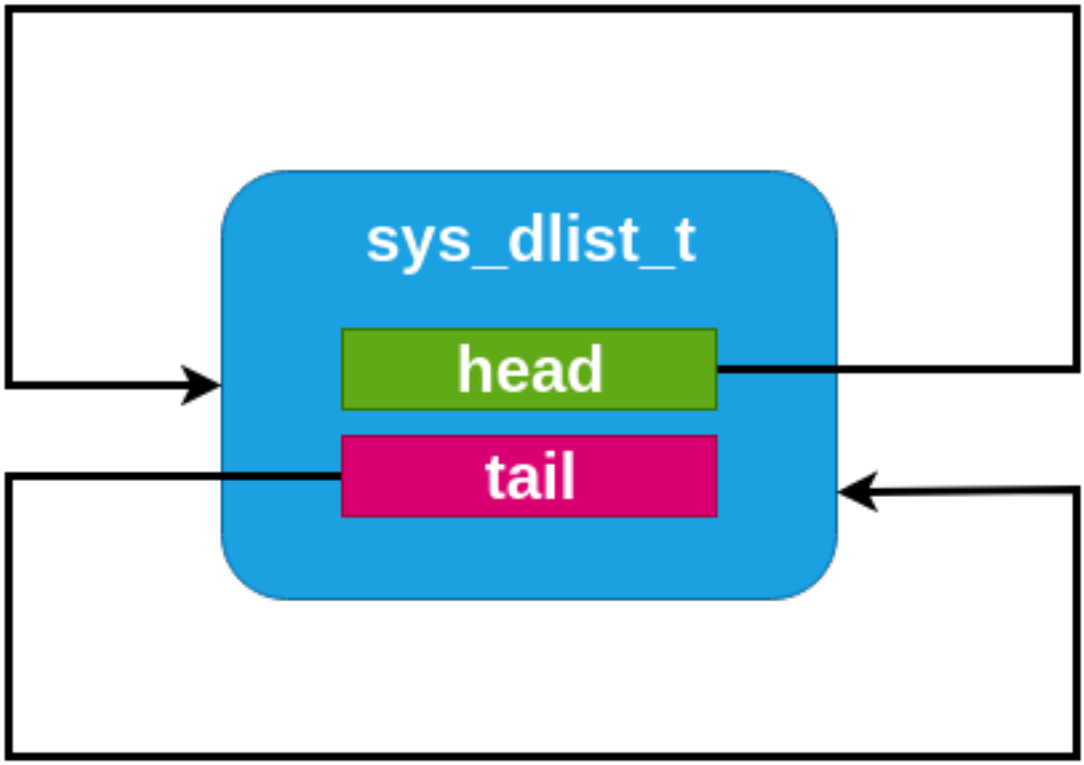


Fig. 8: An empty dlist.

Doubly-linked List API Reference

Doubly-linked list

3.5.3 Multi Producer Single Consumer Packet Buffer

A *Multi Producer Single Consumer Packet Buffer (MPSC_PBUF)* is a circular buffer, whose contents are stored in first-in-first-out order. Variable size packets are stored in the buffer. Packet buffer works under assumption that there is a single context that consumes the data. However, it is possible that another context may interfere to flush the data and never come back (panic case). Packet is produced in two steps: first requested amount of data is allocated, producer fills the data and commits it. Consuming a packet is also performed in two steps: consumer claims the packet, gets pointer to it and length and later on packet is freed. This approach reduces memory copying.

A *MPSC Packet Buffer* has the following key properties:

- Allocate, commit scheme used for packet producing.
- Claim, free scheme used for packet consuming.
- Allocator ensures that contiguous memory of requested length is allocated.
- Following policies can be applied when requested space cannot be allocated:
  - **Overwrite** - oldest entries are dropped until requested amount of memory can be allocated. For each dropped packet user callback is called.
  - **No overwrite** - When requested amount of space cannot be allocated, allocation fails.
- Dedicated, optimized API for storing short packets.
- Allocation with timeout.

Internals

Each packet in the buffer contains MPSC_PBUF specific header which is used for internal management. Header consists of 2 bit flags. In order to optimize memory usage, header can be added on top of the user header using MPSC_PBUF_HDR and remaining bits in the first word can be application specific. Header consists of following flags:

- valid - bit set to one when packet contains valid user packet
- busy - bit set when packet is being consumed (claimed but not free)

Header state:

valid	busy	description
0	0	space is free
1	0	valid packet
1	1	claimed valid packet
0	1	internal skip packet

Packet buffer space contains free space, valid user packets and internal skip packets. Internal skip packets indicates padding, e.g. at the end of the buffer.

Allocation Using pairs for read and write indexes, available space is determined. If space can be allocated, temporary write index is moved and pointer to a space within buffer is returned. Packet header is reset. If allocation required wrapping of the write index, a skip packet is added to the end of buffer. If space cannot be allocated and overwrite is disabled then NULL pointer is returned or context blocks if allocation was with timeout.

Allocation with overwrite If overwrite is enabled, oldest packets are dropped until requested amount of space can be allocated. When packets are dropped busy flag is checked in the header to ensure that currently consumed packet is not overwritten. In that case, skip packet is added before busy packet and packets following the busy packet are dropped. When busy packet is being freed, such situation is detected and packet is converted to skip packet to avoid double processing.

Usage

Packet header definition Packet header details can be found in include/zephyr/sys/mpsc_packet.h. API functions can be found in include/zephyr/sys/mpsc_pbuf.h. Headers are split to avoid include spam when declaring the packet.

User header structure must start with internal header:

```
#include <zephyr/sys/mpsc_packet.h>

struct foo_header {
        MPSC_PBUF_HDR;
        uint32_t length: 32 - MPSC_PBUF_HDR_BITS;
};
```

Packet buffer configuration Configuration structure contains buffer details, configuration flags and callbacks. Following callbacks are used by the packet buffer:

- Drop notification - callback called whenever a packet is dropped due to overwrite.
- Get packet length - callback to determine packet length

Packet producing Standard, two step method:

```
foo_packet *packet = mpsc_pbuf_alloc(buffer, len, K_NO_WAIT);

fill_data(packet);

mpsc_pbuf_commit(buffer, packet);
```

Performance optimized storing of small packets:

- 32 bit word packet
- 32 bit word with pointer packet

Note that since packets are written by value, they should already contain `valid` bit set in the header.

```
mpsc_pbuf_put_word(buffer, data);
mpsc_pbuf_put_word_ext(buffer, data, ptr);
```

Packet consuming Two step method:

```
foo_packet *packet = mpsc_pbuf_claim(buffer);

process(packet);

mpsc_pbuf_free(buffer, packet);
```

3.5.4 Single Producer Single Consumer Packet Buffer

A *Single Producer Single Consumer Packet Buffer (SPSC_PBUF)* is a circular buffer, whose contents are stored in first-in-first-out order. Variable size packets are stored in the buffer. Packet buffer works under assumption that there is a single context that produces packets and a single context that consumes the data.

Implementation is focused on performance and memory footprint.

Packets are added to the buffer using `spsc_pbuf_write()` which copies a data into the buffer. If the buffer is full error is returned.

Packets are copied out of the buffer using `spsc_pbuf_read()`.

3.5.5 Balanced Red/Black Tree

For circumstances where sorted containers may become large at runtime, a list becomes problematic due to algorithmic costs of searching it. For these situations, Zephyr provides a balanced tree implementation which has runtimes on search and removal operations bounded at O(log2(N)) for a tree of size N. This is implemented using a conventional red/black tree as described by multiple academic sources.

The `rbtree` tracking struct for a rbtree may be initialized anywhere in user accessible memory. It should contain only zero bits before first use. No specific initialization API is needed or required.

Unlike a list, where position is explicit, the ordering of nodes within an rbtree must be provided as a predicate function by the user. A function of type `rb_lessthan_t()` should be assigned to the `lessthan_fn` field of the `rbtree` struct before any tree operations are attempted. This function should, as its name suggests, return a boolean True value if the first node argument is "less than" the second in the ordering desired by the tree. Note that "equal" is not allowed, nodes within a tree must have a single fixed order for the algorithm to work correctly.

As with the slist and dlist containers, nodes within an rbtree are represented as a `rbnode` structure which exists in user-managed memory, typically embedded within the data structure being

tracked in the tree. Unlike the list code, the data within an rbnode is entirely opaque. It is not possible for the user to extract the binary tree topology and "manually" traverse the tree as it is for a list.

Nodes can be inserted into a tree with `rb_insert()` and removed with `rb_remove()`. Access to the "first" and "last" nodes within a tree (in the sense of the order defined by the comparison function) is provided by `rb_get_min()` and `rb_get_max()`. There is also a predicate, `rb_contains()`, which returns a boolean True if the provided node pointer exists as an element within the tree. As described above, all of these routines are guaranteed to have at most log time complexity in the size of the tree.

There are two mechanisms provided for enumerating all elements in an rbtree. The first, `rb_walk()`, is a simple callback implementation where the caller specifies a C function pointer and an untyped argument to be passed to it, and the tree code calls that function for each node in order. This has the advantage of a very simple implementation, at the cost of a somewhat more cumbersome API for the user (not unlike ISO C's `bsearch()` routine). It is a recursive implementation, however, and is thus not always available in environments that forbid the use of unbounded stack techniques like recursion.

There is also a `RB_FOR_EACH` iterator provided, which, like the similar APIs for the lists, works to iterate over a list in a more natural way, using a nested code block instead of a callback. It is also nonrecursive, though it requires log-sized space on the stack by default (however, this can be configured to use a fixed/maximally size buffer instead where needed to avoid the dynamic allocation). As with the lists, this is also available in a `RB_FOR_EACH_CONTAINER` variant which enumerates using a pointer to a container field and not the raw node pointer.

Tree Internals

As described, the Zephyr rbtree implementation is a conventional red/black tree as described pervasively in academic sources. Low level details about the algorithm are out of scope for this document, as they match existing conventions. This discussion will be limited to details notable or specific to the Zephyr implementation.

The core invariant guaranteed by the tree is that the path from the root of the tree to any leaf is no more than twice as long as the path to any other leaf. This is achieved by associating one bit of "color" with each node, either red or black, and enforcing a rule that no red child can be a child of another red child (i.e. that the number of black nodes on any path to the root must be the same, and that no more than that number of "extra" red nodes may be present). This rule is enforced by a set of rotation rules used to "fix" trees following modification.

These rotations are conceptually implemented on top of a primitive that "swaps" the position of one node with another in the list. Typical implementations effect this by simply swapping the nodes internal "data" pointers, but because the Zephyr `rbnode` is intrusive, that cannot work. Zephyr must include somewhat more elaborate code to handle the edge cases (for example, one swapped node can be the root, or the two may already be parent/child).

The `rbnode` struct for a Zephyr rbtree contains only two pointers, representing the "left", and "right" children of a node within the binary tree. Traversal of a tree for rebalancing following modification, however, routinely requires the ability to iterate "upwards" from a node as well. It is very common for red/black trees in the industry to store a third "parent" pointer for this purpose. Zephyr avoids this requirement by building a "stack" of node pointers locally as it traverses downward through the tree and updating it appropriately as modifications are made. So a Zephyr rbtree can be implemented with no more runtime storage overhead than a dlist.

These properties, of a balanced tree data structure that works with only two pointers of data per node and that works without any need for a memory allocation API, are quite rare in the industry and are somewhat unique to Zephyr.

Red/Black Tree API Reference

Balanced Red/Black Tree

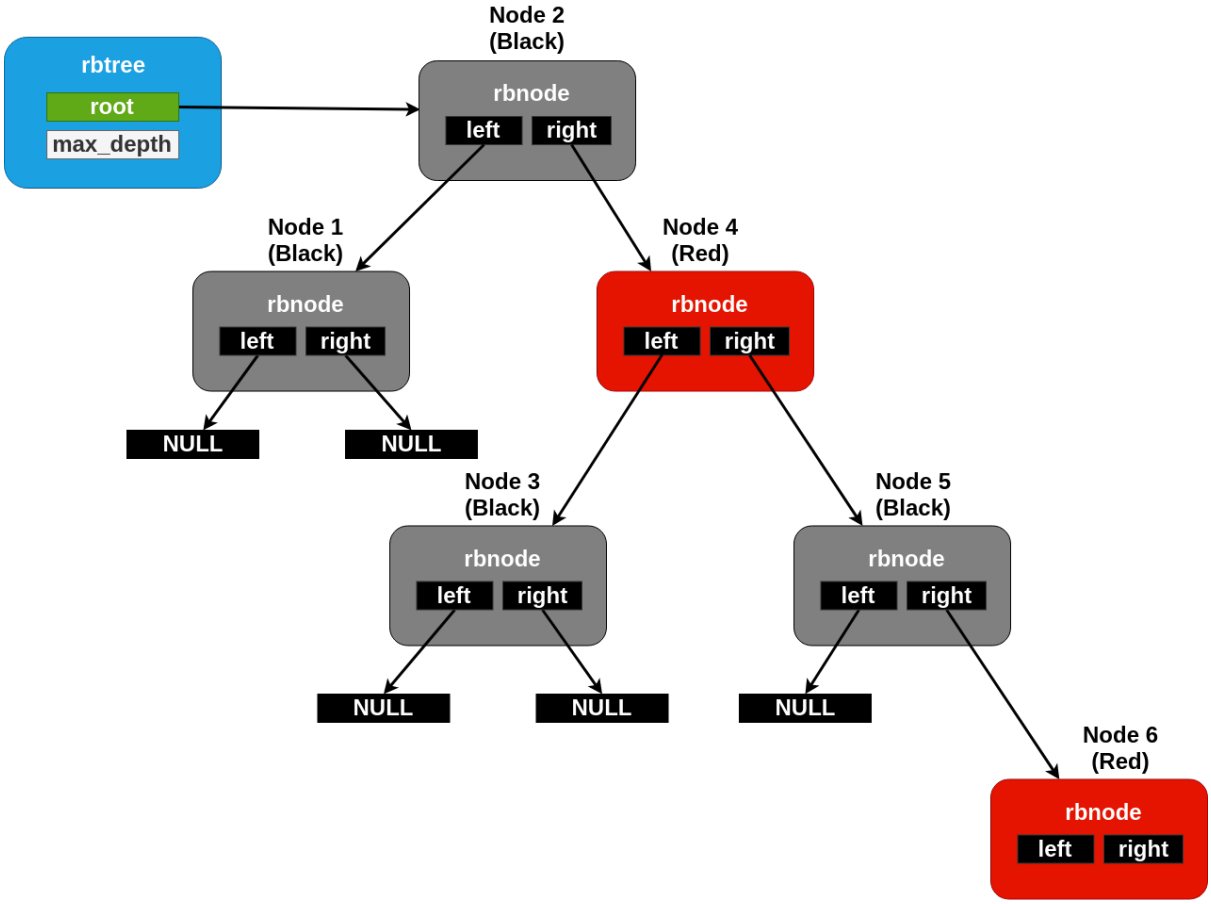


Fig. 9: A maximally unbalanced rbtree with a black height of two. No more nodes can be added underneath the rightmost node without rebalancing.

3.5.6 Ring Buffers

A *ring buffer* is a circular buffer, whose contents are stored in first-in-first-out order.

For circumstances where an application needs to implement asynchronous "streaming" copying of data, Zephyr provides a `struct ring_buf` abstraction to manage copies of such data in and out of a shared buffer of memory.

- *Concepts*
  - *Instantiation and Usage*
  - *Concurrency*
  - *Internal Operation*
- *Implementation*
  - *Defining a Ring Buffer*
  - *Enqueuing Data*
  - *Retrieving Data*
- *Configuration Options*
- *API Reference*

Concepts

Any number of ring buffers can be defined (limited only by available RAM). Each ring buffer is referenced by its memory address.

A ring buffer has the following key properties:

- A **data buffer** of bytes. The data buffer contains the raw bytes that have been added to the ring buffer but not yet removed.
- A **data buffer size**, measured in bytes. This governs the maximum amount of data the ring buffer can hold.

A ring buffer must be initialized before it can be used. This sets its data buffer to empty.

A `struct ring_buf` may be placed anywhere in user-accessible memory, and must be initialized with `ring_buf_init()` before use. This must be provided a region of user-controlled memory for use as the buffer itself. Note carefully that the units of the size of the buffer passed change (either bytes or words) depending on how the ring buffer will be used later. Macros for combining these steps in a single static declaration exist for convenience. `RING_BUF_DECLARE` will declare and statically initialize a ring buffer with a specified byte count, where

"Bytes" data may be copied into the ring buffer using `ring_buf_put()`, passing a data pointer and byte count. These bytes will be copied into the buffer in order, as many as will fit in the allocated buffer. The total number of bytes copied (which may be fewer than provided) will be returned. Likewise `ring_buf_get()` will copy bytes out of the ring buffer in the order that they were written, into a user-provided buffer, returning the number of bytes that were transferred.

To avoid multiply-copied-data situations, a zero-copy "pointer" API exists. `ring_buf_put_ptr()` returns a pointer to memory internal to the ring buffer that can be used to receive bytes, along with the size of the contiguous internal region available for writing (which may be smaller than the total free space if the region wraps). The user can then write data directly into that region (e.g. via DMA) without assembling all the bytes in a single region first. When complete, `ring_buf_commit()` is used to signal the buffer that the transfer is complete, passing the number of bytes actually transferred. At this point a new transfer can be initiated. Similarly, `ring_buf_get_ptr()` returns a pointer to internal ring buffer data from which the user can read

without making a verbatim copy, and `ring_buf_consume()` signals the buffer with how many bytes have been consumed and allows for a new transfer to begin.

The legacy `ring_buf_put_claim()` / `ring_buf_put_finish()` and `ring_buf_get_claim()` / `ring_buf_get_finish()` APIs provide similar functionality but reserve the claimed region until finished, which incurs additional code size and runtime cost. These APIs are deprecated and will be removed in a future release; new code should use the `_ptr` / `_commit` / `_consume` variants described above.

The user can manage the capacity of a ring buffer without modifying it using either `ring_buf_space_get()` which returns the number of free bytes, or by testing the `ring_buf_is_empty()` predicate.

Finally, a `ring_buf_reset()` call exists to immediately empty a ring buffer, discarding the tracking of any bytes already written to the buffer. It does not modify the memory contents of the buffer itself, however.

Instantiation and Usage A ring buffer instance is declared using `RING_BUF_DECLARE()` and accessed using: `ring_buf_put_ptr()`, `ring_buf_commit()`, `ring_buf_get_ptr()`, `ring_buf_consume()`, `ring_buf_put()` and `ring_buf_get()`.

Data can be copied into the ring buffer (see `ring_buf_put()`) or ring buffer memory can be used directly by the user. In the latter case, the operation is split into three stages:

1. Accessing the ring buffers internal buffer (`ring_buf_put_ptr()`) to get a pointer to the next location where data can be written, and the amount of contiguous space available at that location.
2. writing the data by the user (e.g. buffer written by DMA).
3. indicating the amount of data written to the provided buffer (`ring_buf_commit()`). The amount committed can be less than or equal to the amount provided by `ring_buf_put_ptr()`.

Data can be retrieved from a ring buffer through copying (see `ring_buf_get()`) or accessed directly by address. In the latter case, the operation is split into three stages:

1. Accessing the ring buffers internal buffer (see `ring_buf_get_ptr()`) to get a pointer to the next location where data can be read, and the amount of contiguous data available at that location.
2. Processing data
3. Signal to the ring buffer that the data has been consumed (see `ring_buf_consume()`). The amount consumed can be less than or equal to the amount provided by `ring_buf_get_ptr()`.

Concurrency The ring buffer APIs do not provide any internal concurrency control. Depending on usage (particularly with respect to number of concurrent readers/writers) applications may need to protect the ring buffer with mutexes and/or use semaphores to notify consumers that there is data to read.

A single producer and a single consumer running in separate execution contexts (for example two threads, or one thread and one ISR) may use the same ring buffer concurrently without additional locking. The producer side only updates the `put` indices and the consumer side only updates the `get` indices, so the two sides never write the same fields. This holds for both the copying APIs (`ring_buf_put()` / `ring_buf_get()`) and the zero-copy "claim" APIs (`ring_buf_put_claim()` / `ring_buf_put_finish()` and `ring_buf_get_claim()` / `ring_buf_get_finish()`).

When the producer and consumer run on different CPUs (SMP), the application must still ensure that data writes are visible before the index update that publishes them. In practice this happens for free when the producer and consumer use a kernel synchronization primitive to coordinate (for example a `k_sem` signaled by the producer and waited on by the consumer), since those primitives include the necessary memory barriers.

Any use case with more than one concurrent producer, or more than one concurrent consumer, must serialize those accesses externally (for example with a mutex or by disabling preemption).

Internal Operation Data streamed through a ring buffer is always written to the next byte within the buffer, wrapping around to the first element after reaching the end, thus the "ring" structure. Internally, the `struct ring_buf` contains its own buffer pointer and its size, and also a set of "head" and "tail" indices representing where the next read and write operations may occur.

This boundary is invisible to the user using the normal put/get APIs, but becomes a barrier to the "claim" API, because obviously no contiguous region can be returned that crosses the end of the buffer. This can be surprising to application code, and produce performance artifacts when transfers need to happen close to the end of the buffer, as the number of calls to claim/finish needs to double for such transfers.

Implementation

Defining a Ring Buffer A ring buffer is defined using a variable of type `ring_buf`. It must then be initialized by calling `ring_buf_init()`.

A ring buffer can be defined and initialized at compile time using the macros at file scope. The macro defines the ring buffer itself and its data buffer.

The following code defines a ring buffer:

```
#define MY_RING_BUF_BYTES 93
RING_BUF_DECLARE(my_ring_buf, MY_RING_BUF_BYTES);
```

Enqueuing Data Bytes are copied to a ring buffer by calling `ring_buf_put()`.

```
uint8_t my_data[MY_RING_BUF_BYTES];
uint32_t ret;

ret = ring_buf_put(&ring_buf, my_data, MY_RING_BUF_BYTES);
if (ret != MY_RING_BUF_BYTES) {
    /* not enough room, partial copy. */
    ...
}
```

Data can be added to a ring buffer by directly accessing the ring buffer's memory. For example:

```
uint32_t size;
uint32_t rx_size;
uint8_t *data;

/* Get pointer to writable area within the ring buffer memory. */
size = ring_buf_put_ptr(&ring_buf, &data, 0);

/* Work directly on a ring buffer memory. */
rx_size = uart_rx(data, size);

/* Indicate amount of valid data. rx_size must be equal or less than size. */
ring_buf_commit(&ring_buf, rx_size);
```

Retrieving Data Data bytes are copied out from a ring buffer by calling `ring_buf_get()`. For example:

```
uint8_t my_data[MY_DATA_BYTES];
size_t  ret;

ret = ring_buf_get(&ring_buf, my_data, sizeof(my_data));
if (ret != sizeof(my_data)) {
    /* Fewer bytes copied. */
} else {
    /* Requested amount of bytes retrieved. */
    ...
}
```

Data can be retrieved from a ring buffer by direct operations on the ring buffer's memory. For example:

```
uint32_t size;
uint32_t proc_size;
uint8_t *data;

/* Get pointer to readable data within the ring buffer memory. */
size = ring_buf_get_ptr(&ring_buf, &data, 0);

/* Work directly on a ring buffer memory. */
proc_size = process(data, size);

/* Indicate amount of data that has been consumed. proc_size must be equal
 * or less than size.
 */
ring_buf_consume(&ring_buf, proc_size);
```

Configuration Options

Related configuration options:

- `CONFIG_RING_BUFFER`: Restore the deprecated legacy ring buffer APIs (the claim/finish and fixed-size item APIs). The ring buffer itself is header-only and always available, so this option is not required for normal use.
- `CONFIG_RING_BUFFER_LARGE`: Increase the maximum buffer size from 32KB to 1GB.

API Reference

The following ring buffer APIs are provided by include/zephyr/sys/ring_buffer.h:

Ring Buffer APIs

3.5.7 Multi Producer Single Consumer Lock Free Queue

A *Multi Producer Single Consumer Lock Free Queue (MPSC)* is an lockfree intrusive queue based on atomic pointer swaps as described by Dmitry Vyukov at 1024cores.

API Reference

MPSC Lockfree Queue API

3.5.8 Single Producer Single Consumer Lock Free Queue

A *Single Producer Single Consumer Lock Free Queue (SPSC)* is a lock free atomic ring buffer based queue.

API Reference

SPSC API

3.5.9 Min-Heap Data Structure

- *Heap Structure*
- *Use Cases*
- *Value and reference variants*
- *Samples*
- *API Reference*
  - *Value heap*
  - *Reference heap*

The Min-Heap implementation provides an efficient data structure for managing a dynamically changing list of elements while maintaining the ability to quickly extract the minimum value. It's a tree-based data structure that satisfies the heap property and supports common operations such as insertion, removal and popping the minimum element from the Min-Heap

This section explains the motivation behind the implementation, its internal structure, API usage, and example scenarios for embedded systems and real-time environments.

Heap Structure

The heap is maintained as a complete binary tree stored in an array. Each node satisfies the **min-heap** property:

- The value of each node is less than or equal to the values of its children.

This property ensures that the **minimum element is always at the root** (index 0).

```
Index:      0   1   2   3   4   5   6
Values:    [1,  3,  5,  8,  9, 10, 12]

               1
             /   \
           3       5
          / \     / \
         8   9  10  12
```

For any node at index `i`, its children are at indices:

- Left child: $2 * i + 1$
- Right child: $2 * i + 2$

Its parent is at index:

- Parent: $(i - 1)/2$

Use Cases

MinHeap is well suited for:

- Implementing priority queues
- Sorting data incrementally

- Resource arbitration (e.g., lowest-cost resource selection)
- Scheduling in cooperative multitasking systems
- Managing timeouts and delay queues
- Priority-based sensor or data sampling

In RTOS environments like Zephyr, this heap can be used in kernel-level or application-level modules where minimal latency to obtain the highest priority resource is needed.

Value and reference variants

Two variants are provided, differing only in how elements are stored:

- `min_heap` (described above) stores elements **by value** in a contiguous buffer. An arbitrary element is removed by content (search) or by index. This suits priority queues of small values looked up by content.
- `min_heap_ref` stores **references** to caller-owned nodes, each embedding a `min_heap_handle`. Insertion is zero-copy, and a specific node is removed in $O(\log n)$ using the handle the caller already holds, with no search. This suits long-lived objects removed by identity rather than by content, such as kernel timeouts.

The two are complementary: the value heap cannot remove by identity in better than $O(n)$, while the reference heap is a poor fit for value-with-search use. `min_heap_ref` is header-only and requires no Kconfig.

Samples

min-heap sample demos the API usage of Min-Heap implementation in Zephyr RTOS.

API Reference

Value heap
Min-heap

Reference heap
Reference min-heap

3.5.10 sys_ringq Data Structure

- *Overview*
  - *Concurrency*
- *Instantiation & Usage*
- *API Reference*

Overview

A ring queue (or circular buffer) is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure is particularly useful for buffering items in a ordered manner.

The ringq is used whenever you need to decouple producers and consumers of discrete data sizes without worrying about partial reads or variable-length payloads. This differentiation separates it from the *ring_buffer* data structure, which is a streaming data structure of bytes.

Concurrency The sys_ringq api do not provide any concurrency control. Depending on usage, applications should protect the sys_ringq structure with appropriate synchronization mechanisms (e.g., mutexes, semaphores) to ensure thread safety when accessed from multiple threads.

Instantiation & Usage

A `sys_ringq` can be declared either using the `SYS_RINGQ_DEFINE(name, item_size, item_capacity)` macro or in runtime using the `sys_ringq_init(struct sys_ringq *ringq, uint8_t *data, size_t data_size, size_t item_size);` function.

```
SYS_RINGQ_DEFINE(my_ringq, item_size, item_capacity);
/* equivalent to */

static struct sys_ringq my_ringq;
static uint8_t buffer[item_size * item_capacity];
void init_fn (void) {
   sys_ringq_init(&my_ringq, buffer, sizeof(buffer), item_size);
   ....
}
```

Once the `sys_ringq` is initialized, items can be added to the ringq using the `sys_ringq_put()` function and removed using the `sys_ringq_get()` function. The ringq maintains the order of the items and ensures proper boundary checking of the underlying data buffer.

```
struct my_item item_to_add = { ... };
struct my_item item_removed;

/* Add an item to the queue */
if (sys_ringq_put(&my_ringq, &item_to_add) == 0) {
    // Item added successfully
} else {
    // ringq is full
}

/* Remove an item from the queue */
if (sys_ringq_get(&my_ringq, &item_removed) == 0) {
    // Item removed successfully
} else {
    // ringq is empty
}
```

In addition to the standard data operations (sys_ringq_put() and sys_ringq_get()), the sys_ringq API provides a set of utility functions for managing and inspecting the data structures state.

- sys_ringq_capacity() – Returns the total capacity of the sys_ringq in terms of the number of items it can hold.
- sys_ringq_empty() – Returns true if the sys_ringq contains no items.
- sys_ringq_full() – Returns true if the sys_ringq cannot accept more items.
- sys_ringq_space() – Returns the number of free slots remaining.
- sys_ringq_size() – Returns the number of items currently stored.
- sys_ringq_reset() – Resets the sys_ringq to an empty state.

API Reference

Ring queue

3.6 Executing Time Functions

The timing functions can be used to obtain execution time of a section of code to aid in analysis and optimization.

Please note that the timing functions may use a different timer than the default kernel timer, where the timer being used is specified by architecture, SoC or board configuration.

3.6.1 Configuration

To allow using the timing functions, `CONFIG_TIMING_FUNCTIONS` needs to be enabled.

3.6.2 Usage

To gather timing information:

1. Call `timing_init()` to initialize the timer.
2. Call `timing_start()` to signal the start of gathering of timing information. This usually starts the timer.
3. Call `timing_counter_get()` to mark the start of code execution.
4. Call `timing_counter_get()` to mark the end of code execution.
5. Call `timing_cycles_get()` to get the number of timer cycles between start and end of code execution.
6. Call `timing_cycles_to_ns()` with total number of cycles to convert number of cycles to nanoseconds.
7. Repeat from step 3 to gather timing information for other blocks of code.
8. Call `timing_stop()` to signal the end of gathering of timing information. This usually stops the timer.

Example

This shows an example on how to use the timing functions:

```
#include <zephyr/timing/timing.h>

void gather_timing(void)
{
    timing_t start_time, end_time;
    uint64_t total_cycles;
    uint64_t total_ns;

    timing_init();
    timing_start();

    start_time = timing_counter_get();

    code_execution_to_be_measured();

    end_time = timing_counter_get();

    total_cycles = timing_cycles_get(&start_time, &end_time);
    total_ns = timing_cycles_to_ns(total_cycles);

    timing_stop();
}
```

3.6.3 API documentation

Timing Measurement APIs

Arch specific Timing Measurement APIs

SoC specific Timing Measurement APIs

Board specific Timing Measurement APIs

3.7 Object Cores

Object cores are a kernel debugging tool that can be used to both identify and perform operations on registered objects.

- *Object Core Concepts*
- *Object Core Statistics Concepts*
- *Implementation*
  - *Defining a New Object Type*
  - *Initializing a New Object Core*
  - *Walking a List of Object Cores*
  - *Object Core Statistics Querying*
- *Configuration Options*
- *API Reference*

3.7.1 Object Core Concepts

Each instance of an object embeds an object core field named `obj_core`. An object core links the object to its object type, and each object type lets debugging tools enumerate the objects of that type. Object types are linked together via a singly linked list. Together, this allows debugging tools to traverse all the objects in the system.

An object type enumerates its objects from two places:

- Its permanent objects, which are statically defined instances such as those created with `K_SEM_DEFINE`. They are walked in place from their iterable section and are never registered at run time.
- A bounded registry of the objects initialized at run time, for instance a semaphore in a driver's data structure initialized with `k_sem_init()`. The registry only references the objects; it never stores anything inside them.

Because of that, an object may be initialized in any storage, initialized again in place, or discarded without any object core call: the registry cannot be corrupted by an object that no longer exists. Some rules follow from this model:

- An object located in the running thread's stack or in the interrupt stack is not registered: its life ends with the stack frame. The object type counts such objects in its `skipped` field. Other stacks are not recognized, such as the privileged stack a user thread's system calls run on or an exception stack specific to an architecture; an object there is registered and stays reported until its storage is reused.
- When the registry is full, further objects are not registered and the object type counts them in its `dropped` field. The registry size is set with `CONFIG_OBJ_CORE_MAX_DYNAMIC_OBJECTS`.

- An object that was discarded without being unregistered stays reported until its storage is reused: the walk recognizes the reused storage by the missing type tag and drops the entry. The kernel unregisters objects itself when it releases them: on thread abort, in `k_object_free()`, `k_timer_cleanup()`, `k_msgq_cleanup()` and `k_stack_cleanup()`, and, with `CONFIG_OBJ_CORE_EVICT_ON_FREE`, when memory is returned to a heap, a memory slab or a memory blocks allocator. Code that ends an object's life in another way may call `k_obj_core_unlink()` so that the object stops being reported at once; the type tag check is only a backstop, and it reads the object's former storage, which must still be mapped.
- Once a type's `dropped` count is not zero, objects exist that no walk reports, so an inventory is complete only while it is zero.

Object cores have been integrated into the following kernel objects:

- *Condition Variables*
- *Events*
- *FIFOs* and *LIFOs*
- *Mailboxes*
- *Memory Slabs*
- *Message Queues*
- *Mutexes*
- *Pipes*
- *Queues*
- *Semaphores*
- *Threads*
- *Timers*
- *System Memory Blocks*

Developers are free to integrate them if desired into other objects within their projects. The object_cores sample shows the facility step by step and the object_monitor sample uses it to watch the objects of a running application.

3.7.2 Object Core Statistics Concepts

A variety of kernel objects allow for the gathering and reporting of statistics. Object cores provide a uniform means to retrieve that information via object core statistics. When enabled, the object type contains a pointer to a statistics descriptor that defines the various operations that have been enabled for interfacing with the object's statistics. Additionally, the object core contains a pointer to the "raw" statistical information associated with that object. Raw data is the raw, unmanipulated data associated with the statistics. Queried data may be "raw", but it may also have been manipulated in some way by calculation (such as determining an average).

The following table indicates both what objects have been integrated into the object core statistics as well as the structures used for both "raw" and "queried" data.

Object	Raw Data Type	Query Data Type
struct mem_slab	struct mem_slab_info	struct sys_memory_stats
struct sys_mem_blocks	struct sys_mem_blocks_info	struct sys_memory_stats
struct k_thread	struct k_cycle_stats	struct k_thread_runtime_stats
struct _cpu	struct k_cycle_stats	struct k_thread_runtime_stats
struct z_kernel	struct k_cycle_stats[num CPUs]	struct k_thread_runtime_stats

3.7.3 Implementation

Defining a New Object Type

An object type is a global variable of type `k_obj_type`. When the object struct has statically defined instances in an iterable section, the type is defined at build time with `K_OBJ_TYPE_DEFINE` and those instances become its permanent objects. The following code shows how a new object type can be defined for use with object cores and object core statistics.

```
/* Unique object type ID */

#define K_OBJ_TYPE_MY_NEW_TYPE  K_OBJ_TYPE_ID_GEN("UNIQ")

struct my_obj_type_raw_info {
    ...
};

struct my_obj_type_query_stats {
    ...
};

struct my_new_obj {
    ...
    struct k_obj_core obj_core;
    struct my_obj_type_raw_info  info;
};

struct k_obj_core_stats_desc my_obj_type_stats_desc = {
    .raw_size = sizeof(struct my_obj_type_raw_stats),
    .query_size = sizeof(struct my_obj_type_query_stats),
    .raw = my_obj_type_stats_raw,
    .query = my_obj_type_stats_query,
    .reset = my_obj_type_stats_reset,
    .disable = NULL,    /* Stats gathering is always on */
    .enable = NULL,     /* Stats gathering is always on */
};

K_OBJ_TYPE_DEFINE_STATS(my_obj_type, my_new_obj, K_OBJ_TYPE_MY_NEW_TYPE,
                        &my_obj_type_stats_desc, info);
```

A type whose objects are not in an iterable section is initialized at run time instead, before any of its objects. A permanent array of objects can be registered as the type's range so that its objects need no registry entry.

```
struct k_obj_type  my_obj_type;
struct my_new_obj  my_objects[8];

void my_obj_type_init(void)
{
    z_obj_type_init(&my_obj_type, K_OBJ_TYPE_MY_NEW_TYPE,
                    offsetof(struct my_new_obj, obj_core));
    k_obj_type_init_range(&my_obj_type, my_objects,
                          &my_objects[ARRAY_SIZE(my_objects)],
                          sizeof(struct my_new_obj), false);
    k_obj_type_stats_init(&my_obj_type, &my_obj_type_stats_desc);
}
```

Initializing a New Object Core

Kernel objects that have already been integrated into the object core framework automatically have their object cores initialized when the object is initialized. However, developers that wish to add their own objects into the framework need to both initialize the object core and register it. Registering an object that belongs to the type's permanent range has no effect, as it is reported from the range. The following code builds on the example above and initializes the object core.

```
void my_new_obj_init(struct my_new_obj *new_obj)
{
    ...
    k_obj_core_init(K_OBJ_CORE(new_obj), &my_obj_type);
    k_obj_core_link(K_OBJ_CORE(new_obj));
    k_obj_core_stats_register(K_OBJ_CORE(new_obj), &new_obj->raw_stats,
                              sizeof(struct my_obj_type_raw_info));
}
```

Walking a List of Object Cores

Two routines exist for walking the object cores of an object type. These are `k_obj_type_walk_locked()` and `k_obj_type_walk_unlocked()`. Both visit the permanent objects of the type first and the registered objects next. The following code builds upon the example above and prints the addresses of all the objects of that new object type.

```
int walk_op(struct k_obj_core *obj_core, void *data)
{
    uint8_t *ptr;

    ptr = obj_core;
    ptr -= obj_core->type->obj_core_offset;

    printk("%p\n", ptr);

    return 0;
}

void print_object_addresses(void)
{
    struct k_obj_type *obj_type;

    /* Find the object type */

    obj_type = k_obj_type_find(K_OBJ_TYPE_MY_NEW_TYPE);

    /* Walk the list of objects */

    k_obj_type_walk_unlocked(obj_type, walk_op, NULL);
}
```

Object Core Statistics Querying

The following code builds on the examples above and shows how an object integrated into the object core statistics framework can both retrieve queried data and reset the stats associated with the object.

```
struct my_new_obj my_obj;
```

(continues on next page)

(continued from previous page)

```
...

void my_func(void)
{
    struct my_obj_type_query_stats  my_stats;
    int  status;

    my_obj_type_init(&my_obj);

    ...

    status = k_obj_core_stats_query(K_OBJ_CORE(&my_obj),
                                    &my_stats, sizeof(my_stats));
    if (status != 0) {
        /* Failed to get stats */
        ...
    } else {
        k_obj_core_stats_reset(K_OBJ_CORE(&my_obj));
    }

    ...
}
```

3.7.4 Configuration Options

Related configuration options:

- CONFIG_OBJ_CORE
- CONFIG_OBJ_CORE_MAX_DYNAMIC_OBJECTS
- CONFIG_OBJ_CORE_EVICT_ON_FREE
- CONFIG_OBJ_CORE_CONDVAR
- CONFIG_OBJ_CORE_EVENT
- CONFIG_OBJ_CORE_FIFO
- CONFIG_OBJ_CORE_LIFO
- CONFIG_OBJ_CORE_MAILBOX
- CONFIG_OBJ_CORE_MEM_SLAB
- CONFIG_OBJ_CORE_MSGQ
- CONFIG_OBJ_CORE_MUTEX
- CONFIG_OBJ_CORE_PIPE
- CONFIG_OBJ_CORE_QUEUE
- CONFIG_OBJ_CORE_SEM
- CONFIG_OBJ_CORE_STACK
- CONFIG_OBJ_CORE_THREAD
- CONFIG_OBJ_CORE_TIMER
- CONFIG_OBJ_CORE_SYS_MEM_BLOCKS
- CONFIG_OBJ_CORE_STATS
- CONFIG_OBJ_CORE_STATS_MEM_SLAB

- CONFIG_OBJ_CORE_STATS_THREAD
- CONFIG_OBJ_CORE_STATS_SYSTEM
- CONFIG_OBJ_CORE_STATS_SYS_MEM_BLOCKS

3.7.5 API Reference

Object Core APIs

Related code samples

- Object coresEnumerate the kernel objects of a running system and query their statistics.
- Object monitorWatch kernel objects come and go under a running application.

Object Core Statistics APIs

Related code samples

- Object coresEnumerate the kernel objects of a running system and query their statistics.
- Object monitorWatch kernel objects come and go under a running application.

3.8 Time Utilities

3.8.1 Overview

Uptime in Zephyr is based on a tick counter. With the default CONFIG_TICKLESS_KERNEL, this counter advances at a nominally constant rate from zero at the instant the system started. The POSIX equivalent to this counter is something like CLOCK_MONOTONIC or, in Linux, CLOCK_MONOTONIC_RAW. k_uptime_get() provides a millisecond representation of this time.

Applications often need to correlate the Zephyr internal time with external time scales used in daily life, such as local time or Coordinated Universal Time. These systems interpret time in different ways and may have discontinuities due to leap seconds and local time offsets like daylight saving time.

Because of these discontinuities, as well as significant inaccuracies in the clocks underlying the cycle counter, the offset between time estimated from the Zephyr clock and the actual time in a "real" civil time scale is not constant and can vary widely over the runtime of a Zephyr application.

The time utilities API supports:

- *converting between time representations*
- *synchronizing and aligning time scales*
- *comparing, adding, and subtracting representations*

For terminology and concepts that support these functions see *Concepts Underlying Time Support in Zephyr*.

3.8.2 Time Utility APIs

Representation Transformation

Time scale instants can be represented in multiple ways including:

- Seconds since an epoch. POSIX representations of time in this form include `time_t` and `struct timespec`, which are generally interpreted as a representation of "UNIX Time" (see **RFC 8536 Section 2**).
- Calendar time as a year, month, day, hour, minutes, and seconds relative to an epoch. POSIX representations of time in this form include `struct tm`.

Keep in mind that these are simply time representations that must be interpreted relative to a time scale which may be local time, UTC, or some other continuous or discontinuous scale.

Some necessary transformations are available in standard C library routines. For example, `time_t` measuring seconds since the POSIX EPOCH is converted to `struct tm` representing calendar time with gmtime(). Sub-second timestamps like `struct timespec` can also use this to produce the calendar time representation and deal with sub-second offsets separately.

The inverse transformation is not standardized: APIs like `mktime()` expect information about time zones. Zephyr provides this transformation with `timeutil_timegm()` and `timeutil_timegm64()`.

To convert between `struct timespec` and `k_timeout_t` durations, use `timespec_to_timeout()` and `timespec_from_timeout()`.

```
k_timeout_t to;
struct timespec ts;

timespec_from_timeout(K_FOREVER, &ts);
to = timespec_to_timeout(&ts); /* to == K_FOREVER */

timespec_from_timeout(K_MSEC(100), &ts);
to = timespec_to_timeout(&ts); /* to == K_MSEC(100) */
```

Time Representation APIs

Time Scale Synchronization

There are several factors that affect synchronizing time scales:

- The rate of discrete instant representation change. For example, Zephyr uptime is tracked in ticks which advance at events that nominally occur at `CONFIG_SYS_CLOCK_TICKS_PER_SEC` Hertz, while an external time source may provide data in whole or fractional seconds (e.g. microseconds).
- The absolute offset required to align the two scales at a single instant.
- The relative error between observable instants in each scale, required to align multiple instants consistently. For example, a reference clock that's conditioned by a 1-pulse-per-second GPS signal will be much more accurate than a Zephyr system clock driven by a RC oscillator with a +/- 250 ppm error.

Synchronization or alignment between time scales is done with a multi-step process:

- An instant in a time scale is represented by an (unsigned) 64-bit integer, assumed to advance at a fixed nominal rate.
- `timeutil_sync_config` records the nominal rates of a reference time scale/source (e.g. TAI) and a local time source (e.g. `k_uptime_ticks()`).
- `timeutil_sync_instant` records the representation of a single instant in both the reference and local time scales.
- `timeutil_sync_state` provides storage for an initial instant, a recently received second observation, and a skew that can adjust for relative errors in the actual rate of each time scale.

- `timeutil_sync_ref_from_local()` and `timeutil_sync_local_from_ref()` convert instants in one time scale to another taking into account skew that can be estimated from the two instances stored in the state structure by `timeutil_sync_estimate_skew()`.

Time Synchronization APIs

`timespec` Manipulation

Checking the validity of a `timespec` can be done with `timespec_is_valid()`.

```
struct timespec ts = {
    .tv_sec = 0,
    .tv_nsec = -1, /* out of range! */
};

if (!timespec_is_valid(&ts)) {
    /* error-handing code */
}
```

In some cases, invalid `timespec` objects may be re-normalized using `timespec_normalize()`.

```
if (!timespec_normalize(&ts)) {
    /* error-handling code */
}

/* ts should be normalized */
__ASSERT(timespec_is_valid(&ts) == true, "expected normalized timespec");
```

It is possible to compare two `timespec` objects for equality using `timespec_equal()`.

```
if (timespec_equal(then, now)) {
    /* time is up! */
}
```

It is possible to compare and fully order (valid) `timespec` objects using `timespec_compare()`.

```
int cmp = timespec_compare(a, b);

switch (cmp) {
case 0:
    /* a == b */
    break;
case -1:
    /* a < b */
    break;
case +1:
    /* a > b */
    break;
}
```

It is possible to add, subtract, and negate `timespec` objects using `timespec_add()`, `timespec_sub()`, and `timespec_negate()`, respectively. Like `timespec_normalize()`, these functions will output a normalized `timespec` when doing so would not result in overflow. On success, these functions return `true`. If overflow would occur, the functions return `false`.

```
/* a += b */
if (!timespec_add(&a, &b)) {
    /* overflow */
}
```

(continues on next page)

(continued from previous page)

```
/* a -= b */
if (!timespec_sub(&a, &b)) {
    /* overflow */
}

/* a = -a */
if (!timespec_negate(&a)) {
    /* overflow */
}
```

Timespec Utility APIs

3.8.3 Concepts Underlying Time Support in Zephyr

Terms from ISO/TC 154/WG 5 N0038 (ISO/WD 8601-1) and elsewhere:

- A *time axis* is a representation of time as an ordered sequence of instants.
- A *time scale* is a way of representing an instant relative to an origin that serves as the epoch.
- A time scale is *monotonic* (increasing) if the representation of successive time instants never decreases in value.
- A time scale is *continuous* if the representation has no abrupt changes in value, e.g. jumping forward or back when going between successive instants.
- Civil time generally refers to time scales that legally defined by civil authorities, like local governments, often to align local midnight to solar time.

Relevant Time Scales

International Atomic Time (TAI) is a time scale based on averaging clocks that count in SI seconds. TAI is a monotonic and continuous time scale.

Universal Time (UT) is a time scale based on Earth's rotation. UT is a discontinuous time scale as it requires occasional adjustments (leap seconds) to maintain alignment to changes in Earth's rotation. Thus the difference between TAI and UT varies over time. There are several variants of UT, with UTC being the most common.

UT times are independent of location. UT is the basis for Standard Time (or "local time") which is the time at a particular location. Standard time has a fixed offset from UT at any given instant, primarily influenced by longitude, but the offset may be adjusted ("daylight saving time") to align standard time to the local solar time. In a sense local time is "more discontinuous" than UT.

POSIX Time (see **RFC 8536 Section 2**) is a time scale that counts seconds since the "POSIX epoch" at 1970-01-01T00:00:00Z (i.e. the start of 1970 UTC). UNIX Time is an extension of POSIX time using negative values to represent times before the POSIX epoch. Both of these scales assume that every day has exactly 86400 seconds. In normal use instants in these scales correspond to times in the UTC scale, so they inherit the discontinuity.

The continuous analogue is UNIX Leap Time which is UNIX time plus all leap-second corrections added after the POSIX epoch (when TAI-UTC was 8 s).

Example of Time Scale Differences A positive leap second was introduced at the end of 2016, increasing the difference between TAI and UTC from 36 seconds to 37 seconds. There was no leap second introduced at the end of 1999, when the difference between TAI and UTC was only 32 seconds. The following table shows relevant civil and epoch times in several scales:

UTC Date	UNIX time	TAI Date	TAI-UTC	UNIX Leap Time
1970-01-01T00:00:00Z	0	1970-01-01T00:00:08	+8	0
1999-12-31T23:59:28Z	946684768	2000-01-01T00:00:00	+32	946684792
1999-12-31T23:59:59Z	946684799	2000-01-01T00:00:31	+32	946684823
2000-01-01T00:00:00Z	946684800	2000-01-01T00:00:32	+32	946684824
2016-12-31T23:59:59Z	1483228799	2017-01-01T00:00:35	+36	1483228827
2016-12-31T23:59:60Z	undefined	2017-01-01T00:00:36	+36	1483228828
2017-01-01T00:00:00Z	1483228800	2017-01-01T00:00:37	+37	1483228829

Functional Requirements The Zephyr tick counter has no concept of leap seconds or standard time offsets and is a continuous time scale. However it can be relatively inaccurate, with drifts as much as three minutes per hour (assuming an RC timer with 5% tolerance).

There are two stages required to support conversion between Zephyr time and common human time scales:

- Translation between the continuous but inaccurate Zephyr time scale and an accurate external stable time scale;
- Translation between the stable time scale and the (possibly discontinuous) civil time scale.

The API around `timeutil_sync_state_update()` supports the first step of converting between continuous time scales.

The second step requires external information including schedules of leap seconds and local time offset changes. This may be best provided by an external library, and is not currently part of the time utility APIs.

Selecting an External Source and Time Scale If an application requires civil time accuracy within several seconds then UTC could be used as the stable time source. However, if the external source adjusts to a leap second there will be a discontinuity: the elapsed time between two observations taken at 1 Hz is not equal to the numeric difference between their timestamps.

For precise activities a continuous scale that is independent of local and solar adjustments simplifies things considerably. Suitable continuous scales include:

- GPS time: epoch of 1980-01-06T00:00:00Z, continuous following TAI with an offset of TAI-GPS=19 s.
- Bluetooth Mesh time: epoch of 2000-01-01T00:00:00Z, continuous following TAI with an offset of -32.
- UNIX Leap Time: epoch of 1970-01-01T00:00:00Z, continuous following TAI with an offset of -8.

Because C and Zephyr library functions support conversion between integral and calendar time representations using the UNIX epoch, UNIX Leap Time is an ideal choice for the external time scale.

The mechanism used to populate synchronization points is not relevant: it may involve reading from a local high-precision RTC peripheral, exchanging packets over a network using a protocol like NTP or PTP, or processing NMEA messages received a GPS with or without a 1pps signal.

`timespec` Concepts

Originally from POSIX, `struct timespec` has been a part of the C standard since C11. The definition of `struct timespec` is as shown below.

```
struct timespec {
    time_t tv_sec;  /* seconds */
    long   tv_nsec; /* nanoseconds */
};
```

The `tv_nsec` field is only valid with values in the range `[0, 999999999]`. The `tv_sec` field is the number of seconds since the epoch. If `struct timespec` is used to express a difference, the `tv_sec` field may fall into a negative range.

3.9 Precision Timing

The precision timing subsystem provides a small set of reusable mechanisms for PTP, gPTP, and other users that control high-resolution clocks. Enable it with `CONFIG_PRECISION_TIMING`.

> **Note**
>
> The API is at *experimental* maturity and may still change. A protocol does not change maturity by using this subsystem.

The subsystem deliberately does not implement synchronization policy. PTP and gPTP continue to own their protocol state machines, clock selection, step thresholds, sample acceptance, lock detection, source-loss handling, and diagnostics.

3.9.1 Precision time

`precision_time_t` is a signed 64-bit nanosecond value. Checked addition and subtraction helpers report overflow. The type does not identify a time domain or timescale; modelling TAI, UTC, PHCs, monotonic time, and protocol relationships is outside this API.

3.9.2 Precision clock

`precision_clock` dispatches four mandatory clock operations:

- read the current time;
- set an absolute time;
- apply a phase adjustment; and
- set a rate offset from the nominal frequency as parts per million with a 16-bit binary fractional field.

All operations must be implemented by a clock adapter. Adjustment ranges and other hardware constraints remain the responsibility of the underlying clock implementation.

`precision_clock_ptp_adapter` exposes an existing Zephyr PTP clock device through this interface. It performs the required conversion between `net_ptp_time` and `precision_time_t`; it does not add capability discovery or synchronization state.

3.9.3 PI controller

`precision_pi` is an instance-based proportional-integral controller. Each instance stores its own gains and integral term. For every error sample, the update is equivalent to:

```
integral += ki * error;
output = kp * error + integral;
```

The controller does not decide whether an error should be stepped, rejected, or used for rate adjustment. It also does not track synchronization, acquisition, lock, holdover, source timeout, or clock faults. Callers own those decisions and reset the accumulated integral term when their policy requires it.

The PTP and gPTP integrations initialize their controllers from `CONFIG_PRECISION_TIMING_PI_KP` and `CONFIG_PRECISION_TIMING_PI_KI`. The integer values express the gains in thousandths. Direct users of `precision_pi_init()` may provide different gains for each controller instance.

3.9.4 Protocol integration

PTP and the gPTP default clock-update path each keep their existing policy and use a `precision_pi` for the shared calculation. They initialize a PTP-clock adapter once and use `precision_clock` operations to access the PHC.

3.9.5 Sample

The precision_timing sample demonstrates checked time arithmetic, a software-backed precision clock, and PI-driven rate adjustment.

3.9.6 API reference

Precision Timing

Precision Time

Related code samples

- Precision timingUse checked precision-time arithmetic, a precision-clock adapter, and a PI controller.

Precision Clock

Related code samples

- Precision timingUse checked precision-time arithmetic, a precision-clock adapter, and a PI controller.

Precision Clock PTP Adapter

Precision PI Controller

Related code samples

- Precision timingUse checked precision-time arithmetic, a precision-clock adapter, and a PI controller.

3.10 Utilities

This page contains reference documentation for `<sys/util.h>`, which provides miscellaneous utility functions and macros.

Utility Functions

3.11 Iterable Sections

This page contains the reference documentation for the iterable sections APIs, which can be used for defining iterable areas of equally-sized data structures, that can be iterated on using `STRUCT_SECTION_FOREACH`.

3.11.1 Overview

An iterable section is a group of statically-defined instances of the same struct that the linker places into a single contiguous output section. The runtime can then iterate over all instances without having to maintain an explicit list.

Zephyr currently supports two linker-script pipelines: a template-based one that consumes `.ld` scripts registered via `zephyr_linker_sources()`, and a CMake generated one that consumes `zephyr_iterable_section()` calls. To work on all supported toolchains, a new iterable must currently be declared in both.

When `CONFIG_CMAKE_LINKER_GENERATOR=y`, the output sections for iterable sections are emitted from the `zephyr_iterable_section()` calls. Any `ITERABLE_SECTION_RAM/ROM` definitions in `.ld` snippets registered via `zephyr_linker_sources()` are ignored. When `CONFIG_CMAKE_LINKER_GENERATOR=n`, the reverse holds true: `zephyr_iterable_section()` calls are not consumed and the `.ld` scripts provide the section definitions.

Because upstream Zephyr must build under both configurations, a new iterable section must be declared in both places.

To create an iterable section requires three pieces that must agree on the struct name and RAM vs. ROM placement:

1. **C code**: defines the struct and instantiates entries using `STRUCT_SECTION_ITERABLE` (or its `const` variant for ROM).
2. **Linker placement**: declared in one of two ways (*both required upstream*):
   - **CMake**: `zephyr_iterable_section()` consumed by the CMake-generated pipeline.
   - **Linker script**: `ITERABLE_SECTION_RAM/ROM` registered with `zephyr_linker_sources()` consumed by the template-based pipeline.

3.11.2 Step 1: Define the data in C

Define the struct in a common header and provide a helper macro that instantiates entries with `STRUCT_SECTION_ITERABLE`:

```
struct my_data {
        int a, b;
};

#define DEFINE_DATA(name, _a, _b) \
        STRUCT_SECTION_ITERABLE(my_data, name) = { \
                .a = _a, \
                .b = _b, \
        }

...

DEFINE_DATA(d1, 1, 2);
DEFINE_DATA(d2, 3, 4);
DEFINE_DATA(d3, 5, 6);
```

For ROM-resident iterables, the instances must be declared `const` so the compiler emits them into a read-only input section. The C declaration and the CMake placement (see Step 2) must agree on RAM vs. ROM placement.

3.11.3 Step 2: Declare the section in CMake

The `NAME` argument to `zephyr_iterable_section()` must match the struct name passed to `STRUCT_SECTION_ITERABLE`. The `GROUP` argument selects the linker group that the resulting output section is placed inside.

RAM-resident example:

```
# CMakeLists.txt
zephyr_iterable_section(NAME my_data GROUP DATA_REGION ${XIP_ALIGN_WITH_INPUT})
```

ROM-resident example (instances declared `const` in C):

```
# CMakeLists.txt
zephyr_iterable_section(NAME my_data GROUP RODATA_REGION)
```

See `zephyr_iterable_section()` in `cmake/modules/extensions.cmake` for the full argument list and a more detailed explanation of the available `GROUP` options.

3.11.4 Step 3: Provide a linker-script

The linker script uses `ITERABLE_SECTION_RAM` or `ITERABLE_SECTION_ROM` to emit the actual section.

RAM-resident:

```
/* sections-ram.ld */
#include <zephyr/linker/iterable_sections.h>

ITERABLE_SECTION_RAM(my_data, Z_LINK_ITERABLE_SUBALIGN)
```

ROM-resident:

```
/* sections-rom.ld */
#include <zephyr/linker/iterable_sections.h>

ITERABLE_SECTION_ROM(my_data, Z_LINK_ITERABLE_SUBALIGN)
```

Register the linker-script from `CMakeLists.txt`:

```
zephyr_linker_sources(<location> <path-to-ld-file>)
```

See `zephyr_linker_sources()` in `cmake/modules/extensions.cmake` for the full argument list and a more detailed explanation of the available `<location>` options.

3.11.5 Iterating over the entries

Once the section is in place, iterate over its entries with `STRUCT_SECTION_FOREACH`:

```
STRUCT_SECTION_FOREACH(my_data, data) {
        printk("%p: a: %d, b: %d\n", data, data->a, data->b);
}
```

> **Note**
>
> The linker is going to place the entries sorted by name, so the example above would visit `d1`, `d2` and `d3` in that order, regardless of how they were defined in the code.

3.11.6 API Reference

Iterable Sections APIs

3.12 Code And Data Relocation

3.12.1 Overview

This feature will allow relocating .text, .rodata, .data, and .bss sections from required files and place them in the required memory region. The memory region and file are given to the *scripts/build/gen_relocate_app.py* script in the form of a string. This script is always invoked from inside cmake.

This script provides a robust way to re-order the memory contents without actually having to modify the code. In simple terms this script will do the job of `__attribute__((section("name")))` for a bunch of files together.

A regular expression filter can be used to select only the required sections to be relocated.

3.12.2 Details

The memory region and file are given to the *scripts/build/gen_relocate_app.py* script through a file where each line specifies a list of files to be placed in the given region.

An example of such a file is:

```
SRAM2:/home/xyz/zephyr/samples/hello_world/src/main.c,
SRAM1:/home/xyz/zephyr/samples/hello_world/src/main2.c,
```

This script is invoked with the following parameters: `python3 gen_relocate_app.py -i input_file -o generated_linker -c generated_code`

Kconfig `CONFIG_CODE_DATA_RELOCATION` option, when enabled in `prj.conf`, will invoke the script and do the required relocation.

This script also trigger the generation of `linker_relocate.ld` and `code_relocation.c` files. The `linker_relocate.ld` file creates appropriate sections and links the required functions or variables from all the selected files.

> **Note**
>
> The text section is split into 2 parts in the main linker script. The first section will have some info regarding vector tables and other debug related info. The second section will have the complete text section. This is needed to force the required functions and data variables to the correct locations. This is due to the behavior of the linker. The linker will only link once and hence this text section had to be split to make room for the generated linker script.

The `code_relocation.c` file has code that is needed for initializing data sections, and a copy of the text sections (if XIP). Also this contains code needed for bss zeroing and for data copy operations from ROM to required memory type.

The procedure to invoke this feature is:

- Enable `CONFIG_CODE_DATA_RELOCATION` in the `prj.conf` file
- Inside the `CMakeLists.txt` file in the project, mention all the files that need relocation.

  `zephyr_code_relocate(FILES src/main.c LOCATION SRAM2)`

  Where the first argument is the file/files and the second argument is the memory where it must be placed.

> **Note**
>
> function `zephyr_code_relocate()` can be called as many times as required.

Additional Configurations

This section shows additional configuration options that can be set in `CMakeLists.txt`.

- If the memory is `SRAM1`, `SRAM2`, `CCD`, or `AON`, then place the full object in the sections. For example:

```
zephyr_code_relocate(FILES src/file1.c LOCATION SRAM2)
zephyr_code_relocate(FILES src/file2.c LOCATION SRAM1)
```

- If the memory type is appended with `_DATA`, `_TEXT`, `_RODATA`, `_BSS` or `_NOINIT`, only the selected memory is placed in the required memory region. For example:

```
zephyr_code_relocate(FILES src/file1.c LOCATION SRAM2_DATA)
zephyr_code_relocate(FILES src/file2.c LOCATION SRAM2_TEXT)
```

- Multiple regions can also be appended together such as: `SRAM2_DATA_BSS_NOINIT`. This will place all data: value-initialized, zero-initialized and uninitialized inside `SRAM2`.
- Multiple files can be passed to the `FILES` argument, or CMake generator expressions can be used to relocate a comma-separated list of files.

```
file(GLOB sources "file*.c")
zephyr_code_relocate(FILES ${sources} LOCATION SRAM2)
zephyr_code_relocate(FILES $<TARGET_PROPERTY:my_tgt,SOURCES> LOCATION SRAM2)
```

Section Filtering

By default, all sections of the specified files will be relocated. If `FILTER` is used, a regular expression is provided to select only the sections to be relocated.

The regular expression applies to sections names which can be used to select the file's symbols when this one has been built with `-ffunction-sections` and `-fdata-sections` which is the case by default.

```
zephyr_code_relocate(FILES src/file1.c FILTER ".*\\.func1|.*\\.func2"␣
↪LOCATION SRAM2_TEXT)
```

The example above will only relocate `func1()` and `func2()` of file `src/file1.c`

NOKEEP flag

By default, all relocated functions and variables will be marked with `KEEP()` when generating `linker_relocate.ld`. Therefore, if any input file happens to contain unused symbols, then they will not be discarded by the linker, even when it is invoked with `--gc-sections`. If you'd like to override this behavior, you can pass `NOKEEP` to your `zephyr_code_relocate()` call.

```
zephyr_code_relocate(FILES src/file1.c LOCATION SRAM2_TEXT NOKEEP)
```

The example above will help ensure that any unused code found in the .text sections of `file1.c` will not stick to SRAM2.

NOCOPY flag

When a `NOCOPY` option is passed to the `zephyr_code_relocate()` function, the relocation code is not generated in `code_relocation.c`. This flag can be used when we want to move the content of a specific file (or set of files) to a XIP area.

This example will place the .text section of the `xip_external_flash.c` file to the `EXTFLASH` memory region where it will be executed from (XIP). The .data will be relocated as usual into SRAM.

```
zephyr_code_relocate(FILES src/xip_external_flash.c LOCATION␣
↪EXTFLASH_TEXT NOCOPY)
zephyr_code_relocate(FILES src/xip_external_flash.c LOCATION SRAM_DATA)
```

Relocating libraries

Libraries can be relocated using the LIBRARY argument to `zephyr_code_relocation()` with the library name. For example, the following snippet will relocate serial drivers to SRAM2:

```
zephyr_code_relocate(LIBRARY drivers__serial LOCATION SRAM2)
```

Tips

Take care if relocating kernel/arch files, some contain early initialization code that executes before code relocation takes place.

Additional MPU/MMU configuration may be required to ensure that the destination memory region is configured to allow code execution.

Samples/ Tests

A test showcasing this feature is provided at `$ZEPHYR_BASE/tests/application_development/code_relocation`

This test shows how the code relocation feature is used.

This test will place .text, .data, .bss from 3 files to various parts in the SRAM using a custom linker file derived from `include/zephyr/arch/arm/cortex_m/scripts/linker.ld`

A sample showcasing the NOCOPY flag is provided here: code_relocation_nocopy.

3.13 Scope-based Cleanup Helpers

- *Overview*
- *Core Concepts*
  - *Scoped Variables*
  - *Scoped Guards*
  - *Scoped Defers*
- *Defining Scoped Types*

  - *Custom Scoped Variables*
  - *Scoped Guards*
  - *Block-Scoped Guards*
  - *Conditional Guards*
  - *Scoped Defers*
- *Usage Notes*
  - *Order of Cleanup*
  - *Scope Rules*
- *API Reference*

3.13.1 Overview

The Cleanup Helper API provides a mechanism for automatic resource cleanup when variables go out of scope. This is similar to RAII in C++ or defer statements in Go. By leveraging compiler support for the `__cleanup` attribute, this API helps prevent resource leaks and simplifies error handling by ensuring cleanup code is executed automatically.

It is required to set `CONFIG_SCOPE_CLEANUP_HELPERS` to enable this feature, which is particularly useful for:

- Automatic unlocking of mutexes and semaphores
- Automatic freeing of dynamically allocated memory
- Ensuring cleanup actions occur on all code paths (including early returns)
- Reducing boilerplate cleanup code

Warning

The cleanup mechanism is implemented using the `__cleanup` attribute. If the toolchain doesn't support this attribute, the API is not available.

For this reason this API is intended solely for user applications and not for the kernel itself or other subsystems.

3.13.2 Core Concepts

The cleanup API provides three main abstractions:

Scoped Variables

A scoped variable defines a type with automatic init and exit behavior. Variables declared with a scoped variable type are initialized using an init function and automatically cleaned up using an exit function when they go out of scope.

Scoped Guards

Scoped guards are specialized scoped variables that automatically acquire a lock or resource on initialization and release it when going out of scope. This pattern is commonly used with mutexes, semaphores, and other synchronization primitives.

Scoped Defers

Scoped defers execute a specified function when the variable goes out of scope, without acquiring anything on initialization. This is similar to the `defer` statement in languages like Go.

3.13.3 Defining Scoped Types

Custom Scoped Variables

Use `SCOPE_VAR_DEFINE` to define a custom scoped variable type with init and exit functions:

```
static inline struct flash_area *flash_area_init(int area_id)
{
    struct flash_area *fa;

    if (flash_area_open(area_id, &fa) < 0) {
        return NULL;
    }

    return fa;
}

static inline void flash_area_exit(struct flash_area *fa)
{
    if (fa != NULL) {
        flash_area_close(fa);
    }
}

// Define the scoped variable type
SCOPE_VAR_DEFINE(flash_area, struct flash_area *, flash_area_exit(_T),
                 flash_area_init(area_id), int area_id);

static int some_function(void)
{
    // Declare 'fa' with automatic cleanup
    scope_var(flash_area, fa)(PARTITION_ID(storage_partition));
    if (fa == NULL) {
        return -EINVAL;  // Exit function is still called
    }

    // Use fa normally
    printk("Has driver: %d\n", flash_area_has_driver(fa));

    // No need to manually close - exit function is called automatically
    return 0;
}
```

The `_T` variable in the exit function expression contains the value of the variable being cleaned up.

Scoped Guards

Use `SCOPE_GUARD_DEFINE` to define a guard that acquires a lock on initialization and releases it on scope exit:

```
// Example guard definition (already provided by <zephyr/cleanup/kernel.h>)
SCOPE_GUARD_DEFINE(k_mutex, struct k_mutex *,
```

(continues on next page)

(continued from previous page)

```
                    (void)k_mutex_lock(_T, K_FOREVER),
                    (void)k_mutex_unlock(_T));

static K_MUTEX_DEFINE(lock);

void critical_section(void)
{
    scope_guard(k_mutex)(&lock);

    // Lock is held here
    // Perform critical operations

    // Lock is automatically released when guard goes out of scope
}
```

Block-Scoped Guards

While `scope_guard` holds a guard until the end of the *enclosing* scope, `scoped_guard` binds the guard to the brace block that immediately follows, releasing it as soon as that block is exited. This makes short critical sections explicit and keeps the lock object next to the code it protects:

```
static K_MUTEX_DEFINE(lock);

void worker(void)
{
    // ... work that does not need the lock ...

    scoped_guard(k_mutex, &lock) {
        // lock held only inside these braces
    }
    // lock released here

    // ... more work without the lock held ...
}
```

The block runs exactly once. The lock is released on any exit from the block, including `break`, `return` and `goto`. Note that `continue` leaves the block (it behaves like `break`) rather than re-running it.

Conditional Guards

Guards defined with `SCOPE_GUARD_DEFINE` always acquire the lock (they block with `K_FOREVER`). To express a guard whose acquisition may fail (for example a non-blocking `K_NO_WAIT` try-lock), use `SCOPE_COND_GUARD_DEFINE` together with `scoped_cond_guard`. The acquire expression is evaluated for success: on failure the guard stores `NULL`, the supplied fail statement runs, and the block is skipped.

```
// Example guard definition (already provided by <zephyr/cleanup/kernel.h>)
SCOPE_COND_GUARD_DEFINE(k_mutex_try, struct k_mutex *,
                        k_mutex_lock(_T, K_NO_WAIT) == 0,
                        (void)k_mutex_unlock(_T));

static K_MUTEX_DEFINE(lock);

int try_critical_section(void)
{
```

(continues on next page)

(continued from previous page)

```
    scoped_cond_guard(k_mutex_try, return -EBUSY, &lock) {
        // runs only if the lock was acquired
        // released automatically when the block is exited
    }

    return 0;
}
```

The fail statement can be any statement, such as `break`, `return -EBUSY`, or `{}` to silently skip the block.

Scoped Defers

Use `SCOPE_DEFER_DEFINE` to define a defer that executes a cleanup function:

```
// Define a defer for a custom cleanup function
static void cleanup_resources(void)
{
    // Cleanup code here
}

SCOPE_DEFER_DEFINE(cleanup_resources);

void some_function(void)
{
    scope_defer(cleanup_resources)();

    // Do work...

    // cleanup_resources() is called automatically
}
```

For functions with parameters:

```
// Example deferred k_free (already provided by <zephyr/cleanup/kernel.h>)
SCOPE_DEFER_DEFINE(k_free, void *);

void allocate_and_use(void)
{
    void *ptr = k_malloc(100);
    scope_defer(k_free)(ptr);

    // Use ptr...

    // k_free(ptr) is called automatically
}
```

3.13.4 Usage Notes

Order of Cleanup

Cleanup functions are called in reverse order of declaration (LIFO - Last In, First Out), which matches the natural nesting of resources:

```
{
    scope_guard(k_mutex)(&lock);           // Acquired first
```

(continues on next page)

(continued from previous page)

```
    void *ptr = k_malloc(100);
    scope_defer(k_free)(ptr);              // Registered second

    // Do work...

}  // ptr is freed first, then mutex is unlocked
```

Scope Rules

Cleanup occurs when the variable goes out of scope, which includes:

- Reaching the end of a block
- Early return statements
- Break or continue in loops
- Goto statements that jump out of scope

```
void example_with_early_exit(struct k_mutex *lock)
{
    scope_guard(k_mutex)(lock);

    if (error_condition) {
        return;  // Guard cleanup happens here
    }

    // Normal path

}  // Guard cleanup also happens here
```

3.13.5 API Reference

Cleanup Helper Interface

Chapter 4

Services

Zephyr provides a comprehensive set of subsystems and associated services that applications can leverage. These services expose standardized APIs that abstract away hardware implementation details, making applications portable across different platforms.

Connectivity

4.1 Connectivity

4.1.1 Networking

The networking section contains information regarding the network stack of the Zephyr kernel. Use the information to understand the principles behind the operation of the stacks and how they were implemented.

Overview

- *Supported Features*
- *Source Tree Layout*

Supported Features The networking IP stack is modular and configured at build time. Enable only the features your application needs to minimize memory use; almost all of them can be disabled.

- **IPv6** (**RFC 8200**), with build-time configurable sub-features:
  - Number of unicast and multicast addresses active at the same time.
  - Static addresses or SLAAC (Stateless Address Autoconfiguration, **RFC 4862**).
  - Multiple prefixes, up to a configurable maximum.
  - Optional neighbor cache of configurable size.
  - Neighbor Discovery (**RFC 4861**) and Multicast Listener Discovery v2 (**RFC 3810**), enabled by default.
  - 6LoWPAN header compression for IEEE 802.15.4 networks (**RFC 4944**).
  - DHCPv6 (**RFC 8415**) client and server, including prefix delegation. See *DHCPv6*.
  - DNS server addresses from Router Advertisements (RDNSS, **RFC 8106**).

  - Privacy extensions (**RFC 8981**).
  - Fragmentation and Path MTU Discovery (**RFC 8201**).
  - Multicast routing and forwarding.
- **IPv4** (**RFC 791**), for example on Ethernet, Wi-Fi and cellular networks. IEEE 802.15.4 is IPv6 only.
  - DHCPv4 (**RFC 2131**) client and server. See *DHCPv4*.
  - Static addresses, enabled by default.
  - Link-local address autoconfiguration (**RFC 3927**) and address conflict detection (**RFC 5227**). Sample: ipv4-autoconf.
  - IGMPv2 (**RFC 2236**) and IGMPv3 (**RFC 3376**).
  - Fragmentation and Path MTU Discovery (**RFC 1191**).
  - NAT (Network Address Translation): SNAT and DNAT between interfaces, with connection tracking and iptables-like rules to filter and forward packets across subnets.
- **Dual stack.** IPv6 and IPv4 can be used at the same time.
- **UDP** (**RFC 768**), for sending datagrams and listening on a port, and UDP options (**RFC 9868**). Datagram Packetization Layer Path MTU Discovery (DPLPMTUD, **RFC 8899**) for UDP-based protocols, and for plain UDP sockets through UDP options (**RFC 9869**). See *DPLPMTUD*.
- **TCP** (**RFC 9293**), client and server, with a build-time configurable number of sockets. Selective acknowledgment (**RFC 2018**) of received data with `CONFIG_NET_TCP_SACK`.
- **QUIC** (**RFC 9000**) with integrated TLS 1.3 (**RFC 9001**). See *QUIC*. Samples: quic-client-echo, quic-service-echo.
- **BSD Sockets API**, a subset of the *BSD sockets API*: blocking and non-blocking datagram (UDP) and stream (TCP) sockets, and packet sockets (`AF_PACKET`).
- **Secure Sockets API**, TLS and DTLS for the sockets API, backed by Mbed TLS. See *secure sockets*.
- **MQTT** (ISO/IEC PRF 20922) versions 3.1.1 and 5.0. Sample: mqtt-publisher.
- **MQTT-SN** version 1.2. Sample: mqtt-sn-publisher.
- **CoAP** (**RFC 7252**), with block-wise transfers (**RFC 7959**), resource observation (**RFC 7641**), OSCORE (**RFC 8613**), and CoAP over TCP and TLS (**RFC 8323**) for the client. Samples: coap-client, coap-server.
- **LwM2M** OMA Lightweight Machine-to-Machine Protocol (LwM2M specification 1.0.2, and LwM2M specification 1.1.1 through a Kconfig option): Bootstrap, Client Registration, Device Management & Service Enablement and Information Reporting interfaces, the required core objects and several IPSO Smart Objects. Sample: lwm2m-client.
- **HTTP** *client* for HTTP/1.1 (**RFC 2616**) and *server* for HTTP/1.1, HTTP/2 (**RFC 9113**) and HTTP/3 (**RFC 9114**). Samples: sockets-http-client, sockets-http-server.
- **Websocket** (**RFC 6455**) client, and Websocket upgrade in the HTTP server. Sample: sockets-websocket-client.
- **DNS** (**RFC 1035**) resolver for IPv4 (A) and IPv6 (AAAA) records, multicast DNS (mDNS, **RFC 6762**), DNS Service Discovery (DNS-SD, **RFC 6763**) and the deprecated LLMNR (**RFC 4795**). An mDNS responder answers queries and advertises DNS-SD services. Sample: mdns-responder.
- **Network Management API** to receive events from the network stack, for example when an IP address is added or an interface comes up.
- **Wi-Fi Management API** to manage Wi-Fi interfaces, for example to scan for and connect to networks.

- **Wi-Fi Network Manager API** for network managers that register with the Wi-Fi stack and implement the Wi-Fi Management API.
- **Multiple network technologies**, such as Ethernet, Wi-Fi and IEEE 802.15.4, enabled at the same time in Kconfig. No automatic routing is done between them; applications send data to the interface they choose.
- **Minimal copy buffer management.** The stack avoids copying application data on the transmit path.
- **Virtual LANs** partition physical Ethernet networks into logical networks. See *VLAN*.
- **Traffic classification** prioritizes sent and received packets. See *traffic classification*.
- **Time Sensitive Networking** with gPTP (generalized Precision Time Protocol) and PTP (IEEE 1588). See *gPTP* and *PTP*.
- **SNTP** (**RFC 5905**) client and server. See *SNTP*. Samples: sntp-client, sntp-server.
- **SOCKS5** proxy (**RFC 1928**).
- **TFTP** (**RFC 1350**) client. Sample: tftp-client.
- **FTP** (**RFC 959**) client. See *FTP client*. Sample: ftp-client.
- **SSH** SSH-2 (**RFC 4251** to **RFC 4254**) server and client. Session channels carry shell, exec and subsystem requests, and the Zephyr shell is available over SSH. Sample: ssh-server-client.
- **WireGuard** VPN (WireGuard protocol). Sample: wireguard-vpn.
- **RTP** Real-time Transport Protocol (**RFC 3550**). Sample: net-rtp.
- **MIDI2** MIDI 2.0 UDP network transport. Sample: netmidi2.
- **OCPP** Open Charge Point Protocol. Sample: ocpp.
- **Prometheus** metrics server. Sample: prometheus.
- **MCP** Model Context Protocol server, exposing device tools that AI agents discover and invoke over HTTP. See *MCP server*. Sample: mcp-server-hello-world.
- **Packet filtering and capture.** Filter sent and received packets with rules, and capture traffic to a remote host for analysis. See *packet filtering* and *network packet capture*.
- **Network shell** to inspect network status, enable or disable features and run commands such as ping or DNS queries. See *network shell*.
- **zperf** network performance and bandwidth measurement tool, client and server, compatible with iPerf v2 and iperf3. Sample: zperf.

Supported network technologies (link layers):

- IEEE 802.15.4
- Ethernet, IEEE 802.3
- Wi-Fi, IEEE 802.11
- Cellular / PPP (**RFC 1661**)
- Thread (samples: openthread)
- CAN bus for SocketCAN
- SLIP (IP over serial line), used to connect QEMU targets to the host as an Ethernet interface so that host applications can exchange data with Zephyr.

Source Tree Layout The networking stack source code tree is organized as follows:

subsys/net/
Various optional network stack components like connection manager, packet filter code and hostname handling are located here.

subsys/net/ip/
This is where the core network stack code is located.

subsys/net/l2
This is where the IP stack layer 2 code is located. This includes generic support for Ethernet, IEEE 802.15.4 and Wi-Fi.

subsys/net/lib/
Application-level protocols (DNS, MQTT, etc.) and additional stack components (BSD Sockets, etc.).

include/zephyr/net/
Public API header files. These are the header files applications need to include to use IP networking functionality.

samples/net/
Sample networking code. This is a good reference to get started with network application development.

tests/net/
Test applications. These applications are used to verify the functionality of the IP stack, but are not the best source for sample code (see samples/net/ instead).

Network Stack Architecture

Network Packet Processing Statistics This page describes how to get information about network packet processing statistics inside network stack.

Network stack contains infrastructure to figure out how long the network packet processing takes either in sending or receiving path. There are two Kconfig options that control this. For transmit (TX) path the option is called `CONFIG_NET_PKT_TXTIME_STATS` and for receive (RX) path the options is called `CONFIG_NET_PKT_RXTIME_STATS`. Note that for TX, all kind of network packet statistics is collected. For RX, only UDP, TCP or raw packet type network packet statistics is collected.

After enabling these options, the *net stats* network shell command will show this information:

```
Avg TX net_pkt (11484) time 67 us
Avg RX net_pkt (11474) time 43 us
```

> **Note**
> The values above and below are from emulated qemu_x86 board and UDP traffic

The TX time tells how long it took for network packet from its creation to when it was sent to the network. The RX time tells the time from its creation to when it was passed to the application. The values are in microseconds. The statistics will be collected per traffic class if there are more than one transmit or receive queues defined in the system. These are controlled by `CONFIG_NET_TC_TX_COUNT` and `CONFIG_NET_TC_RX_COUNT` options.

If you enable `CONFIG_NET_PKT_TXTIME_STATS_DETAIL` or `CONFIG_NET_PKT_RXTIME_STATS_DETAIL` options, then additional information for TX or RX network packets are collected when the network packet traverses the IP stack.

After enabling these options, the *net stats* will show this information:

```
Avg TX net_pkt (18902) time 63 us    [0->22->15->23=60 us]
Avg RX net_pkt (18892) time 42 us    [0->9->6->11->13=39 us]
```

The numbers inside the brackets contain information how many microseconds it took for a network packet to go from previous state to next.

In the TX example above, the values are averages over **18902** packets and contain this information:

- Packet was created by application so the time is **0**.
- Packet is about to be placed to transmit queue. The time it took from network packet creation to this state, is **22** microseconds in this example.
- The correct TX thread is invoked, and the packet is read from the transmit queue. It took **15** microseconds from previous state.
- The network packet was just sent and the network stack is about to free the network packet. It took **23** microseconds from previous state.
- In total it took on average **60** microseconds to get the network packet sent. The value **63** tells also the same information, but is calculated differently so there is slight difference because of rounding errors.

In the RX example above, the values are averages over **18892** packets and contain this information:

- Packet was created network device driver so the time is **0**.
- Packet is about to be placed to receive queue. The time it took from network packet creation to this state, is **9** microseconds in this example.
- The correct RX thread is invoked, and the packet is read from the receive queue. It took **6** microseconds from previous state.
- The network packet is then processed and placed to correct socket queue. It took **11** microseconds from previous state.
- The last value tells how long it took from there to the application. Here the value is **13** microseconds.
- In total it took on average **39** microseconds to get the network packet sent. The value **42** tells also the same information, but is calculated differently so there is slight difference because of rounding errors.

The Zephyr network stack is a native network stack specifically designed for Zephyr OS. It consists of layers, each meant to provide certain services to other layers. Network stack functionality is highly configurable via Kconfig options.

- *High level overview of the network stack*
- *Network data flow*
  - *Data receiving (RX)*
  - *Data sending (TX)*
- *Network packet processing statistics*

High level overview of the network stack The network stack is layered and consists of the following parts:

- **Network Application.** The network application can either use the provided application-level protocol libraries or access the *BSD socket API* directly to create a network connection,

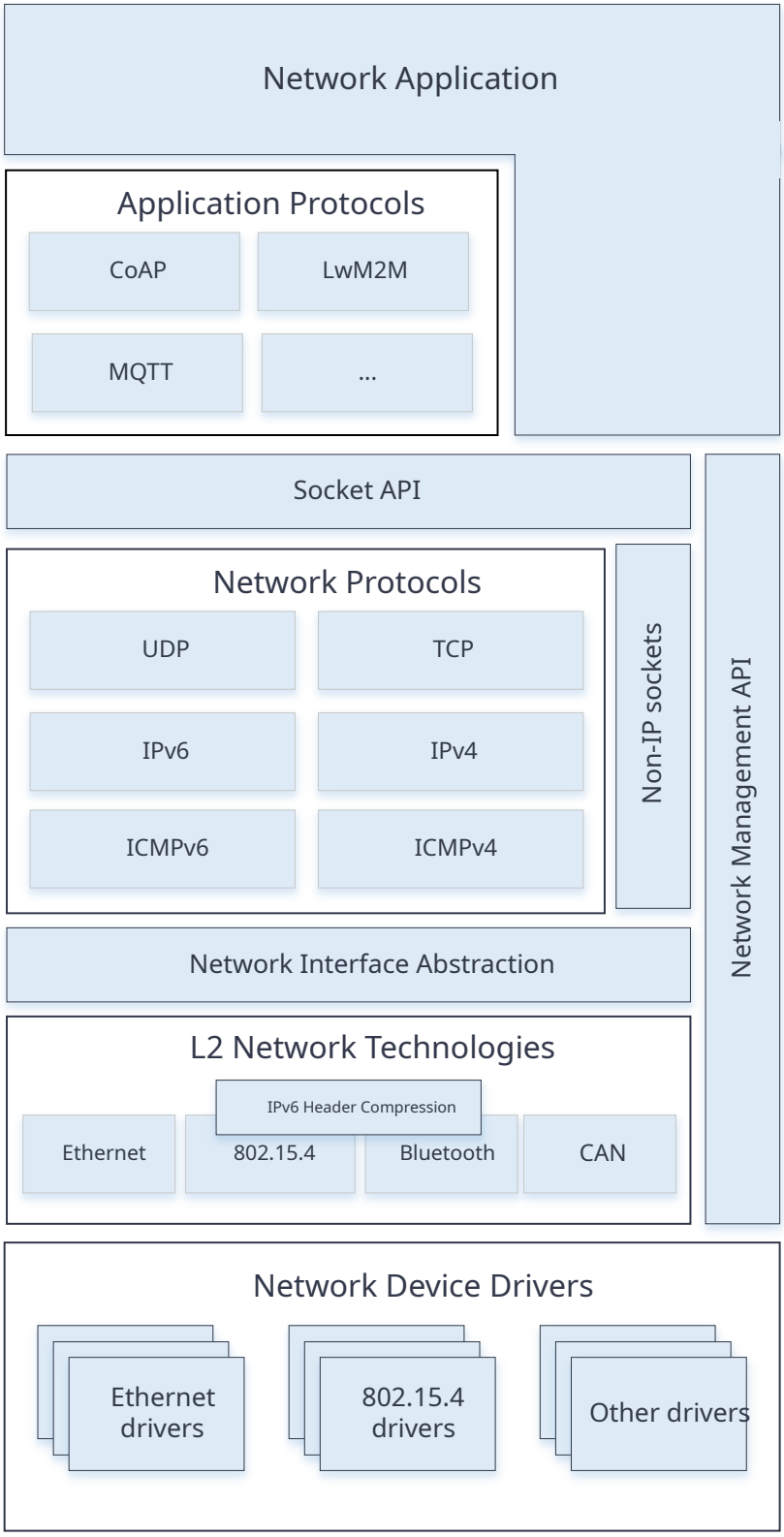


Fig. 1: Network stack overview

send or receive data, and close a connection. The application can also use the *network management API* to configure the network and set related parameters such as network link options, starting a scan (when applicable), listen network configuration events, etc. The *network interface API* can be used to set IP address to a network interface, taking the network interface down, etc.

- **Network Protocols.** This provides implementations for various protocols such as
  - Application-level network protocols like CoAP, LwM2M, and MQTT. See *application protocols chapter* for information about them.
  - Core network protocols like IPv6, IPv4, UDP, TCP, ICMPv4, and ICMPv6. You access these protocols by using the *BSD socket API*.
- **Network Interface Abstraction.** This provides functionality that is common in all the network interfaces, such as setting network interface down, etc. There can be multiple network interfaces in the system. See *network interface overview* for more details.
- **L2 Network Technologies.** This provides a common API for sending and receiving data to and from an actual network device. See *L2 overview* for more details.

  These network technologies include *Ethernet*, *IEEE 802.15.4*, *Bluetooth*, *CANBUS*, etc.

  Some of these technologies support IPv6 header compression (6Lo), see **RFC 6282** for details. For example ARP (**RFC 826**) for IPv4 is done by the *Ethernet component*.
- **Network Device Drivers.** The actual low-level device drivers handle the physical sending or receiving of network packets.

Network data flow An application typically consists of one or more *threads* that execute the application logic. When using the *BSD socket API*, the following things will happen.

Data receiving (RX)

1. A network data packet is received by a device driver.
2. The device driver allocates enough network buffers to store the received data. The network packet is placed in the proper RX queue (implemented by *k_fifo*). By default there is only one receive queue in the system, but it is possible to have up to 8 receive queues. These queues will process incoming packets with different priority. See *Traffic Classification* for more details. The receive queues also act as a way to separate the data processing pipeline (bottom-half) as the device driver is running in an interrupt context and it must do its processing as fast as possible.
3. The network packet is then passed to the correct L2 driver. The L2 driver can check if the packet is proper and modify it if needed, e.g. strip L2 header and frame check sequence, etc.
4. The packet is processed by a network interface. The network statistics are collected if enabled by `CONFIG_NET_STATISTICS`.
5. The packet is then passed to L3 processing. If the packet is IP based, then the L3 layer checks if the packet is a proper IPv6 or IPv4 packet.
6. A socket handler then finds an active socket to which the network packet belongs and puts it in a queue for that socket, in order to separate the networking code from the application. Typically the application is run in userspace context and the network stack is run in kernel context.
7. The application will then receive the data and can process it as needed. The application should have used the *BSD socket API* to create a socket that will receive the data.

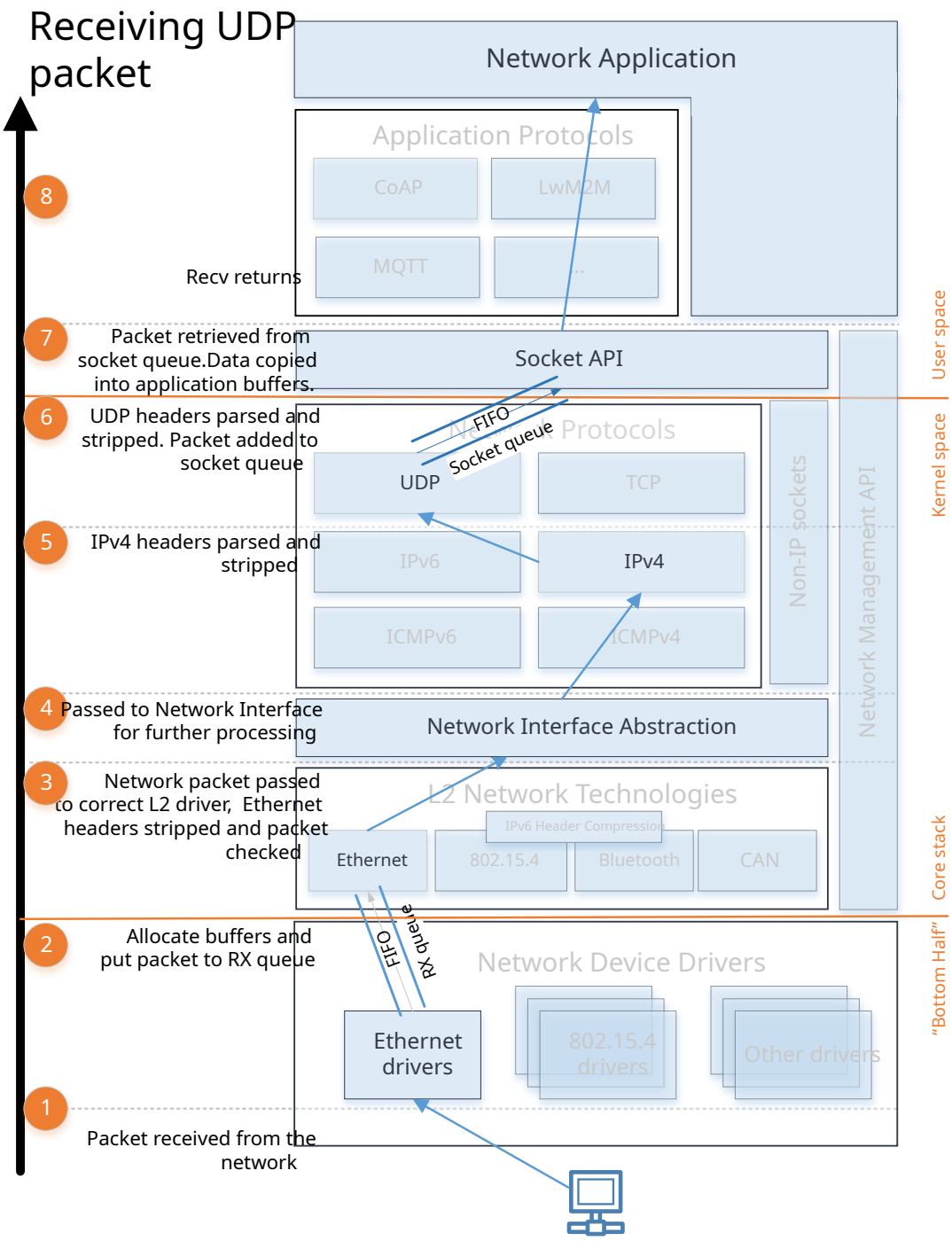


Fig. 2: Network RX data flow

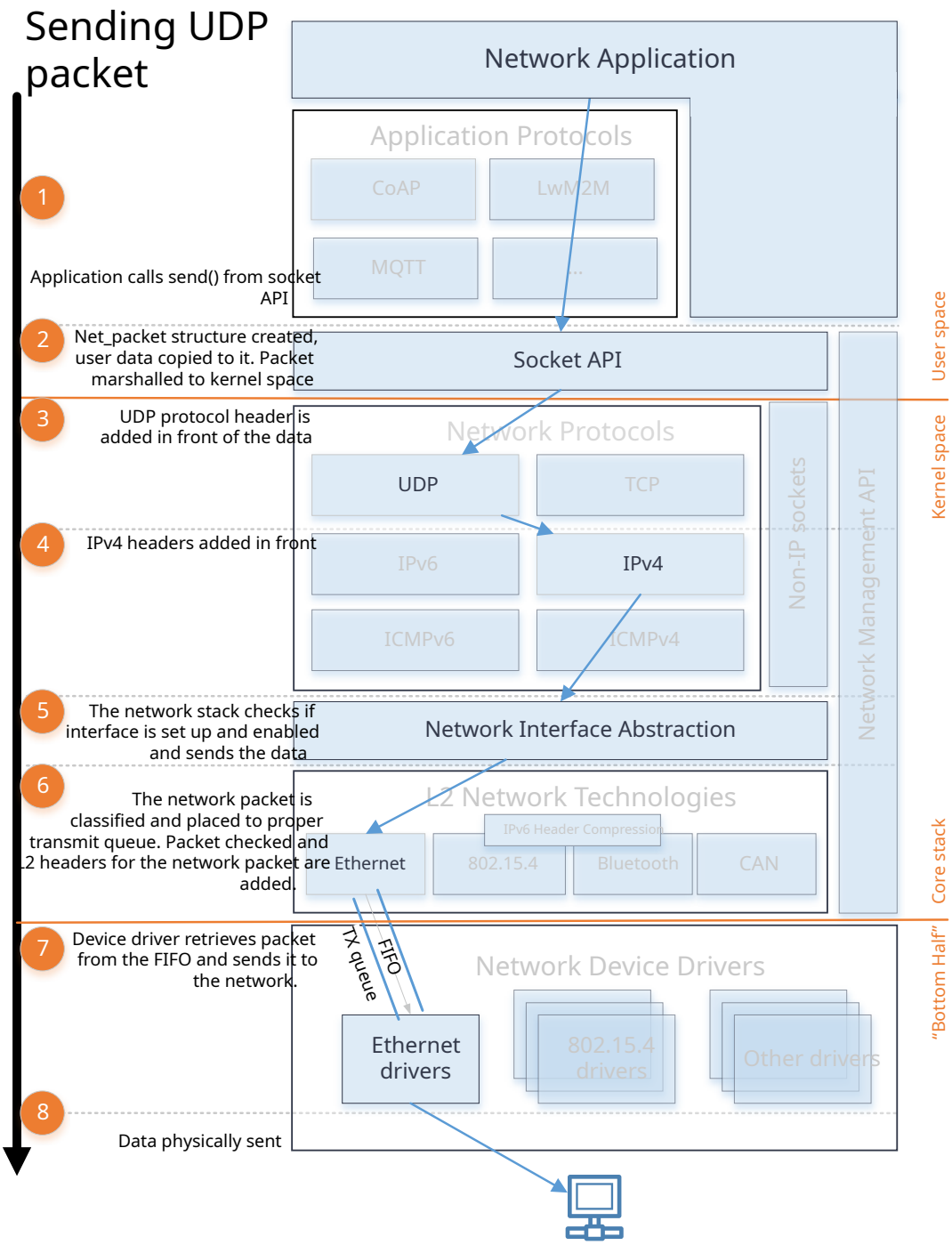


Fig. 3: Network TX data flow

Data sending (TX)

1. The application should use the *BSD socket API* when sending the data.
2. The application data is prepared for sending to kernel space and then copied to internal net_buf structures.
3. Depending on the socket type, a protocol header is added in front of the data. For example, if the socket is a UDP socket, then a UDP header is constructed and placed in front of the data.
4. An IP header is added to the network packet for a UDP or TCP packet.
5. The network stack will check that the network interface is properly set for the network packet, and also will make sure that the network interface is enabled before the data is queued to be sent.
6. The network packet is then classified and placed to the proper transmit queue (implemented by *k_fifo*). By default there is only one transmit queue in the system, but it is possible to have up to 8 transmit queues. These queues will process the sent packets with different priority. See *Traffic Classification* for more details. After the transmit packet classification, the packet is checked by the correct L2 layer module. The L2 module will do additional checks for the data and it will also create any L2 headers for the network packet. If everything is ok, the data is given to the network device driver to be sent out.
7. The device driver will send the packet to the network.

Note that in both the TX and RX data paths, the queues (*k_fifo's*) form separation points where data is passed from one *thread* to another. These *threads* might run in different contexts (*kernel* vs. *userspace*) and with different *priorities*.

Network packet processing statistics See information about network processing statistics *here*.

Network Configuration Guide

- *Network Buffer Configuration Options*
- *Connection Options*
- *Socket Options*
- *TLS Options*
- *IPv4/6 Options*
- *TCP Options*
- *Traffic Class Options*
- *Stack Size Options*

This document describes how various network configuration options can be set according to available resources in the system.

Network Buffer Configuration Options The network buffer configuration options control how much data we are able to either send or receive at the same time.

`CONFIG_NET_PKT_RX_COUNT`
: Maximum amount of network packets we can receive at the same time.

`CONFIG_NET_PKT_TX_COUNT`
: Maximum amount of network packet sends pending at the same time.

`CONFIG_NET_BUF_RX_COUNT`
How many network buffers are allocated for receiving data. Each net_buf contains a small header and either a fixed or variable length data buffer. The `CONFIG_NET_BUF_DATA_SIZE` is used when `CONFIG_NET_BUF_FIXED_DATA_SIZE` is set. This is the default setting. The default size of the buffer is 128 bytes.

The `CONFIG_NET_BUF_VARIABLE_DATA_SIZE` is an experimental setting. There each net_buf data portion is allocated from a memory pool and can be the amount of data we have received from the network. When data is received from the network, it is placed into net_buf data portion. Depending on device resources and desired network usage, user can tweak the size of the fixed buffer by setting `CONFIG_NET_BUF_DATA_SIZE`, and the size of the data pool size by setting `CONFIG_NET_PKT_BUF_RX_DATA_POOL_SIZE` and `CONFIG_NET_PKT_BUF_TX_DATA_POOL_SIZE` if variable size buffers are used.

When using the fixed size data buffers, the memory consumption of network buffers can be tweaked by selecting the size of the data part according to what kind of network data we are receiving. If one sets the data size to 256, but only receives packets that are 32 bytes long, then we are "wasting" 224 bytes for each packet because we cannot utilize the remaining data. One should not set the data size too low because there is some overhead involved for each net_buf. For these reasons the default network buffer size is set to 128 bytes.

The variable size data buffer feature is marked as experimental as it has not received as much testing as the fixed size buffers. Using variable size data buffers tries to improve memory utilization by allocating minimum amount of data we need for the network data. The extra cost here is the amount of time that is needed when dynamically allocating the buffer from the memory pool.

For example, in Ethernet the maximum transmission unit (MTU) size is 1500 bytes. If one wants to receive two full frames, then the net_pkt RX count should be set to 2, and net_buf RX count to (1500 / 128) * 2 which is 24. If TCP is being used, then these values need to be higher because we can queue the packets internally before delivering to the application.

`CONFIG_NET_BUF_TX_COUNT`
How many network buffers are allocated for sending data. This is similar setting as the receive buffer count but for sending.

Connection Options

`CONFIG_NET_MAX_CONN`
This option tells how many network connection endpoints are supported. For example each TCP connection requires one connection endpoint. Similarly each listening UDP connection requires one connection endpoint. Also various system services like DHCP and DNS need connection endpoints to work. The network shell command **net conn** can be used at runtime to see the network connection information.

`CONFIG_NET_MAX_CONTEXTS`
Number of network contexts to allocate. Each network context describes a network 5-tuple that is used when listening or sending network traffic. Each BSD socket in the system uses one network context.

Socket Options

`CONFIG_ZVFS_POLL_MAX`
Maximum number of supported poll() entries. One needs to select proper value here depending on how many BSD sockets are polled in the system.

`CONFIG_ZVFS_OPEN_MAX`
Maximum number of open file descriptors, this includes files, sockets, special devices, etc. One needs to select proper value here depending on how many BSD sockets are created in the system.

`CONFIG_ZVFS_EVENTFD_MAX`
Maximum number of ZVFS eventfd's. Several networking subsystems (e.g. the HTTP server, CoAP server, LwM2M engine, PTP, SSH and the socket service) allocate an eventfd to wake up their poll loop. Each such subsystem declares its requirement through a `CONFIG_ZVFS_EVENTFD_ADD_SIZE_*` option, and the actual eventfd count is the larger of this option and the sum of all those requirements. Only set this option explicitly if an application creates additional eventfds on its own.

`CONFIG_NET_SOCKETPAIR_BUFFER_SIZE`
This option is used by socketpair() function. It sets the size of the internal intermediate buffer, in bytes. This sets the limit how large messages can be passed between two socketpair endpoints.

TLS Options

`CONFIG_NET_SOCKETS_TLS_MAX_CONTEXTS`
Maximum number of TLS/DTLS contexts. Each TLS/DTLS connection needs one context.

`CONFIG_NET_SOCKETS_TLS_MAX_CREDENTIALS`
This variable sets maximum number of TLS/DTLS credentials that can be used with a specific socket.

`CONFIG_NET_SOCKETS_TLS_MAX_CIPHERSUITES`
Maximum number of TLS/DTLS ciphersuites per socket. This variable sets maximum number of TLS/DTLS ciphersuites that can be used with specific socket, if set explicitly by socket option. By default, all ciphersuites that are available in the system are available to the socket.

`CONFIG_NET_SOCKETS_TLS_MAX_APP_PROTOCOLS`
Maximum number of supported application layer protocols. This variable sets maximum number of supported application layer protocols over TLS/DTLS that can be set explicitly by a socket option. By default, no supported application layer protocol is set.

`CONFIG_NET_SOCKETS_TLS_MAX_CLIENT_SESSION_COUNT`
This variable specifies maximum number of stored TLS/DTLS sessions, used for TLS/DTLS session resumption.

`CONFIG_TLS_MAX_CREDENTIALS_NUMBER`
Maximum number of TLS credentials that can be registered. Make sure that this value is high enough so that all the certificates can be loaded to the store.

IPv4/6 Options

`CONFIG_NET_IF_MAX_IPV4_COUNT`
Maximum number of IPv4 network interfaces in the system. This tells how many network interfaces there will be in the system that will have IPv4 enabled. For example if you have two network interfaces, but only one of them can use IPv4 addresses, then this value can be set to 1. If both network interface could use IPv4, then the setting should be set to 2.

`CONFIG_NET_IF_MAX_IPV6_COUNT`
Maximum number of IPv6 network interfaces in the system. This is similar setting as the IPv4 count option but for IPv6.

TCP Options

`CONFIG_NET_TCP_TIME_WAIT_DELAY`
How long to wait in TCP *TIME_WAIT* state (in milliseconds). To avoid a (low-probability) issue when delayed packets from previous connection get delivered to next connection reusing the same local/remote ports, RFC 793 (TCP) suggests to keep an old, closed connection in a special *TIME_WAIT* state for the duration of 2*MSL (Maximum Segment Lifetime). The RFC suggests to use MSL of 2 minutes, but notes

This is an engineering choice, and may be changed if experience indicates it is desirable to do so.

For low-resource systems, having large MSL may lead to quick resource exhaustion (and related DoS attacks). At the same time, the issue of packet misdelivery is largely alleviated in the modern TCP stacks by using random, non-repeating port numbers and initial sequence numbers. Due to this, Zephyr uses much lower value of 1500ms by default. Value of 0 disables *TIME_WAIT* state completely.

`CONFIG_NET_TCP_RETRY_COUNT`
Maximum number of TCP segment retransmissions. The following formula can be used to determine the time (in ms) that a segment will be buffered awaiting retransmission:

$$\sum_{n=0}^{\texttt{NET_TCP_RETRY_COUNT}} \left(1 \ll n\right) \times \texttt{NET_TCP_INIT_RETRANSMISSION_TIMEOUT}$$

With the default value of 9, the IP stack will try to retransmit for up to 1:42 minutes. This is as close as possible to the minimum value recommended by RFC 1122 (1:40 minutes). Only 5 bits are dedicated for the retransmission count, so accepted values are in the 0-31 range. It's highly recommended to not go below 9, though.

Should a retransmission timeout occur, the receive callback is called with `-ETIMEDOUT` error code and the context is dereferenced.

`CONFIG_NET_TCP_MAX_SEND_WINDOW_SIZE`
Maximum sending window size to use. This value affects how the TCP selects the maximum sending window size. The default value 0 lets the TCP stack select the value according to amount of network buffers configured in the system. Note that if there are multiple active TCP connections in the system, then this value might require finetuning (lowering), otherwise multiple TCP connections could easily exhaust net_buf pool for the queued TX data.

`CONFIG_NET_TCP_MAX_RECV_WINDOW_SIZE`
Maximum receive window size to use. This value defines the maximum TCP receive window size. Increasing this value can improve connection throughput, but requires more receive buffers available in the system for efficient operation. The default value 0 lets the TCP stack select the value according to amount of network buffers configured in the system.

`CONFIG_NET_TCP_RECV_QUEUE_TIMEOUT`
How long to queue received data (in ms). If we receive out-of-order TCP data, we queue it. This value tells how long the data is kept before it is discarded if we have not been able to pass the data to the application. If set to 0, then receive queueing is not enabled. The value is in milliseconds.

Note that we only queue data sequentially in current version i.e., there should be no holes in the queue. For example, if we receive SEQs 5,4,3,6 and are waiting SEQ 2, the data in segments 3,4,5,6 is queued (in this order), and then given to application when we receive SEQ 2. But if we receive SEQs 5,4,3,7 then the SEQ 7 is discarded because the list would not be sequential as number 6 is be missing.

Traffic Class Options It is possible to configure multiple traffic classes (queues) when receiving or sending network data. Each traffic class queue is implemented as a thread with different priority. This means that higher priority network packet can be placed to a higher priority network queue in order to send or receive it faster or slower. Because of thread scheduling latencies, in practice the fastest way to send a packet out, is to directly send the packet without using a dedicated traffic class thread. This is why by default the `CONFIG_NET_TC_TX_COUNT` option is set to 0 if userspace is not enabled. If userspace is enabled, then the minimum TX traffic class count is 1. Reason for this is that the userspace application does not have enough permissions to deliver the message directly.

In receiving side, it is recommended to have at least one receiving traffic class queue. Reason is that typically the network device driver is running in IRQ context when it receives the packet,

in which case it should not try to deliver the network packet directly to the upper layers, but to place the packet to the traffic class queue. If the network device driver is not running in IRQ context when it gets the packet, then the RX traffic class option `CONFIG_NET_TC_RX_COUNT` could be set to 0.

Stack Size Options There several network specific threads in a network enabled system. Some of the threads might depend on a configure option which can be used to enable or disable a feature. Each thread stack size is optimized to allow normal network operations.

The network management API is using a dedicated thread by default. The thread is responsible to deliver network management events to the event listeners that are setup in the system if the `CONFIG_NET_MGMT` and `CONFIG_NET_MGMT_EVENT` options are enabled. If the options are enabled, the user is able to register a callback function that the net_mgmt thread is calling for each network management event. By default the net_mgmt event thread stack size is rather small. The idea is that the callback function does minimal things so that new events can be delivered to listeners as fast as possible and they are not lost. The net_mgmt event thread stack size is controlled by `CONFIG_NET_MGMT_EVENT_QUEUE_SIZE` option. It is recommended to not do any blocking operations in the callback function.

The network thread stack utilization can be monitored from kernel shell by the **kernel threads** command.

Networking with the host system

Networking with native_sim board

- *Using virtual/TAP Ethernet driver*
  - *Prerequisites*
  - *Basic Setup*
- *Using offloaded sockets*
  - *Step 1 - Start app in native_sim board*
  - *Step 2 - run echo-client from net-tools*
- *Setting interface name and IPv4 parameters from command line*

Using virtual/TAP Ethernet driver This paragraph describes how to set up a virtual network between a (Linux) host and a Zephyr application running in a native_sim board.

In this example, the sockets-echo-server sample application from the Zephyr source distribution is run in native_sim board. The Zephyr native_sim board instance is connected to a Linux host using a tuntap device which is modeled in Linux as an Ethernet network interface.

Prerequisites On the Linux Host, find the Zephyr net-tools project, which can either be found in a Zephyr standard installation under the `tools/net-tools` directory or installed stand alone from its own git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need three terminal windows:

- Terminal #1 is terminal window with net-tools being the current directory (`cd net-tools`)
- Terminal #2 is your usual Zephyr development terminal, with the Zephyr environment initialized.

- Terminal #3 is the console to the running Zephyr native_sim instance (optional).

Step 1 - Create Ethernet interface Before starting native_sim with network emulation, a network interface should be created.

In terminal #1, type:

```
./net-setup.sh
```

You can tweak the behavior of the net-setup.sh script. See various options by running `net-setup.sh` like this:

```
./net-setup.sh --help
```

Step 2 - Start app in native_sim board Build and start the `echo_server` sample application.

In terminal #2, type:

```
west build -b native_sim samples/net/sockets/echo_server
west build -t run
```

Step 3 - Connect to console (optional) The console window should be launched automatically when the Zephyr instance is started but if it does not show up, you can manually connect to the console. The native_sim board will print a string like this when it starts:

```
UART connected to pseudotty: /dev/pts/5
```

You can manually connect to it like this:

```
screen /dev/pts/5
```

Using offloaded sockets The main advantage over *Using virtual/TAP Ethernet driver* is not needing to setup a virtual network interface on the host machine. This means that no leveraged (root) privileges are needed.

Step 1 - Start app in native_sim board Build and start the `echo_server` sample application:

```
west build -b native_sim samples/net/sockets/echo_server -- -
↪DEXTRA_CONF_FILE=overlay-nsos.conf
west build -t run
```

Step 2 - run echo-client from net-tools On the Linux Host, find the Zephyr net-tools project, which can either be found in a Zephyr standard installation under the `tools/net-tools` directory or installed stand alone from its own git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

> **Note**
>
> Native Simulator with the offloaded sockets network driver is using the same network interface/namespace as any other (Linux) application that uses BSD sockets API. This means that sockets-echo-server and `echo-client` applications will communicate over localhost/loopback interface (address `127.0.0.1`).

To run UDP test, type:

```
./echo-client 127.0.0.1
```

For TCP test, type:

```
./echo-client -t 127.0.0.1
```

Setting interface name and IPv4 parameters from command line By default the Ethernet interface name used by native_sim is determined by the `host-interface` property of the `zephyr,native-tap` devicetree node, but it is also possible to set it from the command line using `--eth-if=<interface_name>`.

The same applies to the IPv4 address, gateway and netmask. They can be set from the command line using `--ipv4-addr=<ip_address>`, `--ipv4-gw=<gateway>` and `--ipv4-nm=<netmask>`.

These options apply to the first interface. When multiple `zephyr,native-tap` interfaces are defined, each additional interface has its own per-interface options prefixed with the devicetree node name, e.g. `--<node>_eth-if` and `--<node>_ipv4-addr`.

Note that the configuration `CONFIG_NET_CONFIG_MY_IPV4_ADDR` and the command line arguments work in parallel. This means that if both are set, the interface might end up with two IP addresses. In most cases, it does make sense to only use one of both at the same time.

This can be useful if the application has to be run in multiple instances and recompiling it for each instance would be troublesome.

```
./zephyr.exe --eth-if=zeth2 --ipv4-addr=192.0.2.2 --ipv4-gw=192.0.0.1
--ipv4-nm=255.255.0.0
```

> **Note**
>
> These command line options can also be supplied at build time through the `CONFIG_NATIVE_EXTRA_CMDLINE_ARGS` configuration option.

Networking with QEMU Ethernet

- *Prerequisites*
- *Basic Setup*
  - *Step 1 - Create Ethernet interface*
  - *Step 2 - Start app in QEMU board*

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in QEMU.

In this example, the sockets-echo-server sample application from the Zephyr source distribution is run in QEMU. The Zephyr instance is connected to a Linux host using a tuntap device which is modeled in Linux as an Ethernet network interface.

Prerequisites On the Linux Host, find the Zephyr net-tools project, which can either be found in a Zephyr standard installation under the `tools/net-tools` directory or installed stand alone from its own git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need two terminal windows:

- Terminal #1 is terminal window with net-tools being the current directory (`cd net-tools`)
- Terminal #2 is your usual Zephyr development terminal, with the Zephyr environment initialized.

When configuring the Zephyr instance, you must select the correct Ethernet driver for QEMU connectivity:

- For `qemu_x86`, select `Intel(R) PRO/1000 Gigabit Ethernet driver` Ethernet driver. Driver is called `e1000` in Zephyr source tree.
- For `qemu_cortex_m3`, select `TI Stellaris MCU family ethernet driver` Ethernet driver. Driver is called `stellaris` in Zephyr source tree.
- For `mps2_an385`, select `SMSC911x/9220 Ethernet driver` Ethernet driver. Driver is called `smsc911x` in Zephyr source tree.
- For `qemu_cortex_a53`, `Intel(R) PRO/1000 Gigabit Ethernet driver` Ethernet driver is selected by default.
- Additionally, the sockets-echo-server sample contains overlay files for the VIRTIO Network device on `qemu_x86_64`.

Step 1 - Create Ethernet interface Before starting QEMU with network connectivity, a network interface should be created in the host system.

In terminal #1, type:

```
./net-setup.sh
```

You can tweak the behavior of the `net-setup.sh` script. See various options by running `net-setup.sh` like this:

```
./net-setup.sh --help
```

Step 2 - Start app in QEMU board Build and start the sockets-echo-server sample application. In this example, the qemu_x86 board is used.

In terminal #2, type:

```
west build -b qemu_x86 samples/net/sockets/echo_server -- -
↪DEXTRA_CONF_FILE=overlay-e1000.conf
west build -t run
```

Alternatively, if you decided to use the VIRTIO Network device on qemu_x86_64:

```
west build -b qemu_x86_64 samples/net/sockets/echo_server -- -
↪DDTC_OVERLAY_FILE=virtnet.overlay -DEXTRA_CONF_FILE=overlay-virtnet.conf
west build -t run
```

Exit QEMU by pressing `CTRL+A x`.

Networking with QEMU

- *Prerequisites*
- *Basic Setup*
  - *Step 1 - Create helper socket*

  - *Step 2 - Start TAP device routing daemon*
  - *Step 3 - Start app in QEMU*
  - *Step 4 - Run apps on host*
  - *Step 5 - Stop supporting daemons*
- *Setting up Zephyr and NAT/masquerading on host to access Internet*
- *Network connection between two QEMU VMs*
  - *Terminal #1:*
  - *Terminal #2:*
- *Running multiple QEMU VMs of the same sample*
  - *Terminal #1:*
  - *Terminal #2:*

This page describes how to set up a virtual network between a (Linux) host and a Zephyr application running in a QEMU virtual machine (built for Zephyr targets such as qemu_x86 and qemu_cortex_m3). Some virtual ARM boards (such as qemu_cortex_a53) only support a single UART, in this case QEMU Ethernet is preferred, see *Networking with QEMU Ethernet* for details.

In this example, the sockets-echo-server sample application from the Zephyr source distribution is run in QEMU. The QEMU instance is connected to a Linux host using a serial port, and SLIP is used to transfer data between the Zephyr application and Linux (over a chain of virtual connections).

Prerequisites On the Linux Host, find the Zephyr net-tools project, which can either be found in a Zephyr standard installation under the `tools/net-tools` directory or installed stand alone from its own git repository:

```
sudo apt install -y socat libpcap-dev
git clone https://github.com/zephyrproject-rtos/net-tools
cd net-tools
make
```

> **Note**
>
> If you get an error about AX_CHECK_COMPILE_FLAG, install package `autoconf-archive` package on Debian/Ubuntu.

Basic Setup For the steps below, you will need at least 4 terminal windows:

- Terminal #1 is your usual Zephyr development terminal, with the Zephyr environment initialized.
- Terminals #2, #3, and #4 are terminal windows with net-tools being the current directory (`cd net-tools`)

Step 1 - Create helper socket Before starting QEMU with network emulation, a Unix socket for the emulation should be created.

In terminal #2, type:

```
./loop-socat.sh
```

Step 2 - Start TAP device routing daemon In terminal #3, type:

```
sudo ./loop-slip-tap.sh
```

For applications requiring DNS, you may need to restart the host's DNS server at this point, as described in *Setting up Zephyr and NAT/masquerading on host to access Internet*.

Step 3 - Start app in QEMU Build and start the `echo_server` sample application.

In terminal #1, type:

```
west build -b qemu_x86 samples/net/sockets/echo_server
west build -t run
```

If you see an error from QEMU about unix:/tmp/slip.sock, it means you missed Step 1 above.

Step 4 - Run apps on host Now in terminal #4, you can run various tools to communicate with the application running in QEMU.

You can start with pings:

```
ping 192.0.2.1
ping6 2001:db8::1
```

You can use the netcat ("nc") utility, connecting using UDP:

```
echo foobar | nc -6 -u 2001:db8::1 4242
foobar
```

```
echo foobar | nc -u 192.0.2.1 4242
foobar
```

If echo_server is compiled with TCP support (now enabled by default for the echo_server sample, CONFIG_NET_TCP=y):

```
echo foobar | nc -6 -q2 2001:db8::1 4242
foobar
```

Note

Use Ctrl+C to exit.

You can also use the telnet command to achieve the above.

Step 5 - Stop supporting daemons When you are finished with network testing using QEMU, you should stop any daemons or helpers started in the initial steps, to avoid possible networking or routing problems such as address conflicts in local network interfaces. For example, stop them if you switch from testing networking with QEMU to using real hardware, or to return your host laptop to normal Wi-Fi use.

To stop the daemons, press Ctrl+C in the corresponding terminal windows (you need to stop both `loop-slip-tap.sh` and `loop-socat.sh`).

Exit QEMU by pressing `CTRL+A x`.

Setting up Zephyr and NAT/masquerading on host to access Internet To access the internet from a Zephyr application, some additional setup on the host may be required. This setup is common for both application running in QEMU and on real hardware, assuming that a development board is connected to the development host. If a board is connected to a dedicated router, it should not be needed.

To access the internet from a Zephyr application using IPv4, a gateway should be set via DHCP or configured manually. For applications using the "Settings" facility (with the config option `CONFIG_NET_CONFIG_SETTINGS` enabled), set the `CONFIG_NET_CONFIG_MY_IPV4_GW` option to the IP address of the gateway. For apps not using the "Settings" facility, set up the gateway by calling the `net_if_ipv4_set_gw()` at runtime. For example: `CONFIG_NET_CONFIG_MY_IPV4_GW="192.0.2.2"`

To access the internet from a custom application running in QEMU, NAT (masquerading) should be set up for QEMU's source address. Assuming `192.0.2.1` is used and the Zephyr network interface is `zeth`, the following command should be run as root:

```
iptables -t nat -A POSTROUTING -j MASQUERADE -s 192.0.2.1/24
iptables -I FORWARD 1 -i zeth -j ACCEPT
iptables -I FORWARD 1 -o zeth -m state --state RELATED,ESTABLISHED -j ACCEPT
```

Additionally, IPv4 forwarding should be enabled on the host, and you may need to check that other firewall (iptables) rules don't interfere with masquerading. To enable IPv4 forwarding the following command should be run as root:

```
sysctl -w net.ipv4.ip_forward=1
```

Some applications may also require a DNS server. A number of Zephyr-provided samples assume by default that the DNS server is available on the host (IP `192.0.2.2`), which, in modern Linux distributions, usually runs at least a DNS proxy. When running with QEMU, it may be required to restart the host's DNS, so it can serve requests on the newly created TAP interface. For example, on Debian-based systems:

```
service dnsmasq restart
```

An alternative to relying on the host's DNS server is to use one in the network. For example, `8.8.8.8` is a publicly available DNS server. You can configure it using `CONFIG_DNS_SERVER1` option.

Network connection between two QEMU VMs Unlike the VM-to-Host setup described above, VM-to-VM setup is automatic. For sample applications that support this mode (such as the echo_server and echo_client samples), you will need two terminal windows, set up for Zephyr development.

Terminal #1:

```
west build -b qemu_x86 -o server samples/net/sockets/echo_server
```

This will start QEMU, waiting for a connection from a client QEMU.

Terminal #2:

```
west build -b qemu_x86 -o client samples/net/sockets/echo_client
```

This will start a second QEMU instance, where you should see logging of data sent and received in both.

Running multiple QEMU VMs of the same sample If you find yourself wanting to run multiple instances of the same Zephyr sample application, which do not need to talk to each other, use the `QEMU_INSTANCE` argument.

Start `socat` and `tunslip6` manually (instead of using the `loop-xxx.sh` scripts) for as many instances as you want. Use the following as a guide, replacing MAIN or OTHER.

Terminal #1:

```
socat PTY,link=/tmp/slip.devMAIN UNIX-LISTEN:/tmp/slip.sockMAIN &
sudo $ZEPHYR_BASE/../tools/net-tools/tunslip6 -t tapMAIN -T -s /tmp/slip.devMAIN␣
↪2001:db8::1/64 &
# Now run Zephyr
make -Cbuild run QEMU_INSTANCE=MAIN
```

Terminal #2:

```
socat PTY,link=/tmp/slip.devOTHER UNIX-LISTEN:/tmp/slip.sockOTHER &
sudo $ZEPHYR_BASE/../tools/net-tools/tunslip6 -t tapOTHER -T -s /tmp/slip.devOTHER␣
↪2001:db8::1/64 &
make -Cbuild run QEMU_INSTANCE=OTHER
```

USB Device Networking

- *Basic Setup*
  - *Choosing IP addresses*
  - *Setting IPv4 address and routing*
  - *Setting IPv6 address and routing*
- *Testing connection*

This page describes how to set up networking between a Linux host and a Zephyr application running on USB supported devices.

The board is connected to Linux host using USB cable and provides an Ethernet interface to the host. The sockets-echo-server application from the Zephyr source distribution is run on supported board. The board is connected to a Linux host using a USB cable providing an Ethernet interface to the host.

Basic Setup To communicate with the Zephyr application over a newly created Ethernet interface, we need to assign IP addresses and set up a routing table for the Linux host. After plugging a USB cable from the board to the Linux host, the `cdc_ether` driver registers a new Ethernet device with a provided MAC address.

You can check that network device is created and MAC address assigned by running dmesg from the Linux host.

```
cdc_ether 1-2.7:1.0 eth0: register 'cdc_ether' at usb-0000:00:01.2-2.7, CDC␣
↪Ethernet Device, 00:00:5e:00:53:01
```

We need to set it up and assign IP addresses as explained in the following section.

Choosing IP addresses To establish network connection to the board we need to choose IP address for the interface on the Linux host.

It make sense to choose addresses in the same subnet we have in Zephyr application. IP addresses usually set in the project configuration files and may be checked also from the shell with following commands. Connect a serial console program (such as puTTY) to the board, and enter this command to the Zephyr shell:

```
shell> net iface

Interface 0xa800e580 (Ethernet)
===============================
Link addr : 00:00:5E:00:53:00
MTU       : 1500
IPv6 unicast addresses (max 2):
        fe80::200:5eff:fe00:5300 autoconf preferred infinite
        2001:db8::1 manual preferred infinite
...
IPv4 unicast addresses (max 1):
        192.0.2.1 manual preferred infinite
```

This command shows that one IPv4 address and two IPv6 addresses have been assigned to the board. We can use either IPv4 or IPv6 for network connection depending on the board network configuration.

Next step is to assign IP addresses to the new Linux host interface, in the following steps `enx00005e005301` is the name of the interface on my Linux system.

Setting IPv4 address and routing

```
# ip address add dev enx00005e005301 192.0.2.2
# ip link set enx00005e005301 up
# ip route add 192.0.2.0/24 dev enx00005e005301
```

Setting IPv6 address and routing

```
# ip address add dev enx00005e005301 2001:db8::2
# ip link set enx00005e005301 up
# ip -6 route add 2001:db8::/64 dev enx00005e005301
```

Testing connection From the host we can test the connection by pinging Zephyr IP address of the board with:

```
$ ping 192.0.2.1
PING 192.0.2.1 (192.0.2.1) 56(84) bytes of data.
64 bytes from 192.0.2.1: icmp_seq=1 ttl=64 time=2.30 ms
64 bytes from 192.0.2.1: icmp_seq=2 ttl=64 time=1.43 ms
64 bytes from 192.0.2.1: icmp_seq=3 ttl=64 time=2.45 ms
...
```

Networking with QEMU User

- *Introduction*
- *Using SLIRP with Zephyr*
- *Limitations*

This page is intended to serve as a starting point for anyone interested in using QEMU SLIRP with Zephyr.

Introduction SLIRP is a network backend which provides the complete TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network. As there are no dependencies on the host, SLIRP is simple to setup.

By default, QEMU uses the `10.0.2.X/24` network and runs a gateway at `10.0.2.2`. All traffic intended for the host network has to travel through this gateway, which will filter out packets based on the QEMU command line parameters. This gateway also functions as a DHCP server for all GOS, allowing them to be automatically assigned with an IP address starting from `10.0.2.15`.

More details about User Networking can be obtained from here: https://wiki.qemu.org/Documentation/Networking#User_Networking_.28SLIRP.29

Using SLIRP with Zephyr In order to use SLIRP with Zephyr, the user has to set the Kconfig option to enable User Networking.

```
CONFIG_NET_QEMU_USER=y
```

Once this configuration option is enabled, all QEMU launches will use SLIRP. In the default configuration, Zephyr only enables User Networking, and does not pass any arguments to it. This means that the Guest will only be able to communicate to the QEMU gateway, and any data intended for the host machine will be dropped by QEMU.

In general, QEMU User Networking can take in a lot of arguments including,

- Information about host/guest port forwarding. This must be provided to create a communication channel between the guest and host.
- Information about network to use. This may be valuable if the user does not want to use the default `10.0.2.X` network.
- Tell QEMU to start DHCP server at user-defined IP address.
- ID and other information.

As this information varies with every use case, it is difficult to come up with good defaults that work for all. Therefore, Zephyr Implementation offloads this to the user, and expects that they will provide arguments based on requirements. For this, there is a Kconfig string which can be populated by the user.

```
CONFIG_NET_QEMU_USER_EXTRA_ARGS="net=192.168.0.0/24,hostfwd=tcp::8080-:8080"
```

This option is appended as-is to the QEMU command line. Therefore, any problems with this command line will be reported by QEMU only. Here's what this particular example will do,

- Make QEMU use the `192.168.0.0/24` network instead of the default.
- Enable forwarding of any TCP data received from port 8080 of host to port 8080 of guest, and vice versa.

Limitations If the user does not have any specific networking requirements other than the ability to access a web page from the guest, user networking (slirp) is a good choice. However, it has several limitations

- There is a lot of overhead so the performance is poor.
- The guest is not directly accessible from the host or the external network.
- In general, ICMP traffic does not work (so you cannot use ping within a guest).
- As port mappings need to be defined before launching qemu, clients which use dynamically generated ports cannot communicate with external network.
- There is a bug in the SLIRP implementation which filters out all IPv6 packets from the guest. See https://bugs.launchpad.net/qemu/+bug/1724590 for details. Therefore, IPv6 will not work with User Networking.

Networking with multiple Zephyr instances

- *Prerequisites*
- *Basic Setup*
  - *Step 1 - Create configuration files*
  - *Step 2 - Create Ethernet interfaces*
  - *Step 3 - Setup network bridging*
  - *Step 4 - Start Zephyr instances*

This page describes how to set up a virtual network between multiple Zephyr instances. The Zephyr instances could be running inside QEMU or could be native_sim board processes. The Linux host can be used to route network traffic between these systems.

Prerequisites On the Linux Host, find the Zephyr net-tools project, which can either be found in a Zephyr standard installation under the `tools/net-tools` directory or installed stand alone from its own git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

Basic Setup For the steps below, you will need five terminal windows:

- Terminal #1 and #2 are terminal windows with net-tools being the current directory (`cd net-tools`)
- Terminal #3, where you setup bridging in Linux host
- Terminal #4 and #5 are your usual Zephyr development terminal, with the Zephyr environment initialized.

As there are multiple ways to setup the Zephyr network, the example below uses `qemu_x86` board with `e1000` Ethernet controller and native_sim board to simplify the setup instructions. You can use other QEMU boards and drivers if needed, see *Networking with QEMU Ethernet* for details. You can also use two or more native_sim board Zephyr instances and connect them together.

Step 1 - Create configuration files Before starting QEMU with network connectivity, a network interfaces for each Zephyr instance should be created in the host system. The default setup for creating network interface cannot be used here as that is for connecting one Zephyr instance to Linux host.

For Zephyr instance #1, create file called `zephyr1.conf` to `net-tools` project, or to some other suitable directory.

```
# Configuration file for setting IP addresses for a network interface.
INTERFACE="$1"
HWADDR="00:00:5e:00:53:11"
IPV6_ADDR_1="2001:db8:100::2"
IPV6_ROUTE_1="2001:db8:100::/64"
IPV4_ADDR_1="198.51.100.2/24"
IPV4_ROUTE_1="198.51.100.0/24"
ip link set dev $INTERFACE up
ip link set dev $INTERFACE address $HWADDR
ip -6 address add $IPV6_ADDR_1 dev $INTERFACE nodad
ip -6 route add $IPV6_ROUTE_1 dev $INTERFACE
```

(continues on next page)

(continued from previous page)

```
ip address add $IPV4_ADDR_1 dev $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
```

For Zephyr instance #2, create file called `zephyr2.conf` to `net-tools` project, or to some other suitable directory.

```
# Configuration file for setting IP addresses for a network interface.
INTERFACE="$1"
HWADDR="00:00:5e:00:53:22"
IPV6_ADDR_1="2001:db8:200::2"
IPV6_ROUTE_1="2001:db8:200::/64"
IPV4_ADDR_1="203.0.113.2/24"
IPV4_ROUTE_1="203.0.113.0/24"
ip link set dev $INTERFACE up
ip link set dev $INTERFACE address $HWADDR
ip -6 address add $IPV6_ADDR_1 dev $INTERFACE nodad
ip -6 route add $IPV6_ROUTE_1 dev $INTERFACE
ip address add $IPV4_ADDR_1 dev $INTERFACE
ip route add $IPV4_ROUTE_1 dev $INTERFACE > /dev/null 2>&1
```

Step 2 - Create Ethernet interfaces The following `net-setup.sh` commands should be typed in net-tools directory (`cd net-tools`).

In terminal #1, type:

```
./net-setup.sh -c zephyr1.conf -i zeth.1
```

In terminal #2, type:

```
./net-setup.sh -c zephyr2.conf -i zeth.2
```

Step 3 - Setup network bridging In terminal #3, type:

```
sudo brctl addbr zeth-br
sudo brctl addif zeth-br zeth.1
sudo brctl addif zeth-br zeth.2
sudo ifconfig zeth-br up
```

Step 4 - Start Zephyr instances In this example we start sockets-echo-server and sockets-echo-client sample applications. You can use other applications too as needed.

In terminal #4, if you are using QEMU, type this:

```
west build -d build/server -b qemu_x86 -t run \
   samples/net/sockets/echo_server -- \
   -DEXTRA_CONF_FILE=overlay-e1000.conf \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"203.0.113.1\" \
   -DCONFIG_ETH_QEMU_IFACE_NAME=\"zeth.1\" \
   -DCONFIG_NET_QEMU_DEVICE_EXTRA_ARGS=\"mac=00:00:5e:00:53:01\"
```

or if you want to use native_sim board, type this:

```
west build -d build/server -b native_sim \
   samples/net/sockets/echo_server -- \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"203.0.113.1\"
build/server/zephyr/zephyr.exe --eth-if=zeth.1 --mac-addr=00:00:5e:00:53:01
```

In terminal #5, if you are using QEMU, type this:

```
west build -d build/client -b qemu_x86 -t run \
   samples/net/sockets/echo_client -- \
   -DEXTRA_CONF_FILE=overlay-e1000.conf \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"198.51.100.1\" \
   -DCONFIG_ETH_QEMU_IFACE_NAME=\"zeth.2\" \
   -DCONFIG_NET_QEMU_DEVICE_EXTRA_ARGS=\"mac=00:00:5e:00:53:02\"
```

or if you want to use native_sim board, type this:

```
west build -d build/client -b native_sim \
   samples/net/sockets/echo_client -- \
   -DCONFIG_NET_CONFIG_MY_IPV4_ADDR=\"203.0.113.1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV4_ADDR=\"198.51.100.1\" \
   -DCONFIG_NET_CONFIG_MY_IPV6_ADDR=\"2001:db8:200::1\" \
   -DCONFIG_NET_CONFIG_PEER_IPV6_ADDR=\"2001:db8:100::1\" \
   -DCONFIG_NET_CONFIG_MY_IPV4_GW=\"198.51.100.1\"
build/client/zephyr/zephyr.exe --eth-if=zeth.2 --mac-addr=00:00:5e:00:53:02
```

Also if you have firewall enabled in your host, you need to allow traffic between `zeth.1`, `zeth.2` and `zeth-br` interfaces.

Ethernet bridge with native_sim board This document describes how to set up a bridged Ethernet network between a (Linux) host and a Zephyr application running in a native_sim board.

This setup is useful when testing the Ethernet bridging feature that can be enabled with `CONFIG_NET_ETHERNET_BRIDGE` Kconfig option. In this setup, the net-tools configuration creates two host network interfaces `zeth0` and `zeth1` and connects them to Zephyr's native_sim application.

First create the host interfaces. In this example two interfaces are created.

```
cd $ZEPHYR_BASE/../tools/net-tools
./net-setup.sh -c zeth-multiface.conf -i zeth0 -t 2
```

The `-c` tells which configuration file to use, where `zeth-multiface.conf` is tailored for generating multiple network interfaces in the host. The `-i` option tells what is the first host interface name. The `-t` tells how many network interfaces to create.

Example output of the host interfaces:

```
zeth0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
       inet 192.0.2.2  netmask 255.255.255.255  broadcast 0.0.0.0
       inet6 2001:db8::2  prefixlen 128  scopeid 0x0<global>
       inet6 fe80::200:5eff:fe00:5300  prefixlen 64  scopeid 0x20<link>
       ether 00:00:5e:00:53:00  txqueuelen 1000  (Ethernet)
```

(continues on next page)

(continued from previous page)

```
        RX packets 33  bytes 2408 (2.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 49  bytes 4092 (4.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

zeth1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 198.51.100.1  netmask 255.255.255.255  broadcast 0.0.0.0
        inet6 fe80::200:5eff:fe00:5301  prefixlen 64  scopeid 0x20<link>
        inet6 2001:db8:2::1  prefixlen 128  scopeid 0x0<global>
        ether 00:00:5e:00:53:01  txqueuelen 1000  (Ethernet)
        RX packets 21  bytes 1340 (1.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 45  bytes 3916 (3.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Then create a sample and enable Ethernet bridging support. In this example we create sockets-echo-server sample application.

Bridging needs a second TAP interface. Create a devicetree overlay file `second-iface.overlay` that adds a second `zephyr,native-tap` interface:

```
/ {
    zeth1: zeth1 {
        compatible = "zephyr,native-tap";
        status = "okay";
        zephyr,random-mac-address;
        host-interface = "zeth1";
    };
};
```

Then build and run the application, pointing at the overlay:

```
west build -p -b native_sim -d ../build/echo-server \
   samples/net/sockets/echo_server -- \
   -DCONFIG_UART_NATIVE_PTY_AUTOATTACH_DEFAULT_CMD="\"gnome-terminal -- screen
↪%s\"" \
   -DCONFIG_NET_ETHERNET_BRIDGE=y \
   -DCONFIG_NET_ETHERNET_BRIDGE_SHELL=y \
   -DEXTRA_DTC_OVERLAY_FILE=second-iface.overlay \
   -DCONFIG_NET_IF_MAX_IPV6_COUNT=2 \
   -DCONFIG_NET_IF_MAX_IPV4_COUNT=2
../build/echo-server/zephyr/zephyr.exe -attach_uart
```

This will create and run sockets-echo-server with bridging enabled but not yet configured. To configure the bridging, you either need to use the bridge shell or call the bridging API directly from the application. We setup the bridging using the bridge shell like this:

```
net bridge addif 1 3 2
net iface up 1
```

In the above example, the bridge interface index is 1, and interfaces 2 and 3 are Ethernet interfaces which are linked to interfaces `zeth0` and `zeth1` in the host side.

The network interfaces look like this in Zephyr's side:

```
net iface
Hostname: zephyr
```

(continues on next page)

(continued from previous page)

```
Interface bridge0 (0x8090ebc) (Virtual) [1]
===================================
Virtual name : <enabled>
No attached network interface.
Link addr : 3B:DB:31:0F:CC:B6
MTU       : 1500
Flags     : NO_AUTO_START
Device    : BRIDGE_0 (0x8088354)
Promiscuous mode : disabled
IPv6 not enabled for this interface.
IPv4 not enabled for this interface.

Interface eth0 (0x8090fcc) (Ethernet) [2]
===================================
Link addr : 02:00:5E:00:53:D2
MTU       : 1500
Flags     : AUTO_START,IPv4,IPv6
Device    : zeth0 (0x808837c)
Promiscuous mode : disabled
Ethernet capabilities supported:
        TXTIME
        Promiscuous mode
Ethernet PHY device: <none> (0)
IPv6 unicast addresses (max 3):
        fe80::5eff:fe00:53d2 autoconf preferred infinite
        2001:db8::1 manual preferred infinite
IPv6 multicast addresses (max 4):
        ff02::1
        ff02::1:ff00:53d2
        ff02::1:ff00:1
IPv6 prefixes (max 2):
        <none>
IPv6 hop limit           : 64
IPv6 base reachable time : 30000
IPv6 reachable time      : 18476
IPv6 retransmit timer    : 0
IPv4 unicast addresses (max 1):
        192.0.2.1/255.255.255.0 manual preferred infinite
IPv4 multicast addresses (max 2):
        224.0.0.1
IPv4 gateway : 0.0.0.0

Interface eth1 (0x80910dc) (Ethernet) [3]
===================================
Link addr : 02:00:5E:00:53:87
MTU       : 1500
Flags     : AUTO_START,IPv4,IPv6
Device    : zeth1 (0x8088368)
Promiscuous mode : disabled
Ethernet capabilities supported:
        TXTIME
        Promiscuous mode
Ethernet PHY device: <none> (0)
IPv6 unicast addresses (max 3):
        fe80::5eff:fe00:5387 autoconf preferred infinite
IPv6 multicast addresses (max 4):
```

(continues on next page)

(continued from previous page)

```
        ff02::1
        ff02::1:ff00:5387
IPv6 prefixes (max 2):
        <none>
IPv6 hop limit           : 64
IPv6 base reachable time : 30000
IPv6 reachable time      : 25158
IPv6 retransmit timer    : 0
IPv4 unicast addresses (max 1):
        <none>
IPv4 multicast addresses (max 2):
        224.0.0.1
IPv4 gateway : 0.0.0.0
```

The `net bridge` command will show the current status of the bridging:

```
net bridge
Bridge Status   Config   Interfaces
1      up       ok       2 3
```

The `addif` command adds Ethernet interfaces 2 and 3 to the bridge interface 1. After the `addif` command, the bridging is still disabled because the bridge interface is not up by default. The `net iface up` command will turn on bridging.

If you have wireshark running in host side and monitoring `zeth0` and `zeth1`, you should see the same network traffic in both host interfaces.

Note that interface index numbers are not fixed, the bridge and Ethernet interface index values might be different in your setup.

The bridging can be disabled by taking the bridge interface down, and the Ethernet interfaces can be removed from the bridge using `delif` command.

```
net iface down 1
net bridge delif 1 2 3
```

Networking with QEMU and IEEE 802.15.4

- *Basic Setup*
  - *Step 1 - Compile and start echo-server*
  - *Step 2 - Compile and start echo-client*

This page describes how to set up a virtual network between two QEMUs that are connected together via UART and are running IEEE 802.15.4 link layer between them. Note that this only works in Linux host.

Basic Setup For the steps below, you will need two terminal windows:

- Terminal #1 is terminal window with `echo-server` Zephyr sample application.
- Terminal #2 is terminal window with `echo-client` Zephyr sample application.

If you want to capture the transferred network data, you must compile the `monitor_15_4` program in the `tools/net-tools` directory.

Open a terminal window and type:

```
cd $ZEPHYR_BASE/../tools/net-tools
make monitor_15_4
```

Step 1 - Compile and start echo-server In terminal #1, type:

```
west build -b qemu_x86 -d build/server samples/net/sockets/echo_server -- -
↪DEXTRA_CONF_FILE=overlay-qemu_802154.conf
west build -t server -d build/server
```

If you want to capture the network traffic between the two QEMUs, type:

```
west build -b qemu_x86 -d build/server samples/net/sockets/echo_server -- -G'Unix␣
↪Makefiles' -DEXTRA_CONF_FILE=overlay-qemu_802154.conf -DPCAP=capture.pcap
west build -t server -d build/server
```

Note that the `make` must be used for `server` target if packet capture option is set in command line. The `build/server/capture.pcap` file will contain the transferred data.

Step 2 - Compile and start echo-client In terminal #2, type:

```
west build -b qemu_x86 -d build/client samples/net/sockets/echo_client -- -
↪DEXTRA_CONF_FILE=overlay-qemu_802154.conf
west build -t client -d build/client
```

You should see data passed between the two QEMUs. Exit QEMU by pressing `CTRL+A x`.

Networking with Arm FVP User Mode

- *Introduction*
- *Using Arm FVP User Mode Networking with Zephyr*
- *Limitations*

This page is intended to serve as a starting point for anyone interested in using Arm FVP user mode networking with Zephyr.

Introduction User mode networking emulates a built-in IP router and DHCP server, and routes TCP and UDP traffic between the guest and host. It uses the user mode socket layer of the host to communicate with other hosts. This allows the use of a significant number of IP network services without requiring administrative privileges, or the installation of a separate driver on the host on which the model is running.

By default, Arm FVP uses the `172.20.51.0/24` network and runs a gateway at `172.20.51.254`. This gateway also functions as a DHCP server for the GOS, allowing it to be automatically assigned with an IP address `172.20.51.1`.

More details about Arm FVP user mode networking can be obtained from here: https://developer.arm.com/documentation/100964/latest/Introduction-to-Fast-Models/User-mode-networking

Using Arm FVP User Mode Networking with Zephyr Arm FVP user mode networking can be enabled in any applications and it doesn't need any configurations on the host system. This feature has been enabled in DHCPv4 client sample. See dhcpv4-client sample application.

Limitations

- You can use TCP and UDP over IP, but not ICMP (ping).
- User mode networking does not support forwarding UDP ports on the host to the model.
- You can only use DHCP within the private network.
- You can only make inward connections by mapping TCP ports on the host to the model. This is common to all implementations that provide host connectivity using NAT.
- Operations that require privileged source ports, for example NFS in its default configuration, do not work.
- If setup fails, or the parameter syntax is incorrect, there is no error reporting.

While developing networking software, it is usually necessary to connect and exchange data with the host system like a Linux desktop computer. Depending on what board is used for development, the following options are possible:

- QEMU using SLIP (Serial Line Internet Protocol).
  - Here IP packets are exchanged between Zephyr and the host system via serial port. This is the legacy way of transferring data. It is also quite slow so use it only when necessary. See *Networking with QEMU* for details.
- QEMU using built-in Ethernet driver.
  - Here IP packets are exchanged between Zephyr and the host system via QEMU's built-in Ethernet driver. Not all QEMU boards support built-in Ethernet so in some cases, you might need to use the SLIP method for host connectivity. See *Networking with QEMU Ethernet* for details.
- QEMU using SLIRP (Qemu User Networking).
  - QEMU User Networking is implemented using "slirp", which provides a full TCP/IP stack within QEMU and uses that stack to implement a virtual NAT'd network. As this support is built into QEMU, it can be used with any model and requires no admin privileges on the host machine, unlike TAP. However, it has several limitations including performance which makes it less valuable for practical purposes. See *Networking with QEMU User* for details.
- Arm FVP (User Mode Networking).
  - User mode networking emulates a built-in IP router and DHCP server, and routes TCP and UDP traffic between the guest and host. It uses the user mode socket layer of the host to communicate with other hosts. This allows the use of a significant number of IP network services without requiring administrative privileges, or the installation of a separate driver on the host on which the model is running. See *Networking with Arm FVP User Mode* for details.
- native_sim board.
  - The Zephyr instance can be executed as a user space process in the host system. This is the most convenient way to debug the Zephyr system as one can attach host debugger directly to the running Zephyr instance. This requires that there is an adaptation driver in Zephyr for interfacing with the host system. Two possible network drivers can be used for this purpose, a TAP virtual Ethernet driver and an offloaded sockets driver. See *Networking with native_sim board* for details.
- USB device networking.
  - Here, the Zephyr instance is run on a real board and the connectivity to the host system is done via USB. See *USB Device Networking* for details.
- Connecting multiple Zephyr instances together.

  - If you have multiple Zephyr instances, either QEMU or native_sim ones, and want to create a connection between them, see *Networking with multiple Zephyr instances* for details.
- Simulating IEEE 802.15.4 network between two QEMUs.
  - Here, two Zephyr instances are running and there is IEEE 802.15.4 link layer run over an UART between them. See *Networking with QEMU and IEEE 802.15.4* for details.
- Simulating Ethernet bridge network with native_sim.
  - Here, one Zephyr instance is running with Ethernet bridge enabled via `CONFIG_NET_ETHERNET_BRIDGE` Kconfig option. There exists two host network interfaces `zeth0` and `zeth1` and the network packets are bridged between those two interfaces. See *Ethernet bridge with native_sim board* for details.

Monitor Network Traffic

- *Host Configuration*
- *Zephyr Configuration*
- *Wireshark Configuration*

It is useful to be able to monitor the network traffic especially when debugging a connectivity issues or when developing new protocol support in Zephyr. This page describes how to set up a way to capture network traffic so that user is able to use Wireshark or similar tool in remote host to see the network packets sent or received by a Zephyr device.

See also the net-capture sample application from the Zephyr source distribution for configuration options that need to be enabled.

Host Configuration The instructions here describe how to setup a Linux host to capture Zephyr network RX and TX traffic. Similar instructions should work also in other operating systems.

On the Linux Host, find the Zephyr net-tools project, which can either be found in a Zephyr standard installation under the `tools/net-tools` directory or installed stand alone from its own git repository:

```
git clone https://github.com/zephyrproject-rtos/net-tools
```

The `net-tools` project provides a configure file to setup IP-to-IP tunnel interface so that we can transfer monitoring data from Zephyr to host.

In terminal #1, type:

```
./net-setup.sh -c zeth-tunnel.conf
```

This script will create following IPIP tunnel interfaces:

Interface name	Description
`zeth-ip6ip`	IPv6-over-IPv4 tunnel
`zeth-ipip`	IPv4-over-IPv4 tunnel
`zeth-ipip6`	IPv4-over-IPv6 tunnel
`zeth-ip6ip6`	IPv6-over-IPv6 tunnel

Zephyr will send captured network packets to one of these interfaces. The actual interface will depend on how the capturing is configured. You can then use Wireshark to monitor the proper network interface.

After the tunneling interfaces have been created, you can use for example `net-capture.py` script from `net-tools` project to print or save the captured network packets. The `net-capture.py` provides an UDP listener, it can print the captured data to screen and optionally can also save the data to a pcap file.

```
$ ./net-capture.py -i zeth-ip6ip -w capture.pcap
[20210408Z14:33:08.959589] Ether / IP / ICMP 192.0.2.1 > 192.0.2.2 echo-request 0 /
↪ Raw
[20210408Z14:33:08.976178] Ether / IP / ICMP 192.0.2.2 > 192.0.2.1 echo-reply 0 /␣
↪Raw
[20210408Z14:33:16.176303] Ether / IPv6 / ICMPv6 Echo Request (id: 0x9feb seq: 0x0)
[20210408Z14:33:16.195326] Ether / IPv6 / ICMPv6 Echo Reply (id: 0x9feb seq: 0x0)
[20210408Z14:33:21.194979] Ether / IPv6 / ICMPv6ND_NS / ICMPv6 Neighbor Discovery␣
↪Option - Source Link-Layer Address 02:00:5e:00:53:3b
[20210408Z14:33:21.217528] Ether / IPv6 / ICMPv6ND_NA / ICMPv6 Neighbor Discovery␣
↪Option - Destination Link-Layer Address 00:00:5e:00:53:ff
[20210408Z14:34:10.245408] Ether / IPv6 / UDP 2001:db8::2:47319 > 2001:db8::1:4242␣
↪/ Raw
[20210408Z14:34:10.266542] Ether / IPv6 / UDP 2001:db8::1:4242 > 2001:db8::2:47319␣
↪/ Raw
```

The `net-capture.py` has following command line options:

```
Listen captured network data from Zephyr and save it optionally to pcap file.
./net-capture.py \
     -i | --interface <network interface>
             Listen this interface for the data
     [-p | --port <UDP port>]
             UDP port (default is 4242) where the capture data is received
     [-q | --quiet]
             Do not print packet information
     [-t | --type <L2 type of the data>]
             Scapy L2 type name of the UDP payload, default is Ether
     [-w | --write <pcap file name>]
             Write the received data to file in PCAP format
```

Instead of the `net-capture.py` script, you can for example use `netcat` to provide an UDP listener so that the host will not send port unreachable message to Zephyr:

```
nc -l -u 2001:db8:200::2 4242 > /dev/null
```

The IP address above is the inner tunnel endpoint, and can be changed and it depends on how the Zephyr is configured. Zephyr will send UDP packets containing the captured network packets to the configured IP tunnel, so we need to terminate the network connection like this.

Zephyr Configuration In this example, we use the `native_sim` board. You can also use any other board that supports networking.

In terminal #3, type:

```
west build -b native_sim samples/net/capture -- -
↪DCONFIG_UART_NATIVE_PTY_AUTOATTACH_DEFAULT_CMD=\""gnome-terminal -- screen %s"\"
```

To see the Zephyr console and shell, start Zephyr instance like this:

```
build/zephyr/zephyr.exe -attach_uart
```

Any other application can be used too, just make sure that suitable configuration options are enabled (see `samples/net/capture/prj.conf` file for examples).

The network capture can be configured automatically if needed, but currently the `capture` sample application does not do that. User has to use `net-shell` to setup and enable the monitoring.

The network packet monitoring needs to be setup first. The `net-shell` has `net capture setup` command for doing that. The command syntax is

```
net capture setup <remote-ip-addr> <local-ip-addr> <peer-ip-addr>
     <remote> is the (outer) endpoint IP address
     <local> is the (inner) local IP address
     <peer> is the (inner) peer IP address
     Local and Peer IP addresses can have UDP port number in them (optional)
     like 198.0.51.2:9000 or [2001:db8:100::2]:4242
```

In Zephyr console, type:

```
net capture setup 192.0.2.2 2001:db8:200::1 2001:db8:200::2
```

This command will create the tunneling interface. The `192.0.2.2` is the remote host where the tunnel is terminated. The address is used to select the local network interface where the tunneling interface is attached to. The `2001:db8:200::1` tells the local IP address for the tunnel, the `2001:db8:200::2` is the peer IP address where the captured network packets are sent. The port numbers for UDP packet can be given in the setup command like this for IPv6-over-IPv4 tunnel

```
net capture setup 192.0.2.2 [2001:db8:200::1]:9999 [2001:db8:200::2]:9998
```

and like this for IPv4-over-IPv4 tunnel

```
net capture setup 192.0.2.2 198.51.100.1:9999 198.51.100.2:9998
```

If the port number is omitted, then `4242` UDP port is used as a default.

The current monitoring configuration can be checked like this:

```
uart:~$ net capture
Network packet capture disabled
                Capture  Tunnel
Device          iface    iface   Local                   Peer
NET_CAPTURE0    -        1      [2001:db8:200::1]:4242  [2001:db8:200::2]:4242
```

which will print the current configuration. As we have not yet enabled monitoring, the `Capture iface` is not set.

Then we need to enable the network packet monitoring like this:

```
net capture enable 2
```

The `2` tells the network interface which traffic we want to capture. In this example, the `2` is the `native_sim` board Ethernet interface. Note that we send the network traffic to the same interface that we are monitoring in this example. The monitoring system avoids to capture already captured network traffic as that would lead to recursion. You can use `net iface` command to see what network interfaces are available. Note that you cannot capture traffic from the tunnel interface as that would cause recursion loop. The captured network traffic can be sent to some other network interface if configured so. Just set the `<remote-ip-addr>` option properly in `net capture setup` so that the IP tunnel is attached to desired network interface. The capture status can be checked again like this:

```
uart:~$ net capture
Network packet capture enabled
                Capture  Tunnel
Device          iface    iface   Local                   Peer
NET_CAPTURE0    2        1      [2001:db8:200::1]:4242  [2001:db8:200::2]:4242
```

After enabling the monitoring, the system will send captured (either received or sent) network packets to the tunnel interface for further processing.

The monitoring can be disabled like this:

```
net capture disable
```

which will turn currently running monitoring off. The monitoring setup can be cleared like this:

```
net capture cleanup
```

It is not necessary to use `net-shell` for configuring the monitoring. The *network capture API* functions can be called by the application if needed.

Wireshark Configuration The Wireshark tool can be used to monitor the captured network traffic in a useful way.

You can monitor either the tunnel interfaces or the `zeth` interface. In order to see the actual captured data inside an UDP packet, see Wireshark decapsulate UDP document for instructions.

Network Tracing

User can enable network core stack and socket API calls tracing.

The `CONFIG_TRACING_NET_CORE` option controls the core network stack tracing. This option is enabled by default if tracing and networking are enabled. The system will start to collect the receiving and sending call verdicts i.e., whether the network packet was successfully sent or received. It will also collect packet sending or receiving timings i.e., how long it took to deliver the network packet, and the network interface, priority and traffic class used.

The `CONFIG_TRACING_NET_SOCKETS` option can be used to track BSD socket call usage in the system. It is enabled if tracing and BSD socket API support are enabled. The system will start to collect what BSD socket API calls are made and what parameters the API calls are using and returning.

See the *tracing documentation* for how to use the tracing service.

Protocol conformance testing with TTCN-3

- *How it fits together*
- *The suites*
- *Known gaps*
  - *DNS-SD legacy unicast queries*
  - *MQTT 5.0, packet identifiers and re-sending*
  - *Overlapping DNS queries*
- *Continuous integration*
- *Other TTCN-3 suites*

Zephyr's network protocols are covered from two directions. The tests under tests/net exercise the implementation from the inside, in C, built into the same image. Conformance suites written in TTCN-3 come at it from the outside: they speak the protocol over a real network interface, and check what Zephyr sends against what the standard requires.

The two catch different things. A test written against the implementation tends to encode what the implementation does. A suite written against the standard does not know what the implementation does, which is the point.

TTCN-3 is a language standardised by ETSI for writing tests. The suites here are compiled with Eclipse Titan, an open source TTCN-3 compiler, and are kept in the `net-tools` repository under `ttcn3`, alongside the other host side tools used for network testing.

Running the conformance suites

- *What a run needs*
- *Installing Titan*
  - *Building Titan from source*
- *Running the suites with the script*
- *Setting up the network interfaces*
  - *`zeth`, the shared interface*
  - *`zethL2`, the address-less interface*
- *Running the suites with Twister*
  - *Running as root*
  - *Why a run is serial*
- *Running a suite by hand*
- *Reading the result*
- *When a suite is skipped*
- *Troubleshooting*
  - *The suite does not compile*
  - *The suite sees nothing and times out*
  - *A run hangs or is cut off*
  - *Leftover state*

A run goes like this. Twister builds the system under test and starts it, the harness builds the TTCN-3 suite with Titan, the suite talks to the running application over a tap interface, and Titan's verdict becomes the test result.

What a run needs

- **A Titan installation**, with `TTCN3_DIR` pointing at it. See *Installing Titan*.
- **The third party TTCN-3 modules**, fetched once with `ttcn3/fetch-modules.sh`. They are cloned at pinned commits and are not part of the `net-tools` repository.
- **A checkout of net-tools**, which a west workspace already has: it is in `west.yml` under the `tools` group, which is not filtered out.
- **make**, because that is how Titan builds a suite.
- **The `zeth` interface**, or `zethL2` for a suite that works below the IP layer. See *Setting up the network interfaces*.
- **Root**, for a suite that cannot avoid a privileged port or a packet socket.
- **expect**, for a suite that runs through Titan's main controller.

The suites lists which suites need which of these, and scripts/net/run-conformance-tests.sh `--list` prints the same.

The harness looks for net-tools in `NET_TOOLS_BASE` if that is set, otherwise at `../tools/net-tools/ttcn3` relative to `ZEPHYR_BASE` and then one directory further up. All of this is checked before anything is built, and a missing piece skips the test with a reason rather than failing it.

Installing Titan Most distributions package a Titan, and that is what continuous integration installs:

```
sudo apt install --no-install-recommends eclipse-titan expect
export TTCN3_DIR=/usr
```

The packaged version trails the protocol modules the suites build against, so if a suite fails to compile, build a current Titan from source instead.

Building Titan from source `net-tools/docker/Dockerfile.ttcn3` builds Titan into `/opt/titan` and is the reference for doing it by hand, either as a container or as a recipe to follow. It is a local option, not what continuous integration uses.

Two things a hand built Titan has to get right. Titan is configured through a `Makefile.personal` in its source tree rather than a `configure` script, and `TTCN3_DIR` there is the install prefix. And `make install` has to be serial: parts of the runtime include headers that another part generates, and a parallel make loses that race.

Running the suites with the script scripts/net/run-conformance-tests.sh does what the next two sections describe in one command, and is the easiest way to run the suites: it finds net-tools, fetches the modules if they are missing, creates whichever interfaces the selected suites need, re-runs itself under `sudo` if a selected suite needs root, runs Twister once, and tears the interfaces down again.

```
export TTCN3_DIR=/usr
./scripts/net/run-conformance-tests.sh
```

Naming suites runs only those, which is the quick way to stay unprivileged while working on one:

```
./scripts/net/run-conformance-tests.sh mdns dns
```

`--list` shows the suites and what each one needs, `--keep` leaves the interfaces up for the next run, and `--start` and `--stop` do only that half. `--help` lists the rest, along with the directories it detected.

Because a privileged run creates files as root, the script hands the Twister output directory and the suite build directories back to the invoking user before it exits.

Setting up the network interfaces The suites use two tap interfaces, because a suite working below the IP layer cannot share a link with a host that answers for itself.

`zeth`, the shared interface Used by the suites that work over sockets, where the tester is just another host on the link:

```
cd $ZEPHYR_BASE/../tools/net-tools
sudo ./net-setup.sh --config zeth.conf start
```

The host holds `192.0.2.2/24` and `2001:db8::2`; Zephyr answers on `192.0.2.1` and `2001:db8::1`. Tear it down with `stop` in place of `start`.

zethL2, the address-less interface Used by the suites that work below the IP layer:

```
sudo ./net-setup.sh --config zeth-l2.conf --iface zethL2 start
```

This interface is deliberately given no IP address. Linux answers address resolution and neighbour discovery for any address it holds on any interface unless it is told otherwise, and an answer from the host would be indistinguishable from an answer from Zephyr. The tester speaks raw frames, so it needs no address of its own.

The configuration sets three sysctls to stop the host joining in: `arp_ignore=8` so it answers address resolution for no local address at all, `arp_announce=2` so it never answers with an address this interface does not hold, and `disable_ipv6=1` so there are no neighbour advertisements or router solicitations.

The name `zethL2` is not freely choosable; see *The test network*.

Running the suites with Twister Fetch the third party modules once. The script is safe to re-run:

```
cd $ZEPHYR_BASE/../tools/net-tools
./ttcn3/fetch-modules.sh
```

Then run the tests:

```
export TTCN3_DIR=/usr
cd $ZEPHYR_BASE
./scripts/twister -p native_sim --enable-slow -T tests/net/conformance
```

`--enable-slow` is required: the suites mark themselves slow, because a full run takes tens of minutes. `native_sim` is the only platform they allow.

A single suite is selected by its test identifier, which is `net.conformance.<suite>`:

```
./scripts/twister -p native_sim --enable-slow -T tests/net/conformance \
    -s net.conformance.mdns
```

They also carry the `net` and `conformance` tags, so `--tag conformance` picks up all of them.

Running as root Some suites have to be run as root. DHCP is defined on ports 67 and 68 and there is no way to move it elsewhere, so the tester cannot avoid binding a privileged port; and reading frames off a link needs a packet socket. Those tests skip themselves when they are not run with enough privilege.

Use `sudo -E` so that `TTCN3_DIR` and the rest of the environment survive. A run is either wholly privileged or wholly not — see *The harness* for why the two cannot be mixed.

Why a run is serial Every system under test answers to the same address on the same interface, so only one conformance test can be running at a time. They take an exclusive lock on the interface and wait for each other, which means a run of the whole directory is serial however many jobs Twister is given.

Running a suite by hand Twister is convenient but slow to go round. While writing or debugging a suite, run the two halves yourself.

Start the system under test and leave it running:

```
cd $ZEPHYR_BASE
west build -p -b native_sim -d ../build/mdns tests/net/conformance/mdns
../build/mdns/zephyr/zephyr.exe
```

Build and run the suite against it:

```
cd $ZEPHYR_BASE/../tools/net-tools/ttcn3
./build.sh mdns
cd suites/mdns/build
./mdns ../mdns.cfg
```

For a suite whose test cases create parallel test components, start it through the main controller instead:

```
ttcn3_start ./coap ../coap.cfg
```

A single test case is run by naming it:

```
./mdns ../mdns.cfg MDNS_Suite.tc_a_query
```

Addresses, the interface and the timeouts all come from the `[MODULE_PARAMETERS]` section of the suite's configuration file, so a run can be moved to a different link by editing one file rather than the suite.

One thing the harness does that you have to do yourself: put Titan's library directory on `LD_LIBRARY_PATH`.

Reading the result A Titan run ends with a count of each verdict and a verdict for the run:

```
Verdict statistics: 0 none (0.00 %), 7 pass (100.00 %), 0 inconc (0.00 %), 0 fail␣
↪(0.00 %), 0 error (0.00 %).
Test execution summary: 7 test cases were executed. Overall verdict: pass
```

`inconc` means a test case could not reach a conclusion, usually because something it depended on did not happen. It is not a pass. `error` means the suite itself failed, rather than the system under test.

The evidence is in two places. Titan writes a log per suite into the build directory, named from the `LogFile` setting in the configuration file. Twister writes `twister_harness.log` under its output directory, which carries the whole suite output at `INFO`.

When a suite is skipped Everything a suite needs is checked before anything is built, and a missing piece skips the test rather than failing it. The reasons, in the order they are checked:

Reason	What to do
`TTCN3_DIR is unset`	Install Titan and export `TTCN3_DIR`; see *Installing Titan*
`make is not installed`	Install make; Titan builds a suite with a generated makefile
`no TTCN-3 suites under ...`	net-tools was not found; set `NET_TOOLS_BASE`
`... has no <suite> suite`	The net-tools checkout predates the suite; update it
`third party modules are missing`	Run `ttcn3/fetch-modules.sh`
`the <iface> interface does not exist`	Create it with `net-setup.sh`; see *Setting up the network interfaces*
`ttcn3_start is not in TTCN3_DIR/bin`	Install `expect` and a Titan that ships the main controller
`has to be run as root`	Re-run under `sudo -E`, or use the script

Set `NET_CONFORMANCE_REQUIRED` in the environment to turn each of these into a failure. That is for a run whose purpose is the suites, such as the nightly job, where a test that did not run must not pass for one that did.

Troubleshooting

The suite does not compile Almost always a packaged Titan that trails the protocol modules the suite builds against. Build Titan from source.

The suite sees nothing and times out Check that the application and the suite are on the same interface: a suite working below IP wants `zethL2`, and the application has to be built with a `host-interface` property naming the same interface, which the test sets in `boards/native_sim.overlay`. If the host is answering on the link instead of Zephyr, the `zethL2` sysctls did not take; confirm with `sysctl net.ipv4.conf.zethL2.arp_ignore`.

A run hangs or is cut off Four timeouts nest around a run, and which one fires says where the problem is: 30 seconds for the application's ready line, 1800 seconds for the suite build, 600 seconds for the suite run, and 900 seconds for the Twister test as a whole. A suite that overruns is killed along with its whole process group, so no main controller is left behind.

Leftover state Tear the interfaces down with `net-setup.sh` and `stop`. The lock the tests take is a file named `zephyr-net-conformance-<euid>.lock` in the temporary directory; it is released when the process exits, so a stale one is harmless.

How the conformance tests are put together

- *The pieces*
- *The Zephyr side*
  - *The system under test*
  - *Twister integration*
- *The harness*
  - *Holding the interface*
  - *Suite traits*
  - *Building and running the suite*
  - *Turning a verdict into a result*
- *The host side*
  - *Layout*
  - *How a suite is built*
  - *The shared TTCN-3 layer*
  - *The ethernet test port*
  - *Third party modules*
- *The test network*
- *Adding a suite*
  - *Choosing the shape*
  - *The host side*
  - *The Zephyr side*
  - *Recording what does not match*
  - *Trying it out*

There are four moving parts, spread over two repositories, and they meet only on the wire: a Zephyr application, a Twister and pytest harness, a shell wrapper around Eclipse Titan, and the TTCN-3 suite itself.

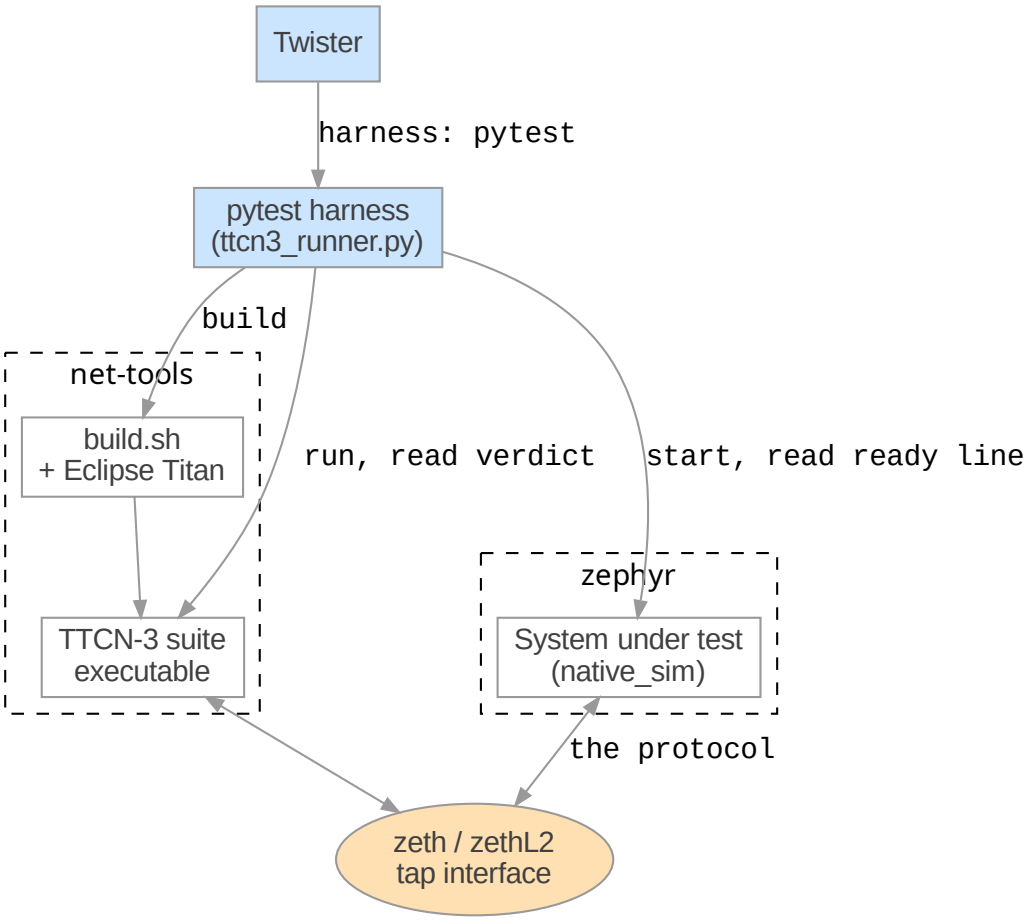


Fig. 4: What builds what, and where the two halves meet

The pieces

Twister
: Builds the system under test, starts it, and runs the pytest harness against it. It contributes the test identifier, the platform restriction and the 900 second budget for the whole thing.

pytest harness
: tests/net/conformance/ttcn3_runner.py, shared by every test. It decides whether the suite can run at all, takes the interface lock, builds the suite, runs it and reads the verdict.

System under test
: An ordinary Zephyr application with the protocol enabled. Nothing about the test is compiled into it.

build.sh and Eclipse Titan
: The host side build. Titan compiles TTCN-3 to `C++` and generates a makefile; `build.sh` arranges the sources so that it can.

TTCN-3 suite executable
The test itself, driving the protocol from the outside.

Tap interface
The only thing the two halves share. There is no control channel, no shared memory and no test hook in the Zephyr image.

The Zephyr side

The system under test Each application is deliberately ordinary: enable the protocol, do whatever keeps traffic flowing that the suite needs to observe, and print a distinctive ready line. That line is the only contract between the application and the harness — it is how the harness knows the stack is up before the suite starts sending.

A suite directory holds:

```
tests/net/conformance/<suite>/
    CMakeLists.txt
    prj.conf
    README.rst
    tests.yaml
    src/main.c
    pytest/pytest.ini
    pytest/conftest.py
    pytest/test_<suite>_conformance.py
```

Twister integration Every `tests.yaml` shares the same block:

```yaml
common:
  harness: pytest
  slow: true
  timeout: 900
  platform_allow:
    - native_sim
  integration_platforms:
    - native_sim
```

`harness: pytest` hands the run to the harness rather than reading console output for a ztest summary. `slow: true` keeps the suites out of an ordinary Twister run, since a full pass takes tens of minutes. The 900 second timeout has to cover building the suite as well as running it. `native_sim` is the only platform because the tap driver is what puts the system under test on a real link.

The test itself is three lines: request the lock, wait for the ready line, run the suite.

```python
def test_mdns_conformance(network_lock, dut, suite_binary):
    dut.readlines_until(regex='mDNS responder ready', timeout=30.0)
    run_suite(suite_binary, SUITE)
```

The `network_lock` argument comes first on purpose; see below. `conftest.py` is the same in every test directory and does one thing: put the shared runner on `sys.path` and re-export the lock fixture.

Fig. 5: One conformance test, from lock to verdict

The harness Everything below lives in tests/net/conformance/ttcn3_runner.py.

Holding the interface Twister runs each test in its own pytest process, so excluding one test from another has to work between processes. The harness takes an exclusive `flock` on a lock file in a session scoped fixture.

The ordering matters as much as the lock. The fixture is requested *before* the `dut` fixture, so the system under test is not even started while another conformance test holds the interface — two applications answering to `192.0.2.1` at once would confuse both suites.

The lock file name carries the effective user id. A privileged suite runs as root, and root cannot open a lock file another user left behind in a sticky temporary directory. Since a run is wholly privileged or wholly not, a lock per user still excludes everything that could collide — which is also why a privileged and an unprivileged run must not be started at the same time.

Suite traits Three facts about a suite are read from `suites/<name>/build.conf`:

`MODE=parallel`
: The test cases create parallel test components, so the suite is run through Titan's main controller rather than as a single executable, and `expect` has to be installed.

`PRIVILEGED=yes`
: The suite binds a privileged port or opens a packet socket, so the run has to be root.

`L2=yes`
: The suite works below the IP layer, so it wants `zethL2` rather than `zeth`.

The same file is sourced as a shell fragment by `build.sh`, which is why it is written as shell assignments; the harness only matches substrings in it.

Building and running the suite Building is `build.sh <suite>` with a 1800 second budget, producing `suites/<suite>/build/<suite>`.

Running has to cope with Titan being installed two different ways. A distribution package puts its libraries in *TTCN3_DIR*`/lib/titan`, a source build in *TTCN3_DIR*`/lib`; the harness picks whichever exists and prepends it to `LD_LIBRARY_PATH`, and prepends *TTCN3_DIR*`/bin` to `PATH`. A parallel suite is started with `ttcn3_start`, any other suite directly.

The suite is started in a new session, so that a suite which overruns can be killed along with everything it started — a main controller left running would hold the interface for the next test.

Turning a verdict into a result Two regular expressions match the lines Titan prints at the end of a run, and three things are asserted: that a verdict was printed at all, that at least one test case ran, and that the overall verdict is `pass`. A suite that produced no output, or that ran nothing because every case was filtered out, fails rather than quietly passing. See *Reading the result* for what the verdicts mean.

The host side Everything below is in the `net-tools` repository, under `ttcn3`.

Layout

```
ttcn3/
    common/            shared TTCN-3 modules and the ethernet test port
    modules/           third party modules, cloned, not in git
    modules.txt        which modules, at which commit
    fetch-modules.sh   clone or check out the pinned commits
    build.sh           build one suite
    suites/<name>/     the suite, its sources.txt and its .cfg
```

How a suite is built `build.sh` wipes the suite's build directory and rebuilds it flat, symlinking every source it needs side by side in one directory.

The flatness is not tidiness, it is a workaround. The makefile Titan generates builds a dependency rule with `sed`, using the target stem as the pattern, and that breaks as soon as a source is named through a path containing a slash. Every source therefore has to be reachable by its bare name.

The link order is the suite's own sources, then `common`, then the module sources named in `common/sources.txt`, then those in the suite's own `sources.txt`. Then `ttcn3_makefilegen` generates the makefile — with `-s` for a single mode suite, without it for a parallel one — and `make` builds it.

The shared TTCN-3 layer `Zephyr_SUT` holds the module parameters every suite shares. A suite never writes an address or a timeout into itself; it takes them from here, so that a run can be moved to a different link by editing one configuration file.

Parameter	Default	Meaning
`tsp_sut_ipv4`	`192.0.2.1`	Where Zephyr answers
`tsp_sut_ipv6`	`2001:db8::1`	Where Zephyr answers, IPv6
`tsp_tester_ipv4`	`192.0.2.2`	Where the suite answers
`tsp_tester_ipv6`	`2001:db8::2`	Where the suite answers, IPv6
`tsp_tester_interface`	`zeth`	Needed for link local multicast
`tsp_sut_hostname`	`zephyr`	The name the responder owns
`tsp_response_timeout`	`5.0`	How long an answer may take
`tsp_silence_timeout`	`2.0`	How long silence is watched for

`Zephyr_Transport` provides the `Zephyr_Tester` component, with an `IPL4asp` port and helpers to listen, open, send, receive with a guard timer, and assert that nothing arrives. A socket based suite extends that component rather than opening sockets itself.

`ARP_Types` is hand written, and lives in `common` rather than in the `arp` suite, because any suite working below IP has to answer address resolution itself on a link where nothing else will. The Titan project publishes no ARP module.

The ethernet test port Titan publishes a raw link layer test port, `LANL2asp`, which this does not use. It captures with libpcap and opens the handle with a zero read timeout, which on Linux means "wait until a capture block fills". On a link as quiet as a test link the frames never reach the test, and there is no parameter to change it.

`common/Ethernet_PT.cc` reads a packet socket instead, which hands up every frame as it arrives. It takes three test port parameters: `interface`, which is required; `source_address`, defaulting to the interface's own; and `ethertype`, which when set passes only that type up.

Third party modules The suites build against the protocol modules and test ports the Titan project publishes, rather than defining their own message formats. Eleven repositories are used, each pinned to a commit in `modules.txt` and cloned on demand by `fetch-modules.sh`. They are not vendored, and the clones are ignored by git, so a suite that passes today still builds tomorrow. Moving a pin is a one line edit and a re-run.

The upstream modules are EPL-2.0 while everything written here is Apache-2.0. Both are OSI approved, which is what Zephyr asks of tooling that never becomes part of a Zephyr image; see *Contributing External Components*.

The test network `zeth` is an ordinary tap with addresses on both sides, and a suite using it is just another host on the link. `zethL2` has no address at all, and the host is configured not to answer on it, so that the only thing replying to address resolution is Zephyr. *Setting up the network interfaces* covers creating them.

Warning

The name `zethL2` is written into three places that have to agree, and nothing checks that they do:

- `L2_INTERFACE` in tests/net/conformance/ttcn3_runner.py, which decides which interface has to exist before the test runs
- `host-interface` in the `boards/native_sim.overlay` of each test on `zethL2`, which decides where the system under test appears
- `system.pt.interface` in the configuration file of each suite on `zethL2`, which decides where the suite listens

A mismatch shows up as a suite that sees no frames and times out, not as an error.

Adding a suite A suite has two halves, one in each repository, and one decision to make before either.

Choosing the shape **Sockets or raw frames.** A suite that can say what it means over UDP or TCP extends `Zephyr_Tester`, uses the `IPL4asp` port, runs on `zeth` and needs no privilege. A suite that has to see or send the frame itself uses `Ethernet_Port`, runs on `zethL2`, has to be root, and has to answer address resolution itself because nothing else on that link will. Prefer sockets; the raw path costs a second interface and a password prompt.

Single or parallel. Single mode unless the test cases create parallel test components. Parallel costs Titan's main controller and a dependency on `expect`, and it makes the suite harder to run by hand.

The host side Create `suites/<name>` containing the TTCN-3 source, a `sources.txt` naming the module sources the suite needs, and a `<name>.cfg` with the module parameters and the list of test cases to execute. A suite that only runs test cases from a third party module needs no source of its own — `coap` is the worked example.

Take addresses and timeouts from `Zephyr_SUT`. Add any new upstream module to `modules.txt` with a pinned commit. Add a `build.conf` if the suite is parallel, privileged or works below IP.

The Zephyr side Create an application under tests/net/conformance with the layout shown above. The `src/main.c` enables the protocol, generates whatever traffic the suite needs to observe, and prints a distinctive ready line. `tests.yaml` copies the common block and names the test `net.conformance.<name>`.

Of the three pytest files, `pytest.ini` and `conftest.py` are copied verbatim; `test_<name>_conformance.py` differs only in the suite name and the ready line it waits for.

Recording what does not match Where a suite asserts behaviour that does not match the standard, say so at the point the assertion is made, so that the divergence is recorded rather than frozen in silently. Ground that no suite covers at all belongs in *Known gaps*, and a new suite is a good moment to add a row to *The suites*.

Trying it out Run the two halves by hand first, as in *Running a suite by hand*, and only go through Twister once the suite passes. Expect the first build to be slow, and remember that a missing prerequisite skips the test rather than failing it — a suite that seems to pass instantly probably never ran.

How it fits together The Zephyr side of a conformance test is only the system under test: an ordinary application, configured to enable the protocol being tested. Nothing about the test is compiled into it, there is no control channel, and the suite drives it entirely over the network.

Those applications, and the harness that runs a suite against them, live under tests/net/conformance. Twister builds and starts the application, and a small pytest harness builds the suite with Titan, runs it, and turns Titan's verdict into a test result.

Each test skips itself when Titan, the third party TTCN-3 modules or the network interface is missing, so the suites are harmless in a run that has not been set up for them. See *Running the conformance suites* for what a run needs, and *How the conformance tests are put together* for how the parts are put together.

The suites Which interface a suite uses, and whether it has to be run as root, follow from what it does: a suite that works below the IP layer reads frames from a packet socket on a link of its own.

Suite	System under test	Interface	Runs as
mdns	tests/net/conformance/mdns	`zeth`	any user
dnssd	tests/net/conformance/dnssd	`zeth`	any user
dns	tests/net/conformance/dns	`zeth`	any user
sntp	tests/net/conformance/sntp	`zeth`	any user
mqtt	tests/net/conformance/mqtt	`zeth`	any user
coap	tests/net/conformance/coap	`zeth`	any user
dhcpv4	tests/net/conformance/dhcpv4	`zeth`	root
dhcpv4_server	tests/net/conformance/dhcpv4_server	`zeth`	root
arp	tests/net/conformance/arp	`zethL2`	root
ndp	tests/net/conformance/ndp	`zethL2`	root
tcp	tests/net/conformance/tcp	`zethL2`	root

Adding a suite is described in *Adding a suite*.

Known gaps Where a suite asserts behaviour that does not match the standard, it says so at the point the assertion is made, so that the divergence is recorded rather than frozen in silently. What follows is the other kind of gap: ground no suite covers yet.

DNS-SD legacy unicast queries The hostname side of the mDNS responder answers a legacy unicast query the way **RFC 6762** section 6.7 asks. The service discovery side does not: it builds its own messages, sets the cache flush bit on the records that belong to one instance, uses the lifetimes it would have used for a multicast answer, and echoes neither the identifier nor the question. Fixing it means reworking name compression offsets that are all computed from a fixed header size.

The `dnssd` suite records this rather than asserting the standard, in `f_check_legacy_shape`, so that a test does not sit failing until somebody gets to it. Each check there says what would have to change with it.

MQTT 5.0, packet identifiers and re-sending The `mqtt` suite covers MQTT 3.1.1. Zephyr also implements MQTT 5.0 (`CONFIG_MQTT_VERSION_5_0`), and the Titan project publishes no protocol module for it, so covering it would mean writing the message types before writing any test.

Packet identifiers and re-sending are not covered either. Zephyr's client leaves both to the application: `mqtt_publish()` sends the identifier and the duplicate flag it is given, so a test of either would test the system under test's own counter rather than the client.

Overlapping DNS queries The resolver renews its source port before sending to a server that has nothing outstanding, which with the default of one query at a time means every query. Queries that overlap on one server still share a port, so the check in the `dns` suite would not catch a regression in that case. See **RFC 5452** section 9.2.

Continuous integration These suites are not part of the ordinary Twister run: they need a Titan installation, a network interface facing the device, and a checkout of the suites, none of which a normal build has. They are also slow, and they cannot run at the same time as each other.

They run nightly instead, from .github/workflows/net_conformance.yml, which can also be started by hand from the Actions tab and runs on a pull request that changes the tests, the script that runs them or the workflow itself. The job runs on a GitHub-hosted virtual machine rather than in a container, because creating a tap interface needs `CAP_NET_ADMIN` and the containers on the project's own runners do not get it. It installs the packaged Titan and then runs scripts/net/run-conformance-tests.sh, which brings up both interfaces and runs the whole directory under `sudo` so that no suite is skipped. It sets `NET_CONFORMANCE_REQUIRED`, so a suite that cannot run fails rather than skips: a run that tested nothing must not look like one that passed. The Twister report and the harness logs are kept as artifacts.

Other TTCN-3 suites The Eclipse Titan project publishes protocol modules and test ports for a large number of protocols, as separate repositories under gitlab.eclipse.org/eclipse/titan. The suites here build against those rather than defining their own message formats.

Some complete suites exist there too. `titan.misc` contains a CoAP conformance suite that can be run against the coap-server sample; see *CoAP* for that one.

An older TTCN-3 suite for TCP, written at Intel for the TCP rewrite, informed what the `tcp` suite covers, but none of its code is used. It drove Zephyr through a JSON control channel and asserted the stack's internal state names, and the option that channel needed has been removed; see the 4.5 migration guide.

Networking APIs

Zephyr provides support for the standard BSD socket APIs (defined in include/zephyr/net/socket.h) for the applications to use. See *BSD socket API* for more details.

Apart of the standard API, Zephyr provides a set of custom networking APIs and libraries for the application to use. See the list below for details.

> **Note**
>
> The legacy connectivity API in include/zephyr/net/net_context.h should not be used by applications.

Network APIs

BSD Sockets

- *Overview*
- *IPv4 and IPv6 socket options*
  - *IPv4 options*
  - *IPv6 options*

- *Secure Sockets*
  - *TLS credentials subsystem*
  - *Secure Socket Creation*
  - *Secure Sockets options*
- *Socket offloading*
  - *Offloaded socket creation*
  - *Dealing with multiple offloaded interfaces*
- *API Reference*
  - *BSD Sockets*
  - *TLS Credentials*

Overview Zephyr offers an implementation of a subset of the BSD Sockets API (a part of the POSIX standard). This API allows to reuse existing programming experience and port existing simple networking applications to Zephyr.

Here are the key requirements and concepts which governed BSD Sockets compatible API implementation for Zephyr:

- Has minimal overhead, similar to the requirement for other Zephyr subsystems.
- Is namespaced by default, to avoid name conflicts with well-known names like `close()`, which may be part of libc or other POSIX compatibility libraries. If enabled by `CONFIG_POSIX_API`, it will also expose POSIX compatible APIs.

BSD Sockets compatible API is enabled using `CONFIG_NET_SOCKETS` config option and implements the following operations: `socket()`, `close()`, `recv()`, `recvfrom()`, `send()`, `sendto()`, `connect()`, `bind()`, `listen()`, `accept()`, `fcntl()` (to set non-blocking mode), `getsockopt()`, `setsockopt()`, `poll()`, `select()`, `getaddrinfo()`, `getnameinfo()`.

Based on the namespacing requirements above, these operations are by default exposed as functions with `zsock_` prefix, e.g. `zsock_socket()` and `zsock_close()`. If the config option `CONFIG_POSIX_API` is defined, all the functions will be also exposed as aliases without the prefix. This includes the functions like `close()` and `fcntl()` (which may conflict with functions in libc or other libraries, for example, with the filesystem libraries).

Another entailment of the design requirements above is that the Zephyr API aggressively employs the short-read/short-write property of the POSIX API whenever possible (to minimize complexity and overheads). POSIX allows for calls like `recv()` and `send()` to actually process (receive or send) less data than requested by the user (on `SOCK_STREAM` type sockets). For example, a call `recv(sock, 1000, 0)` may return 100, meaning that only 100 bytes were read (short read), and the application needs to retry call(s) to receive the remaining 900 bytes.

The BSD Sockets API uses file descriptors to represent sockets. File descriptors are small integers, consecutively assigned from zero, shared among sockets, files, special devices (like stdin/stdout), etc. Internally, there is a table mapping file descriptors to internal object pointers. The file descriptor table is used by the BSD Sockets API even if the rest of the POSIX subsystem (filesystem, stdin/stdout) is not enabled.

Zephyr supports multiple types of BSD sockets, the following table summarizes what socket types are available:

Family	Type	Protocol	Description
AF_INET AF_INET6	SOCK_	IPPROTO_	Enabled if CONFIG_NET_UDP is set. Allows to send and receive UDP datagrams.
		IPPROTO_	Enabled if CONFIG_NET_SOCKETS_ENABLE_DTLS is set. Allows to send and receive DTLS datagrams.
	SOCK_	IPPROTO_	Enabled if CONFIG_NET_TCP is set. Allows to send and receive TCP data stream.
		IPPROTO_	Enabled if CONFIG_NET_SOCKETS_SOCKOPT_TLS is set. Allows to send and receive TLS data stream.
	SOCK_	IPPROTO_ <proto>	Enabled if CONFIG_NET_SOCKETS_INET_RAW is set. Allows to send and receive IPv4/IPv6 datagrams. Packets are filtered by L4 protocol specified. IPPROTO_IP is a wildcard protocol to receive all IP datagrams.
AF_PAC	SOCK_	ETH_P_ <proto>	Enabled if CONFIG_NET_SOCKETS_PACKET_DGRAM is set. Allows to send and receive packets without L2 header. Packets are filtered by L3 protocol specified. ETH_P_ALL is a wildcard protocol to receive all packets.
	SOCK_	ETH_P_	Enabled if CONFIG_NET_SOCKETS_PACKET is set. Allows to send and receive packets with L2 header included.
AF_CAN	SOCK_	CAN_RA	Enabled if CONFIG_NET_SOCKETS_CAN is set. Allows to send and receive CAN packets.

See sockets-echo-server and sockets-echo-client sample applications to learn how to create a simple server or client BSD socket based application.

IPv4 and IPv6 socket options Zephyr supports IP-level socket options through zsock_setsockopt() and zsock_getsockopt() at the NET_IPPROTO_IP (IPv4) and NET_IPPROTO_IPV6 (IPv6) protocol levels. Option availability may depend on Kconfig settings and socket address family.

IPv4 options

Socket options for IPv4

IPv6 options

Socket options for IPv6

The ZSOCK_IP_DONTFRAG and ZSOCK_IPV6_DONTFRAG options are used internally by the QUIC stack during DPLPMTUD probing (see *Path MTU Discovery (DPLPMTUD)*).

Secure Sockets Zephyr provides an extension of standard POSIX socket API, allowing to create and configure sockets with TLS protocol types, facilitating secure communication. Secure functions for the implementation are provided by Mbed TLS library. Secure sockets implementation allows use of both TLS and DTLS protocols with standard socket calls. See net_ip_protocol_secure type for supported secure protocol versions.

To enable secure sockets, set the CONFIG_NET_SOCKETS_SOCKOPT_TLS option. To enable DTLS support, use CONFIG_NET_SOCKETS_ENABLE_DTLS option.

TLS credentials subsystem TLS credentials must be registered in the system before they can be used with secure sockets. See tls_credential_add() for more information.

When a specific TLS credential is registered in the system, it is assigned with numeric value of type sec_tag_t, called a tag. This value can be used later on to reference the credential during secure socket configuration with socket options.

The following TLS credential types can be registered in the system:

- TLS_CREDENTIAL_CA_CERTIFICATE
- TLS_CREDENTIAL_PUBLIC_CERTIFICATE
- TLS_CREDENTIAL_PRIVATE_KEY
- TLS_CREDENTIAL_PSK
- TLS_CREDENTIAL_PSK_ID

An example registration of CA certificate (provided in ca_certificate array) looks like this:

```
ret = tls_credential_add(CA_CERTIFICATE_TAG, TLS_CREDENTIAL_CA_CERTIFICATE,
                         ca_certificate, sizeof(ca_certificate));
```

By default certificates in DER format are supported. PEM support can be enabled in Mbed TLS settings.

Secure Socket Creation A secure socket can be created by specifying secure protocol type, for instance:

```
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TLS_1_2);
```

The protocol version specified when creating the secure socket indicates the minimum TLS version used for the TLS session.

Once created, it can be configured with socket options. For instance, the CA certificate and hostname can be set:

```
sec_tag_t sec_tag_opt[] = {
        CA_CERTIFICATE_TAG,
};

ret = setsockopt(sock, SOL_TLS, TLS_SEC_TAG_LIST,
                 sec_tag_opt, sizeof(sec_tag_opt));
```

```
char host[] = "google.com";

ret = setsockopt(sock, SOL_TLS, TLS_HOSTNAME, host, sizeof(host));
```

Once configured, socket can be used just like a regular TCP socket.

Note

Due to mbed TLS internal data buffering and mbedtls_ssl_write() function requirements, when a non-blocking zsock_send() returns EAGAIN, it is expected that the consecutive call to zsock_send() will contain the same data as the original call.

Several samples in Zephyr use secure sockets for communication. For a sample use see e.g. echo-server sample application or HTTP GET sample application.

Secure Sockets options Secure sockets offer the following options for socket management:

Socket options for TLS

Related code samples

- HTTP ClientImplement an HTTP(S) client that issues a variety of HTTP requests.
- HTTP GET using plain socketsImplement an HTTP(S) client using plain BSD sockets.

Socket offloading Zephyr allows to register custom socket implementations (called offloaded sockets). This allows for seamless integration for devices which provide an external IP stack and expose socket-like API.

Socket offloading can be enabled with `CONFIG_NET_SOCKETS_OFFLOAD` option. A network driver that wants to register a new socket implementation should use `NET_SOCKET_OFFLOAD_REGISTER` macro. The macro accepts the following parameters:

- **`socket_name`**
  An arbitrary name for the socket implementation.
- **`prio`**
  Socket implementation's priority. The higher the priority, the earlier this particular implementation will be processed when creating a new socket. Lower numeric value indicates higher priority.
- **`_family`**
  Socket family implemented by the offloaded socket. `AF_UNSPEC` indicates any family.
- **`_is_supported`**
  A filtering function, used to verify whether a particular socket family, type and protocol are supported by the offloaded socket implementation.
- **`_handler`**
  A function compatible with `socket()` API, used to create an offloaded socket.

Every offloaded socket implementation should also implement a set of socket APIs, specified in `socket_op_vtable` struct.

The function registered for socket creation should allocate a new file descriptor using `zvfs_reserve_fd()` function. Any additional actions, specific to the creation of a particular offloaded socket implementation, should take place after the file descriptor is allocated. As a final step, if the offloaded socket was created successfully, the file descriptor should be finalized with `zvfs_finalize_typed_fd()`, or `zvfs_finalize_fd()` functions. The finalize function allows to register a `socket_op_vtable` structure implementing socket APIs for an offloaded socket along with an optional socket context data pointer.

Finally, when an offloaded network interface is initialized, it should indicate that the interface is offloaded with `net_if_socket_offload_set()` function. The function registers the function used to create an offloaded socket (the same as the one provided in `NET_SOCKET_OFFLOAD_REGISTER`) at the network interface.

Offloaded socket creation When application creates a new socket with `socket()` function, the network stack iterates over all registered socket implementations (native and offloaded). Higher priority socket implementations are processed first. For each registered socket implementation, an address family is verified, and if it matches (or the socket was registered as `AF_UNSPEC`), the corresponding `_is_supported` function is called to verify the remaining socket parameters. The first implementation that fulfills the socket requirements (i. e. `_is_supported` returns true) will create a new socket with its `_handler` function.

The above indicates the importance of the socket priority. If multiple socket implementations support the same set of socket family/type/protocol, the first implementation processed by the system will create a socket. Therefore it's important to give the highest priority to the implementation that should be the system default.

The socket priority for native socket implementation is configured with Kconfig. Use `CONFIG_NET_SOCKETS_TLS_PRIORITY` to set the priority for the native TLS sockets. Use `CONFIG_NET_SOCKETS_PRIORITY_DEFAULT` to set the priority for the remaining native sockets.

Dealing with multiple offloaded interfaces As the `socket()` function does not allow to specify which network interface should be used by a socket, it's not possible to choose a specific implementation in case multiple offloaded socket implementations, supporting the same type of sockets, are available. The same problem arises when both native and offloaded sockets are available in the system.

To address this problem, a special socket implementation (called socket dispatcher) was introduced. The sole reason for this module is to postpone the socket creation for until the first operation on a socket is performed. This leaves an opening to use `SO_BINDTODEVICE` socket option, to bind a socket to a particular network interface (and thus offloaded socket implementation). The socket dispatcher can be enabled with `CONFIG_NET_SOCKETS_OFFLOAD_DISPATCHER` Kconfig option.

When enabled, the application can specify the network interface to use with `setsockopt()` function:

```
/* A "dispatcher" socket is created */
sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

struct ifreq ifreq = {
   .ifr_name = "SimpleLink"
};

/* The socket is "dispatched" to a particular network interface
 * (offloaded or not).
 */
setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, &ifreq, sizeof(ifreq));
```

Similarly, if TLS is supported by both native and offloaded sockets, `TLS_NATIVE` socket option can be used to indicate that a native TLS socket should be created. The underlying socket can then be bound to a particular network interface:

```
/* A "dispatcher" socket is created */
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TLS_1_2);

int tls_native = 1;

/* The socket is "dispatched" to a native TLS socket implmeentation.
 * The underlying socket is a "dispatcher" socket now.
 */
setsockopt(sock, SOL_TLS, TLS_NATIVE, &tls_native, sizeof(tls_native));

struct ifreq ifreq = {
   .ifr_name = "SimpleLink"
};

/* The underlying socket is "dispatched" to a particular network interface
 * (offloaded or not).
 */
setsockopt(sock, SOL_SOCKET, SO_BINDTODEVICE, &ifreq, sizeof(ifreq));
```

In case no `SO_BINDTODEVICE` socket option is used on a socket, the socket will be dispatched according to the default priority and filtering rules on a first socket API call.

API Reference

BSD Sockets

BSD Sockets compatible API

Related code samples

- Asynchronous echo server using poll()Implement an asynchronous IPv4/IPv6 TCP echo server using BSD sockets and poll()
- Asynchronous echo server using select()Implement an asynchronous IPv4/IPv6 TCP echo server using BSD sockets and select()
- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- Dumb HTTP serverImplement a simple, portable, HTTP server using BSD sockets.
- Dumb HTTP server (multi-threaded)Implement a simple HTTP server supporting simultaneous connections using BSD sockets.
- Echo client (advanced)Implement a client that sends IP packets, waits for data to be sent back, and verifies it.
- Echo server (advanced)Implement a UDP/TCP server that sends received packets back to the sender.
- Echo server (service)Implements a simple IPv4/IPv6 TCP echo server using BSD sockets and socket service API.
- Echo server (simple)Implements a simple IPv4/IPv6 TCP echo server using BSD sockets.
- HTTP ClientImplement an HTTP(S) client that issues a variety of HTTP requests.
- HTTP GET using plain socketsImplement an HTTP(S) client using plain BSD sockets.
- Large HTTP downloadDownload a large file from a web server using BSD sockets.
- mDNS responderListen and respond to mDNS queries.
- Microsoft Azure IoT Hub MQTTConnect to Azure IoT Hub and publish messages using MQTT.
- Modbus TCP serverImplement a Modbus TCP server exposing Modbus commands to control LEDs.
- Modbus TCP-to-serial gatewayImplement a gateway between an Ethernet TCP-IP network and a Modbus serial line.
- Network management socketListen to network management events using a network management socket.
- Packet socketUse raw packet sockets over Ethernet.
- SNTP clientUse SNTP to get the current time from the host.
- SNTP serverServe the current time to SNTP clients on the network.
- SocketCANSend and receive raw CAN frames using BSD sockets API.
- SocketpairImplement communication between threads using socket pairs.
- TagoIO HTTP PostSend random temperature values to TagoIO IoT Cloud Platform using HTTP.
- UDP sender using SO_TXTIMEControl the transmission time of a packet using SO_TXTIME socket option.
- Video TCP server sinkCapture video frames and send them over the network to a TCP client.
- WebSocket ClientImplement a Websocket client that connects to a Websocket server.
- Wireguard VPNImplement a echo-server application that sends received packets back to the sender over Wireguard VPN connection.

TLS Credentials

TLS credentials management

Related code samples

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- Dumb HTTP server (multi-threaded)Implement a simple HTTP server supporting simultaneous connections using BSD sockets.
- Echo client (advanced)Implement a client that sends IP packets, waits for data to be sent back, and verifies it.
- Echo server (advanced)Implement a UDP/TCP server that sends received packets back to the sender.
- HTTP ClientImplement an HTTP(S) client that issues a variety of HTTP requests.
- HTTP GET using plain socketsImplement an HTTP(S) client using plain BSD sockets.
- HTTP ServerImplement an HTTP(s) Server demonstrating various resource types.
- Large HTTP downloadDownload a large file from a web server using BSD sockets.
- Microsoft Azure IoT Hub MQTTConnect to Azure IoT Hub and publish messages using MQTT.
- Prometheus SampleImplement a Prometheus Metric Server demonstrating various metric types.
- TagoIO HTTP PostSend random temperature values to TagoIO IoT Cloud Platform using HTTP.

Socket Services

- *Overview*
- *API Description*
- *Application Overview*
- *API Reference*

Overview The socket service API can be used to install a handler that is called if there is data received on a socket. The API helps to avoid creating a dedicated thread for each TCP or UDP service that the application provides. Instead one thread is created that serves data to multiple listening sockets which saves memory as only one thread needs to be created in the system in this case.

See sockets-service-echo sample application to learn how to create a simple BSD socket based server application using the sockets service API. The source code for this sample application can be found at: samples/net/sockets/echo_service.

API Description Socket service API is enabled using `CONFIG_NET_SOCKETS_SERVICE` config option and implements the following operations:

- `NET_SOCKET_SERVICE_SYNC_DEFINE`

  Define a network socket service. This socket service is created with extern scope so it can be used from multiple C source files.

- `NET_SOCKET_SERVICE_SYNC_DEFINE_STATIC`

Define a network socket service with static scope. This socket service can only be used within one C source file.

- `net_socket_service_register()`

  Register pollable sockets for this service. User must create the sockets before this call.

- `net_socket_service_unregister()`

  Remove pollable sockets for this service. User can close the sockets after this call.

- `net_socket_service_handler_t`

  User specified callback which is called when data is received on the listening socket.

Application Overview If the socket service API is enabled, an application must create the service like this:

```
#define MAX_BUF_LEN 1500
#define MAX_SERVICES 1

static void udp_service_handler(struct net_socket_service_event *pev)
{
     struct pollfd *pfd = &pev->event;
     int client = pfd->fd;
     struct sockaddr_in6 addr;
     socklen_t addrlen = sizeof(addr);

     /* In this example we use one static buffer in order to avoid
      * having a large stack.
      */
     static char buf[MAX_BUF_LEN];

     len = recvfrom(client, buf, sizeof(buf), 0,
                    (struct sockaddr *)&addr, &addrlen);
     if (len <= 0) {
             /* Error */
             ...
             return;
     }

     /* Do something with the received data. The pev variable contains
      * user data that was stored in the socket service when it was
      * registered.
      */
}

NET_SOCKET_SERVICE_SYNC_DEFINE_STATIC(service_udp, udp_service_handler,␣
↪MAX_SERVICES);
```

The application needs to create the sockets, then register them to the socket service after which the socket service thread will start to call the callback for any incoming data.

```
/* Create one or multiple sockets */

struct pollfd sockfd_udp[1] = { 0 };
int sock, ret;

sock = socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDP);
if (sock < 0) {
```

(continues on next page)

(continued from previous page)

```
    LOG_ERR("socket: %d", -errno);
    return -errno;
}

/* Set possible socket options after creation */
...

/* Then bind the socket to local address */
if (bind(sock, (struct sockaddr *)addr, sizeof(*addr)) < 0) {
    LOG_ERR("bind: %d", -errno);
    return -errno;
}

/* Set the polled sockets */
sockfd_udp[0].fd = sock;
sockfd_udp[0].events = POLLIN;

/* Register UDP socket to service handler */
ret = net_socket_service_register(&service_udp, sockfd_udp,
                                  ARRAY_SIZE(sockfd_udp), NULL);
if (ret < 0) {
    LOG_ERR("Cannot register socket service handler (%d)", ret);
    return ret;
}

/* Application logic happens here. When application is ready to
 * quit, one should unregister the socket service and close the
 * socket.
 */

(void)net_socket_service_unregister(&service_udp);
close(sock);
```

The TCP socket needs slightly different logic as we need to add any accepted socket to the listening socket by calling the `net_socket_service_register()` for the accepted socket.

```
struct sockaddr_in6 client_addr;
socklen_t client_addr_len = sizeof(client_addr);
struct pollfd sockfd_tcp[1] = { 0 };
int client;

/* TCP socket service is created similar way as the UDP one */
sock = socket(AF_INET6, SOCK_STREAM, IPPROTO_TCP);
if (sock < 0) {
    LOG_ERR("socket: %d", -errno);
    return -errno;
}

if (bind(sock, (struct sockaddr *)addr, sizeof(*addr)) < 0) {
    LOG_ERR("bind: %d", -errno);
    return -errno;
}

if (listen(sock, 5) < 0) {
    LOG_ERR("listen: %d", -errno);
    return -errno;
```

(continues on next page)

(continued from previous page)

```
}

while (1) {
     client = accept(tcp_sock, (struct sockaddr *)&client_addr,
                     &client_addr_len);
     if (client < 0) {
             LOG_ERR("accept: %d", -errno);
             continue;
     }

     inet_ntop(client_addr.sin6_family, &client_addr.sin6_addr,
               addr_str, sizeof(addr_str));
     LOG_INF("Connection from %s (%d)", addr_str, client);

     sockfd_tcp[0].fd = client;
     sockfd_tcp[0].events = POLLIN;

     /* Register all the sockets to service handler */
     ret = net_socket_service_register(&service_tcp, sockfd_tcp,
                                       ARRAY_SIZE(sockfd_tcp), NULL);
     if (ret < 0) {
             LOG_ERR("Cannot register socket service handler (%d)", ret);
             break;
     }
}
```

For any closed TCP client connection we need to remove the closed socket from the polled socket list.

```
/* If the TCP socket is closed while reading the data in the handler,
 * mark it as non pollable.
 */
if (sockfd_tcp[0].fd == client) {
     sockfd_tcp[0].fd = -1;

     /* Update the handler so that client connection is
      * not monitored any more.
      */
      (void)net_socket_service_register(&service_tcp, sockfd_tcp,
                                        ARRAY_SIZE(sockfd_tcp), NULL);
      close(client);

      LOG_INF("Connection from %s closed", addr_str);
}
```

Please see a more complete example in the echo_service sample source code at samples/net/sockets/echo_service/src/main.c.

API Reference

BSD socket service API

IPv4/IPv6 Primitives and Helpers

- *Overview*
- *API Reference*

Overview Miscellaneous defines and helper functions for IP addresses and IP protocols.

API Reference

IPv4/IPv6 primitives and helpers

DNS Resolve

- *Overview*
- *Network management events*
- *Sample usage*
- *API Reference*

Overview The DNS resolver implements a basic DNS resolver according to **RFC 1035**. Supported DNS answers are IPv4/IPv6 addresses and CNAME.

If a CNAME is received, the DNS resolver will create another DNS query. The number of additional queries is controlled by the `CONFIG_DNS_RESOLVER_ADDITIONAL_QUERIES` Kconfig variable.

The multicast DNS (mDNS) client resolver support can be enabled by setting `CONFIG_MDNS_RESOLVER` Kconfig option. See **RFC 6762** for more details about mDNS.

The link-local multicast name resolution (LLMNR) client resolver support can be enabled by setting the `CONFIG_LLMNR_RESOLVER` Kconfig variable. See **RFC 4795** for more details about LLMNR.

For more information about DNS configuration variables, see: subsys/net/lib/dns/Kconfig. The DNS resolver API can be found at include/zephyr/net/dns_resolve.h.

DNS-based service discovery queries described in **RFC 6763** can be done by `dns_resolve_service()` API. The returned service descriptions are passed to user supplied callback and the API sets the address family to `AF_LOCAL` to indicate that the value is not an IPv4 or IPv6 address but a service description.

Example:

```
#include <zephyr/net/dns_resolve.h>

#define MAX_STR_LEN CONFIG_DNS_RESOLVER_MAX_NAME_LEN

static void dns_result_cb(enum dns_resolve_status status,
                          struct dns_addrinfo *info,
                          void *user_data)
{
     if (status == DNS_EAI_CANCELED) {
             /* dns: Timeout while resolving name */
             return;
     }

     if (status == DNS_EAI_INPROGRESS && info) {
             char str[MAX_STR_LEN + 1];
```

(continues on next page)

(continued from previous page)

```
            if (info->ai_family == NET_AF_INET) {
                    net_addr_ntop(NET_AF_INET,
                                  &net_sin(&info->ai_addr)->sin_addr,
                                  str, NET_IPV4_ADDR_LEN);
            } else if (info->ai_family == NET_AF_INET6) {
                    net_addr_ntop(NET_AF_INET6,
                                  &net_sin6(&info->ai_addr)->sin6_addr,
                                  str, NET_IPV6_ADDR_LEN);
            } else if (info->ai_family == AF_LOCAL) {
                    /* service discovery */
                    memset(str, 0, MAX_STR_LEN);
                    memcpy(str, info->ai_canonname,
                           MIN(info->ai_addrlen, MAX_STR_LEN));
            } else {
                    strncpy(str, "Invalid proto family", MAX_STR_LEN + 1);
            }

            str[MAX_STR_LEN] = '\0';

            printk("dns: %s\n", str);
            return;
    }

    if (status == DNS_EAI_ALLDONE) {
            printk("dns: All results received\n");
            return;
    }

    if (status == DNS_EAI_FAIL) {
            printk("dns: No such name found.\n");
            return;
    }

    printk("dns: Unhandled status %d received (errno %d)\n", status, errno);
}

#define DNS_TIMEOUT (MSEC_PER_SEC * 5) /* in ms */

static void discover_service(void)
{
    int ret = dns_resolve_service(dns_resolve_get_default(),
                                  "_http._tcp.dns-sd.org",
                                  NULL, dns_result_cb,
                                  NULL, DNS_TIMEOUT);
    ...
}
```

The above query would return output like this:

As the service discovery query could return long strings and the packet size could be large, you might need to adjust following Kconfig options:

- `CONFIG_DNS_RESOLVER_MAX_ANSWER_SIZE`. This tells the maximum size of the answer, typical value for this option could be 1024. The default size for this option is 512 bytes.
- `CONFIG_DNS_RESOLVER_MAX_NAME_LEN`. This tells the maximum length of the returned name. The value depends on your expected data size, typical value might be 128 bytes.

Network management events The DNS resolver raises the following NET_MGMT_EVENT events:

- NET_EVENT_DNS_SERVER_ADD — a single DNS server slot was added to the resolver. Raised once per slot, with the originating interface as the event payload.
- NET_EVENT_DNS_SERVER_DEL — a single DNS server slot was removed, with the originating interface as the event payload.
- NET_EVENT_DNS_SERVERS_RECONFIGURED — the DNS server configuration was refreshed via dns_resolve_reconfigure() or dns_resolve_reconfigure_with_interfaces(). Raised on every successful call regardless of whether individual server slots actually changed, with NULL iface (system-level event).

ADD and DEL are *delta* events: when dns_resolve_reconfigure() is called with a server set identical to the existing one, neither ADD nor DEL fires — the resolver intentionally suppresses them to avoid cancelling in-flight queries on DHCP-offer retransmit and IPv6 RA. Consumers that need a "DNS configuration is ready again" gate after a network event (for example, after a PPP iface re-establishes and pushes the same DNS servers via IPCP on every PM resume cycle) should listen for NET_EVENT_DNS_SERVERS_RECONFIGURED rather than NET_EVENT_DNS_SERVER_ADD.

Sample usage See dns-resolve sample application for details.

API Reference
DNS Resolve Library

Related code samples

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- DNS resolveResolve an IP address for a given hostname.
- TagoIO HTTP PostSend random temperature values to TagoIO IoT Cloud Platform using HTTP.

Datagram PLPMTUD API

- *Overview*
- *Configuration*
- *Usage model*
- *API Reference*

Overview Zephyr provides a generic **Datagram Packetization Layer Path MTU Discovery** (DPLPMTUD) implementation based on **RFC 8899**. It is intended for UDP-based transports (QUIC, CoAP over UDP, custom protocols) that must discover how large a datagram payload can be without relying on local IP fragmentation.

The subsystem splits responsibilities as follows:

Generic stack (subsys/net/ip/dplpmtud.c)

- Per-destination search state: validated PLPMTU, probe size, retry count, bounds
- Binary search between the base PLPMTU (1200 bytes) and the path ceiling
- Integration with the existing PMTU destination cache (ICMP PTB in, validated size out)
- Black-hole handling when the PMTU cache reports an MTU below the base PLPMTU

Transport consumer

- Construct probe datagrams at the size returned by `net_dplpmtud_get_path_probe_size()`
- Send probes with don't fragment enabled (see *IPv4 and IPv6 socket options* and `ZSOCK_IP_DONTFRAG` / `ZSOCK_IPV6_DONTFRAG`)
- Map transport ACK/loss to `net_dplpmtud_on_path_probe_acked()` and `net_dplpmtud_on_path_probe_lost()`

QUIC is the first in-tree consumer. Other transports can use the same API without duplicating the RFC 8899 state machine.

Configuration Enable DPLPMTUD per address family:

- `CONFIG_NET_IPV4_PMTU` and `CONFIG_NET_IPV4_PMTU_DPLPMTUD` for IPv4
- `CONFIG_NET_IPV6_PMTU` and `CONFIG_NET_IPV6_PMTU_DPLPMTUD` for IPv6

Both families also require `CONFIG_NET_UDP`. The umbrella option `CONFIG_NET_PMTU_DPLPMTUD` is selected when either family option is enabled.

ICMP-based PMTU reduction (Packet Too Big) remains available through `CONFIG_NET_IPV4_PMTU_PTB` and `CONFIG_NET_IPV6_PMTU_PTB`. DPLPMTUD complements PTB by actively probing for larger sizes when the path allows it.

The number of concurrent paths tracked matches `CONFIG_NET_IPV4_PMTU_DESTINATION_CACHE_ENTRIES` and `CONFIG_NET_IPV6_PMTU_DESTINATION_CACHE_ENTRIES` respectively.

Usage model Initialize one `net_dplpmtud_path` per remote destination (or reuse the handle for the lifetime of a connection):

```
struct net_dplpmtud_path path;
uint16_t max_plpmtu = 1452U; /* transport / peer limit, or 0 if uncapped */
int mtu;
int probe;
int ret;

ret = net_dplpmtud_init_path(&path, &remote_addr, max_plpmtu);
if (ret < 0) {
    /* handle error */
}

mtu = net_dplpmtud_get_path_mtu(&path);
if (mtu < 0) {
    mtu = NET_DPLPMTUD_BASE_PLPMTU;
}

/* Application data must fit in @a mtu bytes (transport framing excluded). */
```

Probe lifecycle:

1. Call `net_dplpmtud_get_path_probe_size()`. A return value of `0` means no probe is needed. The path must already be initialized; getters do not create cache entries.
2. If `net_dplpmtud_path_probe_in_flight()` is false, build and send a probe datagram of that size with don't fragment set (socket option or per-datagram control message in `zsock_sendmsg()`).
3. Call `net_dplpmtud_on_path_probe_sent()` to reserve the probe in the generic state machine, then transmit the probe datagram.
4. If the send fails, call `net_dplpmtud_on_path_probe_lost()` so core and transport state stay aligned.

5. On transport confirmation, call `net_dplpmtud_on_path_probe_acked()` or `net_dplpmtud_on_path_probe_lost()`.

Update `net_dplpmtud_set_path_max_plpmtu()` when the transport learns a new ceiling (for example a QUIC `max_udp_payload_size` transport parameter).

Call `net_dplpmtud_note_path_blackhole()` if the transport detects a black hole independent of ICMP. PTB updates below the base PLPMTU are applied automatically through the PMTU cache.

API Reference

Datagram PLPMTUD API

Network Management

- *Overview*
- *Requesting a defined procedure*
- *Listening to network events*
- *Defining a network management procedure*
- *Signaling a network event*
- *API Reference*

Overview The Network Management APIs allow applications, as well as network layer code itself, to call defined network routines at any level in the IP stack, or receive notifications on relevant network events. For example, by using these APIs, application code can request a scan be done on a Wi-Fi- or Bluetooth-based network interface, or request notification if a network interface IP address changes.

The Network Management API implementation is designed to save memory by eliminating code at build time for management routines that are not used. Distinct and statically defined APIs for network management procedures are not used. Instead, defined procedure handlers are registered by using a `NET_MGMT_REGISTER_REQUEST_HANDLER` macro. Procedure requests are done through a single `net_mgmt()` API that invokes the registered handler for the corresponding request.

The current implementation is experimental and may change and improve in future releases.

Requesting a defined procedure All network management requests are of the form `net_mgmt(mgmt_request, ...)`. The `mgmt_request` parameter is a bit mask that tells which stack layer is targeted, if a `net_if` object is implied, and the specific management procedure being requested. The available procedure requests depend on what has been implemented in the stack.

To avoid extra cost, all `net_mgmt()` calls are direct. Though this may change in a future release, it will not affect the users of this function.

Listening to network events You can receive notifications on network events by registering a callback function and specifying a set of events used to filter when your callback is invoked. The callback will have to be unique for a pair of layer and code, whereas on the command part it will be a mask of events.

At runtime two functions are available, `net_mgmt_add_event_callback()` for registering the callback function, and `net_mgmt_del_event_callback()` for unregistering a callback. A helper function, `net_mgmt_init_event_callback()`, can be used to ease the initialization of the callback structure.

Additionally `NET_MGMT_REGISTER_EVENT_HANDLER` can be used to register a callback handler at compile time.

When an event occurs that matches a callback's event set, the associated callback function is invoked with the actual event code. This makes it possible for different events to be handled by the same callback function, if desired.

Warning

Event set filtering allows false positives for events that have the same layer and layer code. A callback handler function **must** check the event code (passed as an argument) against the specific network events it will handle, **regardless** of how many events were in the set passed to `net_mgmt_init_event_callback()`.

Note that in order to receive events from multiple layers, one must have multiple listeners registered, one for each layer being listened. The callback handler function can be shared between different layer events.

(False positives can occur for events which have the same layer and layer code.)

An example follows.

```
/*
 * Set of events to handle.
 * See e.g. include/zephyr/net/net_event.h for some NET_EVENT_xxx values.
 */
#define EVENT_IFACE_SET (NET_EVENT_IF_xxx | NET_EVENT_IF_yyy)
#define EVENT_IPV4_SET (NET_EVENT_IPV4_xxx | NET_EVENT_IPV4_yyy)

struct net_mgmt_event_callback iface_callback;
struct net_mgmt_event_callback ipv4_callback;

void callback_handler(struct net_mgmt_event_callback *cb,
                      uint64_t mgmt_event,
                      struct net_if *iface)
{
        if (mgmt_event == NET_EVENT_IF_xxx) {
                /* Handle NET_EVENT_IF_xxx */
        } else if (mgmt_event == NET_EVENT_IF_yyy) {
                /* Handle NET_EVENT_IF_yyy */
        } else if (mgmt_event == NET_EVENT_IPV4_xxx) {
                /* Handle NET_EVENT_IPV4_xxx */
        } else if (mgmt_event == NET_EVENT_IPV4_yyy) {
                /* Handle NET_EVENT_IPV4_yyy */
        } else {
                /* Spurious (false positive) invocation. */
        }
}

void register_cb(void)
{
        net_mgmt_init_event_callback(&iface_callback, callback_handler,
                                     EVENT_IFACE_SET);
        net_mgmt_init_event_callback(&ipv4_callback, callback_handler,
                                     EVENT_IPV4_SET);
        net_mgmt_add_event_callback(&iface_callback);
        net_mgmt_add_event_callback(&ipv4_callback);
}
```

Or similarly using NET_MGMT_REGISTER_EVENT_HANDLER.

> **Note**
>
> The info and info_length arguments are only usable if CONFIG_NET_MGMT_EVENT_INFO is enabled. Otherwise these are NULL and zero.

```
/*
 * Set of events to handle.
 */
#define EVENT_IFACE_SET (NET_EVENT_IF_xxx | NET_EVENT_IF_yyy)
#define EVENT_IPV4_SET (NET_EVENT_IPV4_xxx | NET_EVENT_IPV4_yyy)

static void event_handler(uint64_t mgmt_event, struct net_if *iface,
                          void *info, size_t info_length,
                          void *user_data)
{
        if (mgmt_event == NET_EVENT_IF_xxx) {
                /* Handle NET_EVENT_IF_xxx */
        } else if (mgmt_event == NET_EVENT_IF_yyy) {
                /* Handle NET_EVENT_IF_yyy */
        } else if (mgmt_event == NET_EVENT_IPV4_xxx) {
                /* Handle NET_EVENT_IPV4_xxx */
        } else if (mgmt_event == NET_EVENT_IPV4_yyy) {
                /* Handle NET_EVENT_IPV4_yyy */
        } else {
                /* Spurious (false positive) invocation. */
        }
}

NET_MGMT_REGISTER_EVENT_HANDLER(iface_event_handler, EVENT_IFACE_SET,
                                event_handler, NULL);
NET_MGMT_REGISTER_EVENT_HANDLER(ipv4_event_handler, EVENT_IPV4_SET,
                                event_handler, NULL);
```

See include/zephyr/net/net_event.h for available generic core events that can be listened to.

Defining a network management procedure You can provide additional management procedures specific to your stack implementation by defining a handler and registering it with an associated mgmt_request code.

Management request code are defined in relevant places depending on the targeted layer or eventually, if l2 is the layer, on the technology as well. For instance, all IP layer management request code will be found in the include/zephyr/net/net_event.h header file. But in case of an L2 technology, let's say Ethernet, these would be found in include/zephyr/net/ethernet.h

You define your handler modeled with this signature:

```
static int your_handler(uint64_t mgmt_event, struct net_if *iface,
                        void *data, size_t len);
```

and then register it with an associated mgmt_request code:

```
NET_MGMT_REGISTER_REQUEST_HANDLER(<mgmt_request code>, your_handler);
```

This new management procedure could then be called by using:

```
net_mgmt(<mgmt_request code>, ...);
```

Signaling a network event You can signal a specific network event using the `net_mgmt_event_notify()` function and provide the network event code. See include/zephyr/net/net_mgmt.h for details. As for the management request code, event code can be also found on specific L2 technology mgmt headers, for example include/zephyr/net/ieee802154_mgmt.h would be the right place if 802.15.4 L2 is the technology one wants to listen to events.

API Reference
Network Management

Related code samples

- DHCPv4 clientStart a DHCPv4 client to obtain an IPv4 address from a DHCPv4 server.
- DNS resolveResolve an IP address for a given hostname.
- IPv4 autoconf clientPerform IPv4 autoconfiguration and self-assign a random IPv4 address
- Telnet consoleAccess Zephyr shell over telnet.

Network Statistics

- *Overview*
- *API Reference*

Overview Network statistics are collected if `CONFIG_NET_STATISTICS` is set. Individual component statistics for IPv4 or IPv6 can be turned off if those statistics are not needed. See various options in subsys/net/ip/Kconfig.stats file for details.

By default, the system collects network statistics per network interface. This can be controlled by `CONFIG_NET_STATISTICS_PER_INTERFACE` option.

The `CONFIG_NET_STATISTICS_USER_API` option can be set if the application wants to collect statistics for further processing. The network management interface API is used for that. See *Network Management* for details.

The `CONFIG_NET_STATISTICS_ETHERNET` option can be set to collect generic Ethernet statistics. If the `CONFIG_NET_STATISTICS_ETHERNET_VENDOR` option is set, then Ethernet device driver can collect Ethernet device specific statistics. These statistics can then be transferred to application for processing.

If the `CONFIG_NET_SHELL` option is set, then network shell can show statistics information with `net stats` command.

API Reference
Network Statistics Library

Related code samples

- Network statisticsQuery and display network statistics from a user application.

- OpenThread shellTest Thread and IEEE 802.15.4 using the OpenThread shell.
- Wi-Fi shellTest Wi-Fi functionality using the Wi-Fi shell module.

Network Timeout

- *Overview*
- *Use*
- *API Reference*

Overview Zephyr's network infrastructure mostly uses the millisecond-resolution uptime clock to track timeouts, with both deadlines and durations measured with 32-bit unsigned values. The 32-bit value rolls over at 49 days 17 hours 2 minutes 47.296 seconds.

Timeout processing is often affected by latency, so that the time at which the timeout is checked may be some time after it should have expired. Handling this correctly without arbitrary expectations of maximum latency requires that the maximum delay that can be directly represented be a 31-bit non-negative number (`INT32_MAX`), which overflows at 24 days 20 hours 31 minutes 23.648 seconds.

Most network timeouts are shorter than the delay rollover, but a few protocols allow for delays that are represented as unsigned 32-bit values counting seconds, which corresponds to a 42-bit millisecond count.

The net_timeout API provides a generic timeout mechanism to correctly track the remaining time for these extended-duration timeouts.

Use The simplest use of this API is:

1. Configure a network timeout using `net_timeout_set()`.
2. Use `net_timeout_evaluate()` to determine how long it is until the timeout occurs. Schedule a timeout to occur after this delay.
3. When the timeout callback is invoked, use `net_timeout_evaluate()` again to determine whether the timeout has completed, or whether there is additional time remaining. If the latter, reschedule the callback.
4. While the timeout is running, use `net_timeout_remaining()` to get the number of seconds until the timeout expires. This may be used to explicitly update the timeout, which should be done by canceling any pending callback and restarting from step 1 with the new timeout.

The `net_timeout` contains a `sys_snode_t` that allows multiple timeout instances to be aggregated to share a single kernel timer element. The application must use `net_timeout_evaluate()` on all instances to determine the next timeout event to occur.

`net_timeout_deadline()` may be used to reconstruct the full-precision deadline of the timeout. This exists primarily for testing but may have use in some applications, as it does allow a millisecond-resolution calculation of remaining time.

API Reference

Network long timeout primitives and helpers

Networking Context The net_context API is not meant for application use. Application should use *BSD Sockets* API instead.

Promiscuous Mode

- *Overview*
- *Sample usage*
- *API Reference*

Overview Promiscuous mode is a mode for a network interface controller that causes it to pass all traffic it receives to the application rather than passing only the frames that the controller is specifically programmed to receive. This mode is normally used for packet sniffing as used to diagnose network connectivity issues by showing an application all the data being transferred over the network. (See the Wikipedia article on promiscuous mode for more information.)

The network promiscuous APIs are used to enable and disable this mode, and to wait for and receive a network data to arrive. Not all network technologies or network device drivers support promiscuous mode.

Sample usage First the promiscuous mode needs to be turned ON by the application like this:

```
ret = net_promisc_mode_on(iface);
if (ret < 0) {
        if (ret == -EALREADY) {
                printf("Promiscuous mode already enabled\n");
        } else {
                printf("Cannot enable promiscuous mode for "
                       "interface %p (%d)\n", iface, ret);
        }
}
```

If there is no error, then the application can start to wait for network data:

```
while (true) {
        pkt = net_promisc_mode_wait_data(K_FOREVER);
        if (pkt) {
                print_info(pkt);
        }

        net_pkt_unref(pkt);
}
```

Finally the promiscuous mode can be turned OFF by the application like this:

```
ret = net_promisc_mode_off(iface);
if (ret < 0) {
        if (ret == -EALREADY) {
                printf("Promiscuous mode already disabled\n");
        } else {
                printf("Cannot disable promiscuous mode for "
                       "interface %p (%d)\n", iface, ret);
        }
}
```

See net-promiscuous-mode for a more comprehensive example.

API Reference
Promiscuous mode

Related code samples

- Promiscuous modeEnable promiscuous mode on all interfaces and print information about incoming packets.

Simple Network Time Protocol Library

- *Overview*
- *API Reference*

Overview The SNTP library implements **RFC 4330**.

SNTP provides a way to synchronize clocks in computer networks.

The client (`CONFIG_SNTP`) queries the time from an SNTP server. The server (`CONFIG_SNTP_SERVER`) answers such queries on UDP port 123, on every enabled address family. The application is responsible for setting the system clock, and tells the server where its time comes from with `sntp_server_clock_source()`. Until it does so, the server answers with the leap indicator set to "clock not synchronized" and stratum 16, so that clients discard its timestamps.

API Reference

SNTP

Related code samples

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- SNTP clientUse SNTP to get the current time from the host.
- SNTP serverServe the current time to SNTP clients on the network.

SNTP server

Related code samples

- SNTP serverServe the current time to SNTP clients on the network.

SOCKS5 Proxy Support

- *Overview*
- *SOCKS5 API*
- *SOCKS5 Proxy Usage in MQTT*

Overview The SOCKS library implements SOCKS5 support, which allows Zephyr to connect to peer devices via a network proxy.

See this SOCKS5 Wikipedia article for a detailed overview of how SOCKS5 works.

For more information about the protocol itself, see **RFC 1928**.

SOCKS5 API The SOCKS5 support is enabled by `CONFIG_SOCKS` Kconfig variable. Application wanting to use the SOCKS5 must set the SOCKS5 proxy host address by calling `setsockopt()` like this:

```
static int set_proxy(int sock, const struct sockaddr *proxy_addr,
                     socklen_t proxy_addrlen)
{
    int ret;

    ret = setsockopt(sock, SOL_SOCKET, SO_SOCKS5,
                     proxy_addr, proxy_addrlen);
    if (ret < 0) {
            return -errno;
    }

    return 0;
}
```

SOCKS5 Proxy Usage in MQTT For MQTT client, there is `mqtt_client_set_proxy()` API that the application can call to setup SOCKS5 proxy. See mqtt-publisher sample application for usage example.

Trickle Timer Library

- *Overview*
- *API Reference*

Overview The Trickle timer library implements **RFC 6206**.

The Trickle algorithm allows nodes in a lossy shared medium (e.g., low-power and lossy networks) to exchange information in a highly robust, energy efficient, simple, and scalable manner.

API Reference
Trickle Algorithm Library

Websocket Client API

- *Overview*
- *Websocket Transport*
- *API Reference*

Overview The Websocket client library allows Zephyr to connect to a Websocket server. The Websocket client API can be used directly by application to establish a Websocket connection to server, or it can be used as a transport for other network protocols like MQTT.

See this Websocket Wikipedia article for a detailed overview of how Websocket works.

For more information about the protocol itself, see **RFC 6455**.

Websocket Transport The Websocket API allows it to be used as a transport for other high level protocols like MQTT. The Zephyr MQTT client library can be configured to use Websocket transport by enabling `CONFIG_MQTT_LIB_WEBSOCKET` and `CONFIG_WEBSOCKET_CLIENT` Kconfig options.

First a socket needs to be created and connected to the Websocket server:

```
sock = socket(family, SOCK_STREAM, IPPROTO_TCP);
...
ret = connect(sock, addr, addr_len);
...
```

The Websocket transport socket is then created like this:

```
ws_sock = websocket_connect(sock, &config, timeout, user_data);
```

The Websocket socket can then be used to send or receive data, and the Websocket client API will encapsulate the sent or received data to/from Websocket packet payload. Both the `websocket_xxx()` API or normal BSD socket API functions can be used to send and receive application data.

```
ret = websocket_send_msg(ws_sock, buf_to_send, buf_len,
                         WEBSOCKET_OPCODE_DATA_BINARY, true, true,
                         K_FOREVER);
...
ret = send(ws_sock, buf_to_send, buf_len, 0);
```

If normal BSD socket functions are used, then currently only TEXT data is supported. In order to send BINARY data, the `websocket_send_msg()` must be used.

When done, the Websocket transport socket must be closed. User should handle the lifecycle(close/reuse) of tcp socket after websocket_disconnect.

```
ret = close(ws_sock);
or
ret = websocket_disconnect(ws_sock);
```

API Reference

Websocket API

Related code samples

- WebSocket ClientImplement a Websocket client that connects to a Websocket server.

Network Packet Capture

- *Overview*
- *Cooked Mode Capture*
- *Sample usage*
- *API Reference*

Overview The `net_capture` API allows user to monitor the network traffic in one of the Zephyr network interfaces and send that traffic to external system for analysis. The monitoring can be setup either manually using `net-shell` or automatically by using the `net_capture` API.

Cooked Mode Capture If capturing is enabled and configured, the system will automatically capture network traffic for a given network interface. If you would like to capture network data when there is no network interface involved, then you need to use the cooked mode capture API.

In cooked mode capture, arbitrary network packets can be captured and there does not need to be network interface involved. For example low level HDLC packets in PPP can be captured, as the HDLC L2 layer data is stripped away when using the normal network interface based capture. Also CANBUS or Bluetooth network data could be captured although currently there is no support in the network stack to capture those.

The cooked mode capture works like this:

- An `any` network interface is created. It acts as a sink where the cooked mode captured packets are written by the cooked mode capture API.
- A `cooked` virtual network interface is attached on top of this `any` interface.
- The `cooked` interface must be configured to capture certain L2 packet types using the network interface configuration API.
- When cooked mode capture API is used, the caller must specify what is the layer 2 protocol type of the captured data. The cooked mode capture API is then able to determine what to capture when receiving such a L2 packet.
- The network packet capturing infrastructure is then setup so that the `cooked` interface is marked as captured network interface. The packets received by the `cooked` interface via the `any` interface are then automatically placed to the capture IP tunnel and sent to remote host for analysis.

For example, in the sample capture application, these network interfaces are created:

```
Interface any (0x808ab3c) (Dummy) [1]
=================================
Virtual interfaces attached to this : 2
Device    : NET_ANY (0x80849a4)

Interface cooked (0x808ac94) (Virtual) [2]
==================================
Virtual name : Cooked mode capture
Attached  : 1 (Dummy / 0x808ab3c)
Device    : NET_COOKED (0x808497c)

Interface eth0 (0x808adec) (Ethernet) [3]
===================================
Virtual interfaces attached to this : 4
Device    : zeth0 (0x80849b8)
IPv6 unicast addresses (max 4):
     fe80::5eff:fe00:53e6 autoconf preferred infinite
     2001:db8::1 manual preferred infinite
IPv4 unicast addresses (max 2):
     192.0.2.1/255.255.255.0 overridable preferred infinite

Interface net0 (0x808af44) (Virtual) [4]
==================================
Virtual name : Capture tunnel
Attached  : 3 (Ethernet / 0x808adec)
Device    : IP_TUNNEL0 (0x8084990)
IPv6 unicast addresses (max 4):
     2001:db8:200::1 manual preferred infinite
     fe80::efed:6dff:fef2:b1df autoconf preferred infinite
     fe80::56da:1eff:fe5e:bc02 autoconf preferred infinite
```

In this example, the `192.0.2.2` is the address of the outer end point of the host that terminates the tunnel. Zephyr uses this address to select the internal interface to use for the tunnel. In this example it is interface 3.

The interface 2 is a virtual interface that runs on top of interface 1. The cooked capture packets are written by the capture API to sink interface 1. The packets propagate to interface 2 because it is linked to the first interface. The `net capture enable 2` net-shell command will cause the packets sent to interface 2 to be written to capture interface 4, which in turn then capsulates the packets and tunnels them to peer via the Ethernet interface 3.

The above IP addresses might change if you change the addresses in the sample samples/net/capture/overlay-tunnel.conf file.

Sample usage See net-capture sample application and *Monitor Network Traffic* for details.

API Reference
Network packet capture

Related code samples

- Network packet captureCapture network packets and send them to a remote host via IPIP tunnel.

Network Packet Management

Packet Management

- *Overview*
  - *Architectural notes*
  - *Concurrency and Thread Safety*
- *Memory management*
  - *Allocation*
  - *Buffer allocation*
  - *Deallocation*
- *Operations*
  - *Read and Write access*
  - *Data access*
- *API Reference*

Overview Network packets are the main data the networking stack manipulates. Such data is represented through the net_pkt structure which provides a means to hold the packet, write and read it, as well as necessary metadata for the core to hold important information. Such an object is called net_pkt in this document.

The data structure and the whole API around it are defined in include/zephyr/net/net_pkt.h.

Architectural notes There are two network packets flows within the stack, **TX** for the transmission path, and **RX** for the reception one. In both paths, each net_pkt is written and read from the beginning to the end, or more specifically from the headers to the payload.

Concurrency and Thread Safety The `net_pkt` structure and its associated APIs are **not thread-safe**. The network stack relies on a strict **Exclusive Ownership** model. When a network packet is created or received, it is owned by a single thread or execution context at any given time.

Concurrency is managed using the following primary patterns:

- **Ownership Transfer via FIFOs:** The most common lifecycle of a `net_pkt` involves passing it between isolated execution contexts (e.g., from an RX driver thread to the IP stack). Once a packet is enqueued, the sender drops its reference and loses access.
- **Shallow Cloning ("net_pkt_shallow_clone"):** If a packet must be retained by one layer (such as TCP for potential retransmission) while simultaneously processed by another, `net_pkt_shallow_clone()` is used to create a new wrapper pointing to the same underlying read-only data (the data buffers themselves are thread-safely reference counted).
- **Coarse-Grained Protocol Locks:** When packets are deliberately held in memory queues, they are protected by higher-level subsystem locks (such as connection mutexes).

The `atomic_ref` field within `struct net_pkt` is for memory lifecycle management (preventing Use-After-Free conditions) and is not a lock for concurrent mutation.

Memory management

Allocation All net_pkt objects come from a pre-defined pool of struct net_pkt. Such pool is defined via

```
NET_PKT_SLAB_DEFINE(name, count)
```

Note, however, one will rarely have to use it, as the core provides already two pools, one for the TX path and one for the RX path.

Allocating a raw net_pkt can be done through:

```
pkt = net_pkt_alloc(timeout);
```

However, by its nature, a raw net_pkt is useless without a buffer and needs various metadata information to become relevant as well. It requires at least to get the network interface it is meant to be sent through or through which it was received. As this is a very common operation, a helper exist:

```
pkt = net_pkt_alloc_on_iface(iface, timeout);
```

A more complete allocator exists, where both the net_pkt and its buffer can be allocated at once:

```
pkt = net_pkt_alloc_with_buffer(iface, size, family, proto, timeout);
```

See below how the buffer is allocated.

Buffer allocation The net_pkt object does not define its own buffer, but instead uses an existing object for this: `net_buf`. (See *Network Buffers* for more information). However, it mostly hides the usage of such a buffer because net_pkt brings network awareness to buffer allocation and, as we will see later, its operation too.

To allocate a buffer, a net_pkt needs to have at least its network interface set. This works if the family of the packet is unknown at the time of buffer allocation. Then one could do:

```
net_pkt_alloc_buffer(pkt, size, proto, timeout);
```

Where proto could be 0 if unknown (there is no IPPROTO_UNSPEC).

As seen previously, the net_pkt and its buffer can be allocated at once via `net_pkt_alloc_with_buffer()`. It is actually the most widely used allocator.

The network interface, the family, and the protocol of the packet are used by the buffer allocation to determine if the requested size can be allocated. Indeed, the allocator will use the network interface to know the MTU and then the family and protocol for the headers space (if only these 2 are specified). If the whole fits within the MTU, the allocated space will be of the requested size plus, eventually, the headers space. If there is insufficient MTU space, the requested size will be shrunk so the possible headers space and new size will fit within the MTU.

For instance, on an Ethernet network interface, with an MTU of 1500 bytes:

```
pkt = net_pkt_alloc_with_buffer(iface, 800, NET_AF_INET4, IPPROTO_UDP, K_FOREVER);
```

will successfully allocate 800 + 20 + 8 bytes of buffer for the new net_pkt where:

```
pkt = net_pkt_alloc_with_buffer(iface, 1600, NET_AF_INET4, IPPROTO_UDP, K_FOREVER);
```

will successfully allocate 1500 bytes, and where 20 + 8 bytes (IPv4 + UDP headers) will not be used for the payload.

On the receiving side, when the family and protocol are not known:

```
pkt = net_pkt_rx_alloc_with_buffer(iface, 800, AF_UNSPEC, 0, K_FOREVER);
```

will allocate 800 bytes and no extra header space. But a:

```
pkt = net_pkt_rx_alloc_with_buffer(iface, 1600, AF_UNSPEC, 0, K_FOREVER);
```

will allocate 1514 bytes, the MTU + Ethernet header space.

One can increase the amount of buffer space allocated by calling `net_pkt_alloc_buffer()`, as it will take into account the existing buffer. It will also account for the header space if net_pkt's family is a valid one, as well as the proto parameter. In that case, the newly allocated buffer space will be appended to the existing one, and not inserted in the front. Note however such a use case is rather limited. Usually, one should know from the start how much size should be requested.

Deallocation Each net_pkt is reference counted. At allocation, the reference is set to 1. The reference count can be incremented with `net_pkt_ref()` or decremented with `net_pkt_unref()`. When the count drops to zero the buffer is also un-referenced and net_pkt is automatically placed back into the free net_pkt_slabs

If net_pkt's buffer is needed even after net_pkt deallocation, one will need to reference once more all the chain of net_buf before calling last net_pkt_unref. See *Network Buffers* for more information.

Operations There are two ways to access the net_pkt buffer, explained in the following sections: basic read/write access and data access, the latter being the preferred way.

Read and Write access As said earlier, though net_pkt uses net_buf for its buffer, it provides its own API to access it. Indeed, a network packet might be scattered over a chain of net_buf objects, the functions provided by net_buf are then limited for such case. Instead, net_pkt provides functions which hide all the complexity of potential non-contiguous access.

Data movement into the buffer is made through a cursor maintained within each net_pkt. All read/write operations affect this cursor. Note as well that read or write functions are strict on their length parameters: if it cannot r/w the given length it will fail. Length is not interpreted as an upper limit, it is instead the exact amount of data that must be read or written.

As there are two paths, TX and RX, there are two access modes: write and overwrite. This might sound a bit unusual, but is in fact simple and provides flexibility.

In write mode, whatever is written in the buffer affects the length of actual data present in the buffer. Buffer length should not be confused with the buffer size which is a limit any mode cannot

pass. In overwrite mode then, whatever is written must happen on valid data, and will not affect the buffer length. By default, a newly allocated net_pkt is on write mode, and its cursor points to the beginning of its buffer.

Let's see now, step by step, the functions and how they behave depending on the mode.

When freshly allocated with a buffer of 500 bytes, a net_pkt has 0 length, which means no valid data is in its buffer. One could verify this by:

```
len = net_pkt_get_len(pkt);
```

Now, let's write 8 bytes:

```
net_pkt_write(pkt, data, 8);
```

The buffer length is now 8 bytes. There are various helpers to write a byte, or big endian uint16_t, uint32_t.

```
net_pkt_write_u8(pkt, &foo);
net_pkt_write_be16(pkt, &ba);
net_pkt_write_be32(pkt, &bar);
```

Logically, net_pkt's length is now 15. But if we try to read at this point, it will fail because there is nothing to read at the cursor where we are at in the net_pkt. It is possible, while in write mode, to read what has been already written by resetting the cursor of the net_pkt. For instance:

```
net_pkt_cursor_init(pkt);
net_pkt_read(pkt, data, 15);
```

This will reset the cursor of the pkt to the beginning of the buffer and then let you read the actual 15 bytes present. The cursor is then again pointing at the end of the buffer.

To set a large area with the same byte, a memset function is provided:

```
net_pkt_memset(pkt, 0, 5);
```

Our net_pkt has now a length of 20 bytes.

Switching between modes can be achieved via `net_pkt_set_overwrite()` function. It is possible to switch mode back and forth at any time. The net_pkt will be set to overwrite and its cursor reset:

```
net_pkt_set_overwrite(pkt, true);
net_pkt_cursor_init(pkt);
```

Now the same operators can be used, but it will be limited to the existing data in the buffer, i.e. 20 bytes.

If it is necessary to know how much space is available in the net_pkt call:

```
net_pkt_available_buffer(pkt);
```

Or, if headers space needs to be accounted for, call:

```
net_pkt_available_payload_buffer(pkt, proto);
```

If you want to place the cursor at a known position use the function `net_pkt_skip()`. For example, to go after the IP header, use:

```
net_pkt_cursor_init(pkt);
net_pkt_skip(pkt, net_pkt_ip_header_len(pkt));
```

Data access Though the API shown previously is rather simple, it involves always copying things to and from the net_pkt buffer. In many occasions, it is more relevant to access the information stored in the buffer contiguously, especially with network packets which embed headers.

These headers are, most of the time, a known fixed set of bytes. It is then more natural to have a structure representing a certain type of header. In addition to this, if it is known the header size appears in a contiguous area of the buffer, it will be way more efficient to cast the actual position in the buffer to the type of header. Either for reading or writing the fields of such header, accessing it directly will save memory.

Net pkt comes with a dedicated API for this, built on top of the previously described API. It is able to handle both contiguous and non-contiguous access transparently.

There are two macros used to define a data access descriptor: `NET_PKT_DATA_ACCESS_DEFINE` when it is not possible to tell if the data will be in a contiguous area, and `NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE` when it is guaranteed the data is in a contiguous area.

Let's take the example of IP and UDP. Both IPv4 and IPv6 headers are always found at the beginning of the packet and are small enough to fit in a net_buf of 128 bytes (for instance, though 64 bytes could be chosen).

```
NET_PKT_DATA_ACCESS_CONTIGUOUS_DEFINE(ipv4_access, struct net_ipv4_hdr);
struct net_ipv4_hdr *ipv4_hdr;

ipv4_hdr = (struct net_ipv4_hdr *)net_pkt_get_data(pkt, &ipv4_access);
```

It would be the same for struct net_ipv4_hdr. For a UDP header it is likely not to be in a contiguous area in IPv6 for instance so:

```
NET_PKT_DATA_ACCESS_DEFINE(udp_access, struct net_udp_hdr);
struct net_udp_hdr *udp_hdr;

udp_hdr = (struct net_udp_hdr *)net_pkt_get_data(pkt, &udp_access);
```

At this point, the cursor of the net_pkt points at the beginning of the requested data. On the RX path, these headers will be read but not modified so to proceed further the cursor needs to advance past the data. There is a function dedicated for this:

```
net_pkt_acknowledge_data(pkt, &ipv4_access);
```

On the TX path, however, the header fields have been modified. In such a case:

```
net_pkt_set_data(pkt, &ipv4_access);
```

If the data are in a contiguous area, it will advance the cursor relevantly. If not, it will write the data and the cursor will be updated. Note that `net_pkt_set_data()` could be used in the RX path as well, but it is slightly faster to use `net_pkt_acknowledge_data()` as this one does not care about contiguity at all, it just advances the cursor via `net_pkt_skip()` directly.

API Reference

Network Packet Library

Related code samples

- Dumb HTTP serverImplement a simple, portable, HTTP server using BSD sockets.
- Dumb HTTP server (multi-threaded)Implement a simple HTTP server supporting simultaneous connections using BSD sockets.

Networking Technologies

Ethernet

- *Overview*
- *API Reference*

MAC Address Configuration Ethernet drivers can delegate most MAC address handling during initialization to `net_eth_mac_config` and `net_eth_mac_load()`. The structure is typically stored in the driver configuration and initialized with `NET_ETH_MAC_DT_CONFIG_INIT` or `NET_ETH_MAC_DT_INST_CONFIG_INIT`, which translate the devicetree properties into one of the following behaviors:

- `NET_ETH_MAC_STATIC` – use the full `local-mac-address` property.
- `NET_ETH_MAC_RANDOM` – generate a random locally administered MAC address, optionally using the bytes provided in `zephyr,mac-address-prefix` as the first octets.
- `NET_ETH_MAC_NVMEM` – read the remaining bytes from the `"mac-address"` *NVMEM* cell, again optionally prefixed by `zephyr,mac-address-prefix`.
- `NET_ETH_MAC_DEFAULT` – fall back to the driver's default logic (for example, a factory-programmed MAC address stored in peripheral registers).

Driver integration Embed the `net_eth_mac_config` structure inside the driver's configuration and a static buffer inside the driver's data:

```
struct my_eth_config {
    struct net_eth_mac_config mac_cfg;
    /* more config fields */
};

struct my_eth_data {
    uint8_t mac_addr[NET_ETH_ADDR_LEN];
    /* more data fields */
};

static const struct my_eth_config my_eth_config_0 = {
    .mac_cfg = NET_ETH_MAC_DT_INST_CONFIG_INIT(0),
};
static struct my_eth_data my_eth_data_0;
```

During initialization, call `net_eth_mac_load()` before registering the address with the network interface. The helper copies any statically provided bytes, fills the remaining octets, and performs the necessary validation. Drivers can still fall back to SoC-specific storage when no configuration was provided:

```
static int my_eth_init(const struct device *dev)
{
    const struct my_eth_config *cfg = dev->config;
    struct my_eth_data *data = dev->data;
    int ret;

    ret = net_eth_mac_load(&cfg->mac_cfg, data->mac_addr);
    if (ret == -ENODATA) {
```

(continues on next page)

(continued from previous page)

```
        ret = my_eth_hw_read_mac(dev, data->mac_addr);
    }

    return ret;
}

static void my_eth_iface_init(struct net_if *iface)
{
    const struct device *dev = net_if_get_device(iface);
    struct my_eth_data *data = dev->data;

    net_if_set_link_addr(iface, data->mac_addr, sizeof(data->mac_addr),␣
↪NET_LINK_ETHERNET);
}
```

Devicetree examples The examples below show how to pick a MAC address configuration for an ethernet controller node such as `&eth0`.

Static MAC address

```
&eth0 {
    local-mac-address = [00 11 22 33 44 55];
};
```

Random MAC address with prefix

```
&eth0 {
    zephyr,mac-address-prefix = [00 04 25];
    zephyr,random-mac-address;
};
```

NVMEM-provided MAC address with prefix MAC address can be retrieved from a non-volatile memory, typically an EEPROM, thanks to the *NVMEM API*.

```
&eth0 {
    zephyr,mac-address-prefix = [00 12 34];
    nvmem-cells = <&macaddr_cell>;
    nvmem-cell-names = "mac-address";
};

&eeprom0 {
    nvmem-layout {
        compatible = "fixed-layout";
        #address-cells = <1>;
        #size-cells = <1>;

        macaddr_cell: cell@0 {
            reg = <0x0 0x6>;
            #nvmem-cell-cells = <0>;
        };
    };
};
```

When no MAC-related properties are present, `net_eth_mac_load()` returns `-ENODATA` and the driver is expected to use its existing mechanism (for example, reading the hardware registers or using a build-time constant).

Changing the MAC address at runtime Applications that need to set MAC addresses dynamically (for example to adopt an address obtained from a management interface) need to enable the networking management API with `CONFIG_NET_MGMT` and use `net_mgmt` with `NET_REQUEST_ETHERNET_SET_MAC_ADDRESS`. The request internally calls the driver's `ethernet_api.set_config()` implementation.

```
static int app_set_mac_address(const struct device *dev)
{
    struct net_if *iface = net_if_lookup_by_dev(dev);
    struct ethernet_req_params params = {
        .mac_address = { { 0x02, 0x00, 0x5E, 0x01, 0x02, 0x03 } },
    };

    /* Make sure the iface is down */

    return net_mgmt(NET_REQUEST_ETHERNET_SET_MAC_ADDRESS, iface,
                    &params, sizeof(params));
}
```

Virtual LAN (VLAN) Support

- *Overview*
- *API Reference*

Overview Virtual LAN (VLAN) is a partitioned and isolated computer network at the data link layer (OSI layer 2). For ethernet network this refers to IEEE 802.1Q

In Zephyr, each individual VLAN is modeled as a virtual network interface. This means that there is an ethernet network interface that corresponds to a real physical ethernet port in the system. A virtual network interface is created for each VLAN, and this virtual network interface connects to the real network interface. This is similar to how Linux implements VLANs. The *eth0* is the real network interface and *vlan0* is a virtual network interface that is run on top of *eth0*.

VLAN support must be enabled at compile time by setting option `CONFIG_NET_VLAN` and `CONFIG_NET_VLAN_COUNT` to reflect how many network interfaces there will be in the system. For example, if there is one network interface without VLAN support, and two with VLAN support, the `CONFIG_NET_VLAN_COUNT` option should be set to 3.

Even if VLAN is enabled in a `prj.conf` file, the VLAN needs to be activated at runtime by the application. The VLAN API provides a `net_eth_vlan_enable()` function to do that. The application needs to give the network interface and desired VLAN tag as a parameter to that function. The VLAN tagging for a given network interface can be disabled by a `net_eth_vlan_disable()` function. The application needs to configure the VLAN network interface itself, such as setting the IP address, etc.

See also the VLAN sample application for API usage example. The source code for that sample application can be found at samples/net/ethernet/vlan.

The net-shell module contains *net vlan add* and *net vlan del* commands that can be used to enable or disable VLAN tags for a given network interface.

See the IEEE 802.1Q spec for more information about ethernet VLANs.

API Reference
Virtual LAN definitions and helpers

Related code samples

- Virtual LANSetup two virtual LAN networks and use net-shell to view the networks' settings.

Link Layer Discovery Protocol

- *Overview*
- *API Reference*

Overview The Link Layer Discovery Protocol (LLDP) is a vendor-neutral link layer protocol used by network devices for advertising their identity, capabilities, and neighbors on a wired Ethernet network.

For more information, see this LLDP Wikipedia article.

API Reference
Link Layer Discovery Protocol definitions and helpers

Related code samples

- Link Layer Discovery Protocol (LLDP)Enable LLDP support and setup VLANs.

IEEE 802.1Qav

Overview Credit-based shaping is an alternative scheduling algorithm used in network schedulers to achieve fairness when sharing a limited network resource. Zephyr has support for configuring a credit-based shaper described in the IEEE 802.1Qav-2009 standard. Zephyr does not implement the actual shaper; it only provides a way to configure the shaper implemented by the Ethernet device driver.

Enabling 802.1Qav To enable 802.1Qav shaper, the Ethernet device driver must declare that it supports credit-based shaping. The Ethernet driver's capability function must return `ETHERNET_QAV` value for this purpose. Typically also priority queues `ETHERNET_PRIORITY_QUEUES` need to be supported.

```
static enum ethernet_hw_caps eth_get_capabilities(const struct device *dev)
{
        ARG_UNUSED(dev);

        return ETHERNET_QAV | ETHERNET_PRIORITY_QUEUES |
               ETHERNET_HW_VLAN | ETHERNET_LINK_10BASE |
               ETHERNET_LINK_100BASE;
}
```

See `sam-e70-xplained` board Ethernet driver drivers/ethernet/eth_sam_gmac.c for an example.

Configuring 802.1Qav The application can configure the credit-based shaper like this:

```
#include <zephyr/net/net_if.h>
#include <zephyr/net/ethernet.h>
#include <zephyr/net/ethernet_mgmt.h>

static void qav_set_status(struct net_if *iface,
                           int queue_id, bool enable)
{
        struct ethernet_req_params params;
        int ret;

        memset(&params, 0, sizeof(params));

        params.qav_param.queue_id = queue_id;
        params.qav_param.enabled = enable;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_STATUS;

        /* Disable or enable Qav for a queue */
        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot %s Qav for queue %d for interface %p",
                        enable ? "enable" : "disable",
                        queue_id, iface);
        }
}

static void qav_set_bandwidth_and_slope(struct net_if *iface,
                                        int queue_id,
                                        unsigned int bandwidth,
                                        unsigned int idle_slope)
{
        struct ethernet_req_params params;
        int ret;

        memset(&params, 0, sizeof(params));

        params.qav_param.queue_id = queue_id;
        params.qav_param.delta_bandwidth = bandwidth;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_DELTA_BANDWIDTH;

        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
                LOG_ERR("Cannot set Qav delta bandwidth %u for "
                        "queue %d for interface %p",
                        bandwidth, queue_id, iface);
        }

        params.qav_param.idle_slope = idle_slope;
        params.qav_param.type = ETHERNET_QAV_PARAM_TYPE_IDLE_SLOPE;

        ret = net_mgmt(NET_REQUEST_ETHERNET_SET_QAV_PARAM,
                       iface, &params,
                       sizeof(struct ethernet_req_params));
        if (ret) {
```

(continues on next page)

(continued from previous page)

```
                LOG_ERR("Cannot set Qav idle slope %u for "
                        "queue %d for interface %p",
                        idle_slope, queue_id, iface);
        }
}
```

Overview Ethernet is a networking technology commonly used in local area networks (LAN). For more information, see this Ethernet Wikipedia article.

Zephyr supports following Ethernet features:

- 10, 100 and 1000 Mbit/sec links
- Auto negotiation
- Half/full duplex
- Promiscuous mode
- TX and RX checksum offloading
- MAC address filtering
- *MAC address configuration*
- *Virtual LANs*
- *Priority queues*
- *IEEE 802.1AS (gPTP)*
- *IEEE 802.1Qav (credit based shaping)*
- *LLDP (Link Layer Discovery Protocol)*

Not all Ethernet device drivers support all of these features. You can see what is supported by `net iface` net-shell command. It will print currently supported Ethernet features.

API Reference

Ethernet Support Functions

Related code samples

- bridgeTest and debug Ethernet bridge
- Inter-VM Shared Memory (ivshmem) EthernetCommunicate with another "cell" in the Jailhouse hypervisor using IVSHMEM Ethernet.
- Packet socketUse raw packet sockets over Ethernet.
- Quality of ServiceImplements a demo of quality of service on the ethernet layer.
- UDP sender using SO_TXTIMEControl the transmission time of a packet using SO_TXTIME socket option.

Ethernet MII Support Functions

IEEE 802.15.4

- *Introduction*
- *Supported Drivers*

- *API Reference*
  - *IEEE 802.15.4 API Overview*
  - *IEEE 802.15.4 Management API*
  - *IEEE 802.15.4 Driver API*
  - *IEEE 802.15.4 L2 / Native Stack API*
  - *OpenThread L2 Adaptation Layer API*

Introduction IEEE 802.15.4 is a technical standard which defines the operation of low-rate wireless personal area networks (LR-WPANs). For a more detailed overview of this standard, see the IEEE 802.15.4 Wikipedia article.

The most recent version of the standard is accessible through the IEEE GET Program. You need to create a free IEEE account and can then downloading it.

We're currently following the IEEE 802.15.4-2020 specification. This version is backwards compatible with IEEE 802.15.4-2015, parts of which are contained in the Thread protocol stack. The 2020 version also includes prior extensions that were accepted into the standard, namely IEEE 802.15.4g (SUN FSK) and IEEE 802.15.4e (TSCH) which are of relevance to industrial IoT and automation. For recent developments in UWB ranging technology, see IEEE 802.15.4z which is not yet integrated into the standard's mainline.

Whenever sections from the standard are cited in the documentation, they refer to IEEE 802.15.4-2020 section, table and figure numbering - unless otherwise specified.

Zephyr supports both, native IEEE 802.15.4 and Thread, with 6LoWPAN. Zephyr's *Thread protocol* implementation is based on OpenThread. The IPv6 header compression in 6LoWPAN is used for native IEEE 802.15.4.

Supported Drivers Zephyr includes a number of in-tree IEEE 802.15.4 radio drivers for various hardware platforms. The following table provides an overview of all available drivers, their vendors, supported frequency bands, bus interfaces, and the Kconfig option used to enable each driver.

Table 1: IEEE 802.15.4 In-Tree Radio Drivers

Driver	Vendor	Compatible	Frequency Band	Bus
CC1200	Texas Instruments	`ti,cc1200`	Sub-GHz	SPI
CC13xx/CC26xx	Texas Instruments	`ti,cc13xx-cc26xx-ieee802154`	2.4 GHz	On-chip
CC13xx/CC26xx Sub-GHz	Texas Instruments	`ti,cc13xx-cc26xx-ieee802154-s`	Sub-GHz	On-chip
CC2520	Texas Instruments	`ti,cc2520`	2.4 GHz	SPI
DW1000	Decawave (Qorvo)	`decawave,dw1000`	UWB (3.5–6.5 GHz)	SPI
ESP32	Espressif	`espressif,esp32-ieee802154`	2.4 GHz	On-chip
KW41Z	NXP	`nxp,kw41z-ieee802154`	2.4 GHz	On-chip
MCR20A	NXP	`nxp,mcr20a`	2.4 GHz	SPI
MCXW	NXP	`nxp,mcxw-ieee802154`	2.4 GHz	On-chip
nRF5	Nordic Semiconductor	`nordic,nrf-ieee802154`	2.4 GHz	On-chip
RF2XX (AT86RF2xx)	Atmel (Microchip)	`atmel,rf2xx`	2.4 GHz / Sub-GHz	SPI
STM32WBA	STMicroelectronics	`st,stm32wba-ieee802154`	2.4 GHz	On-chip
Telink B91	Telink Semiconductor	`telink,b91-zb`	2.4 GHz	On-chip
UART Pipe	Virtual (QEMU)	`zephyr,ieee802154-uart-pipe`	N/A (emulated)	UART

Note

The UART Pipe driver is a virtual driver used for testing IEEE 802.15.4 networking in QEMU environments. It is not intended for use with real radio hardware.

For detailed configuration options of each driver, refer to the corresponding Kconfig file in drivers/ieee802154/.

API Reference

IEEE 802.15.4 API Overview Gives an introduction and overview over the whole IEEE 802.15.4 subsystem and all of its APIs, configuration and user interfaces for all audiences.

IEEE 802.15.4 and Thread APIs

Related code samples

- 802.15.4 ”serial-radio”Implement a slip-radio device for Contiki-based border routers.

IEEE 802.15.4 Management API This is the main subsystem-specific API of interest to IEEE 802.15.4 **application developers** as it allows to configure the IEEE 802.15.4 subsystem at runtime. Other relevant interfaces for application developers are the typical shell, socket, Kconfig and devicetree APIs that can be accessed through Zephyr’s generic subsystem-independent documentation. Look out for IEEE802154/ieee802154 prefixes there.

IEEE 802.15.4 Net Management

IEEE 802.15.4 Driver API This is the main API of interest to IEEE 802.15.4 **driver developers**.

IEEE 802.15.4 Drivers

IEEE 802.15.4 L2 / Native Stack API This documents the IEEE 802.15.4 L2 native stack, which neither applications nor drivers will ever access directly. It is called internally by Zephyr's upper network layers (L3+), its socket and network context abstractions. This API is therefore of interest to IEEE 802.15.4 **subsystem contributors** only.

IEEE 802.15.4 L2

OpenThread L2 Adaptation Layer API Zephyr's OpenThread L2 platform adaptation layer glues the external OpenThread stack together with Zephyr's IEEE 802.15.4 protocol agnostic driver API. This API is of interest to OpenThread L2 **subsystem contributors** only.

The OpenThread API is part of the *Thread protocol* subsystem and documented there.

Thread protocol

- *Overview*
- *Internet connectivity*
- *Sample usage*
- *Thread related APIs*
  - *OpenThread Driver API*
  - *OpenThread L2 Adaptation Layer API*
  - *OpenThread Platform API*

Overview Thread is a low-power mesh networking technology, designed specifically for home automation applications. It is an IPv6-based standard, which uses 6LoWPAN technology over IEEE 802.15.4 protocol. IP connectivity lets you easily connect a Thread mesh network to the internet with a Thread Border Router.

The Thread specification provides a high level of network security. Mesh networks built with Thread are secure - only authenticated devices can join the network and all communications within the mesh are encrypted. More information about Thread protocol can be found at Thread Group website.

Zephyr integrates an open source Thread protocol implementation called OpenThread, documented on the OpenThread website.

Internet connectivity A Thread Border Router is required to connect mesh network to the internet. An open source implementation of Thread Border Router is provided by the OpenThread community. See OpenThread Border Router guide for instructions on how to set up a Border Router.

Sample usage You can try using OpenThread with the Zephyr Echo server and Echo client samples, which provide out-of-the-box configuration for OpenThread. To enable OpenThread support in these samples, build them with `overlay-ot.conf` overlay config file. See sockets-echo-server and sockets-echo-client samples for details.

Zephyr also provides an openthread-shell, which is useful for testing and debugging Thread and its underlying IEEE 802.15.4 drivers.

Thread related APIs

OpenThread Driver API OpenThread L2 uses Zephyr's protocol agnostic IEEE 802.15.4 driver API internally. This API is of interest to **driver developers** that want to support OpenThread.

The driver API is part of the *IEEE 802.15.4 Driver API* subsystem and documented there.

OpenThread L2 Adaptation Layer API Zephyr's OpenThread L2 platform adaptation layer glues the external OpenThread stack together with Zephyr's IEEE 802.15.4 protocol agnostic driver API. This API is of interest to OpenThread L2 **subsystem contributors** only.

OpenThread Platform API The OpenThread platform API is defined by the OpenThread stack and implemented in Zephyr as an OpenThread module. Applications can use this implementation directly, or access it through the OpenThread L2 adaptation layer.

Using the OpenThread L2 Adaptation Layer API To use the OpenThread platform API via the OpenThread L2 adaptation layer, enable both the `CONFIG_NET_L2_OPENTHREAD` and `CONFIG_NETWORKING` Kconfig options by setting them to `y`. The adaptation layer will use the OpenThread radio API implementation found in `modules/openthread/platform/radio.c`. In this setup, the OpenThread stack is initialized and managed by the adaptation layer.

Using the OpenThread Platform API Directly You can also use the OpenThread platform API directly, bypassing the OpenThread L2 adaptation layer. However, this approach requires you to provide your own implementation of the OpenThread radio API that is compatible with your specific radio driver.

To use the OpenThread platform API directly, set the `CONFIG_OPENTHREAD` Kconfig option to `y`, and do **not** set `CONFIG_NET_L2_OPENTHREAD`. In this case, you must implement the following functions from the OpenThread radio API using your own radio driver:

- `otPlatRadioGetPromiscuous`
- `otPlatRadioGetCcaEnergyDetectThreshold`
- `otPlatRadioGetTransmitPower`
- `otPlatRadioGetIeeeEui64`
- `otPlatRadioSetPromiscuous`
- `otPlatRadioGetCaps`
- `otPlatRadioGetTransmitBuffer`
- `otPlatRadioSetPanId`
- `otPlatRadioEnable`
- `otPlatRadioDisable`
- `otPlatRadioReceive`
- `otPlatRadioGetRssi`
- `otPlatRadioGetReceiveSensitivity`
- `otPlatRadioEnergyScan`
- `otPlatRadioSetExtendedAddress`
- `otPlatRadioSetShortAddress`

- `otPlatRadioAddSrcMatchExtEntry`
- `otPlatRadioTransmit`
- `otPlatRadioClearSrcMatchShortEntries`
- `otPlatRadioClearSrcMatchExtEntries`
- `otPlatRadioEnableSrcMatch`
- `otPlatRadioAddSrcMatchShortEntry`
- `otPlatRadioClearSrcMatchShortEntry`
- `otPlatRadioClearSrcMatchExtEntry`

Additionally, you must implement the following functions from the OpenThread radio API (see include/zephyr/net/openthread.h) to handle radio initialization and event processing:

- `platformRadioInit()`
- `platformRadioProcess()`

To initialize the OpenThread stack in this approach, either call the `ot_platform_init()` function in your application, or enable the `CONFIG_OPENTHREAD_SYS_INIT` Kconfig option to automatically initialize OpenThread during system startup. You can set the initialization priority using the `CONFIG_OPENTHREAD_SYS_INIT_PRIORITY` Kconfig option.

OpenThread stack

Related code samples

- OpenThread co-processorBuild a Thread border-router using OpenThread's co-processor designs.
- OpenThread CoAP client and server applicationBuild a Full Thread Device (FTD) CoAP server and client.

Point-to-Point Protocol (PPP) Support

- *Overview*
- *Testing*

Overview Point-to-Point Protocol (PPP) is a data link layer (layer 2) communications protocol used to establish a direct connection between two nodes. PPP is used over many types of serial links since IP packets cannot be transmitted over a modem line on their own, without some data link protocol.

In Zephyr, each individual PPP link is modelled as a network interface. This is similar to how Linux implements PPP.

PPP support must be enabled at compile time by setting option `CONFIG_NET_L2_PPP`. The PPP implementation supports only these protocols:

- LCP (Link Control Protocol, **RFC 1661**)
- HDLC (High-level data link control, **RFC 1662**)
- IPCP (IP Control Protocol, **RFC 1332**)
- IPV6CP (IPv6 Control Protocol, **RFC 5072**)

For using PPP with a cellular modem, see cellular-modem sample for additional information.

Testing See the net-tools README file for more details on how to test the Zephyr PPP against pppd running in Linux.

Wi-Fi Management

Overview The Wi-Fi management API is used to manage Wi-Fi networks. It supports below modes:

- IEEE802.11 Station (STA)
- IEEE802.11 Access Point (AP)
- IEEE802.11 P2P (Wi-Fi Direct)

Only personal mode security is supported with below types:

- Open
- OWE
- WEP
- WPA2-PSK
- WPA2-PSK-256
- WPA3-SAE

The Wi-Fi management API is implemented in the `wifi_mgmt` module as a part of the networking L2 stack. Currently, two types of Wi-Fi drivers are supported:

- Networking or socket offloaded drivers
- Native L2 Ethernet drivers

Compiled Features To support applications that only require a certain subset of Wi-Fi features, `CONFIG_WIFI_USAGE_MODE` can be used as a hint for drivers to limit the functionality that needs to be compiled in. The following usage hints are available:

- `CONFIG_WIFI_USAGE_MODE_STA` (Connecting to an access point)
- `CONFIG_WIFI_USAGE_MODE_AP` (Being an access point)
- `CONFIG_WIFI_USAGE_MODE_STA_AP` (Both being and connecting to an access point)
- `CONFIG_WIFI_USAGE_MODE_SCAN_ONLY` (Access point SSID scanning only)

> **Note**
>
> Support for a requested usage mode is hardware dependent.

Wi-Fi shell The Wi-Fi shell provides commands for testing and exploring the Wi-Fi management API through an interactive interface, without requiring a dedicated application. Enable the `CONFIG_NET_L2_WIFI_SHELL` option to add the `wifi` command.

The main Wi-Fi shell subcommands include:

Table 2: Wi-Fi shell subcommands

Subcommand	Description
`scan`	Scan for Wi-Fi networks.
`connect`	Connect to a Wi-Fi network.
`disconnect`	Disconnect from a Wi-Fi network.
`status`	Show the Wi-Fi interface status.

Use `wifi --help` to list all available subcommands and `wifi <subcommand> --help` for command-specific help. See wifi-shell for a sample application that enables the Wi-Fi shell.

Warning

By default, the Wi-Fi shell scan command does not limit scan results. Due to the UART shell backend being slow, printing all scan results may cause network management events to be dropped. To avoid the warning, adjust `CONFIG_NET_MGMT_EVENT_QUEUE_TIMEOUT` or `CONFIG_NET_MGMT_EVENT_QUEUE_SIZE` if needed.

Wi-Fi PSA crypto supported build To enable PSA crypto API supported Wi-Fi build, the `CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_ALT` and the `CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_MBEDTLS_PSA` need to be set.

Wi-Fi feature to crypto mapping For a mapping of Wi-Fi features (WPA3-SAE, DPP, SAE-PK, WPA2-PSK, Enterprise EAP, etc.) to crypto primitives (bignum, ECDH, TLS, hashes, AES) and which use **Legacy crypto** vs **PSA crypto**, see the dedicated sub-page:

Wi-Fi feature to crypto mapping This page maps Wi-Fi features supported in Zephyr (via the hostap-based wpa_supplicant) to the underlying MbedTLS crypto primitives. Use it to see which features need bignum, ECDH, TLS, etc., and which code paths use **Legacy crypto** (MbedTLS legacy APIs) vs **PSA crypto** (Platform Security Architecture APIs) when `CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_MBEDTLS_PSA` is enabled.

The implementation lives in the hostap module: `crypto_mbedtls_alt.c` (generic crypto) and `tls_mbedtls_alt.c` (TLS/EAP). Only the MbedTLS backend is considered here.

Feature set (from hostap Kconfig) Features are gated by Kconfig. Relevant options include:

- `CONFIG_WIFI_NM_WPA_SUPPLICANT_WEP` — WEP (legacy)
- `CONFIG_WIFI_NM_WPA_SUPPLICANT_WPA3_COMMON` — WPA3-SAE when Internal or External is selected (`WIFI_NM_WPA_SUPPLICANT_WPA3_IMPLEMENTATION`; default Internal). `CONFIG_WIFI_NM_WPA_SUPPLICANT_WPA3` is promptless and turns on the internal bignum SAE path when Internal is chosen (set the implementation choice in `prj.conf`, not this symbol).
- `CONFIG_WIFI_NM_WPA_SUPPLICANT_DPP` — Wi-Fi Easy Connect (DPP)
- `CONFIG_WIFI_NM_WPA_SUPPLICANT_WPS` — Wi-Fi Protected Setup
- `CONFIG_WIFI_NM_WPA_SUPPLICANT_P2P` — P2P / Wi-Fi Direct (implies WPS)
- `CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_ENTERPRISE` — EAP (EAP-TLS, EAP-TTLS-MSCHAPV2, EAP-PEAP-MSCHAPV2, EAP-PEAP-GTC, EAP-PEAP-TLS)

WPA2-PSK and WPA2-PSK-256 are available whenever crypto is not set to `CRYPTO_NONE`.

Feature crypto primitives (MbedTLS)

Table 3: Wi-Fi feature to crypto mapping

Feature	Crypto primitives	Legacy crypto	PSA crypto
WPA3-SAE	• Bignum (mpi), modulo, exponentiation • EC group • HMAC-SHA256 • AES (CCMP). SAE uses Dragonfly (PWE) with bignum + modulo.	• Bignum (mbedtls_mpi) • EC • HMAC, AES (unless PSA build)	• Hashes, HMAC, AES when PSA enabled • Bignum/EC still legacy
SAE-PK	Same as WPA3-SAE plus ECDH and EC key operations (certificate-based SAE).	ECDH/EC key ops (legacy MbedTLS or PSA ECDH depending on config).	• ECDH can use PSA • Bignum/SAE core remain legacy
DPP (Easy Connect)	• ECDH (P-256, P-384, P-521) • EC key gen/sign/verify • Hashes, AES. DPP2 adds PKCS#7; DPP3 adds HPKE.	• ECDH, EC, RSA (if used) • X.509/CSR in TLS/crypto layer	• Hashes, HMAC, AES via PSA • ECDH/EC may use PSA • TLS/CSR/PKCS#7/HPKE layer legacy
WPA2-PSK / WPA2-PSK-256	• PBKDF2-SHA1 (or SHA256 for -256) • HMAC • AES (CCMP) • OMAC1-AES (key wrap)	All via mbedtls (PBKDF2, HMAC, AES, CMAC) if PSA disabled.	PBKDF2, HMAC, AES, OMAC1 implemented via PSA when PSA enabled.
WEP	• RC4/ARC4 (stream cipher) • Optionally AES for some wrappers. Deprecated.	Legacy MbedTLS only (no PSA path for WEP in current code).	N/A (WEP not migrated to PSA).
WPS	• DH (finite-field) • Bignum • Hashes, HMAC, AES, TLS-PRF. Registrar uses TLS.	• DH (mbedtls_dhm), bignum (mbedtls_mpi) • TLS in tls_mbedtls_alt	• Hashes, HMAC, AES, PBKDF2 via PSA • DH/bignum and TLS legacy
EAP-TLS / EAP-TTLS / EAP-PEAP	• TLS 1.2 (and optionally 1.3) • RSA • X.509 parse/verify • Hashes, HMAC, AES (cipher suites)	• Full TLS stack (mbedtls_ssl_*, mbedtls_x509_*) • RSA • No PSA in tls_mbedtls_alt	• TLS layer remains legacy • Underlying hashes/HMAC/AES can use PSA in crypto_mbedtls_alt

> **Note**
>
> WEP must be explicitly enabled with `CONFIG_WIFI_NM_WPA_SUPPLICANT_WEP`. It is deprecated and insecure; use only for legacy networks.

Summary: Legacy vs PSA (MbedTLS backend) When `CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_MBEDTLS_PSA` is enabled, the implementation in `crypto_mbedtls_alt.c` (and `supp_psa_api.h` / `supp_psa_api.c`) splits as below. Use this table to see which operations use **PSA** vs **Legacy** MbedTLS.

Table 4: Legacy vs PSA by crypto operation

Operation	API	Used by / notes
Message digests (MD5, SHA1, SHA256, SHA384, SHA512)	PSA	
HMAC (all above hash types)	PSA	
PBKDF2-SHA1	PSA	WPA2-PSK key derivation
AES (block, CBC, CTR, OMAC1-AES)	PSA	Key wrap, CCMP, etc.
Bignum (crypto_bignum_*)	Legacy	SAE, EAP-PWD, EAP-EKE, EAP-IKEV2, WPS; no PSA bignum in hostap
ECDH / EC key operations	Legacy	DPP, SAE-PK, EAP-PWD (EC). May be PSA-backed when `MBEDTLS_ECDH_C` / `CONFIG_PSA_WANT_ALG_ECDH`; wrapper layer common
TLS/SSL	Legacy	EAP-TLS, EAP-TTLS, EAP-PEAP; full stack in `tls_mbedtls_alt.c`
RSA	Legacy	TLS, X.509
X.509 / CSR	Legacy	Parse and generation
WEP	Legacy	No PSA path

So: **WPA2-PSK and WPA2-PSK-256** use only PSA for their crypto; **WPA3-SAE**, **DPP**, **SAE-PK**, **WPS**, and **Enterprise EAP** still rely on legacy bignum, EC, or TLS. See the feature table above for per-feature impact.

Wi-Fi Enterprise test: X.509 Certificate management Wi-Fi enterprise security requires use of X.509 certificates, two methods of installing certificates are supported:

Compile time certificates Test certificates in PEM format are committed to the repo at samples/net/wifi/test_certs and the during the build process the certificates are converted to a C header file that is included by the Wi-Fi shell module.

If you want to use your own certificates, you can replace the existing certificates with your own certificates in the same directory.

```
$ export WIFI_TEST_CERTS_DIR=samples/net/wifi/test_certs/rsa3k
$ cp client.pem $WIFI_TEST_CERTS_DIR
$ cp client-key.pem $WIFI_TEST_CERTS_DIR
$ cp ca.pem $WIFI_TEST_CERTS_DIR
$ cp client2.pem $WIFI_TEST_CERTS_DIR
$ cp client-key2.pem $WIFI_TEST_CERTS_DIR
$ cp ca2.pem $WIFI_TEST_CERTS_DIR
$ west build -p -b <board> samples/net/wifi -S wifi-enterprise
```

For RSA 2048-bit certificates, use `rsa2k_no_des`.

```
$ export WIFI_TEST_CERTS_DIR=samples/net/wifi/test_certs/rsa2k_no_des
```

or you can set the `WIFI_TEST_CERTS_DIR` environment variable to point to the directory containing your certificates.

```
$ west build -p -b <board> samples/net/wifi -S wifi-enterprise -- -
↪DWIFI_TEST_CERTS_DIR=<path_to_your_certificates>
```

Run time certificates The Wi-Fi shell module uses TLS credentials subsystem to store and manage the certificates. The certificates can be added at runtime using the shell commands, see *TLS Credentials Shell* for more details. The sample or application need to enable the `CONFIG_WIFI_SHELL_RUNTIME_CERTIFICATES` option to use this feature.

To facilitate installation of the certificates, a helper script is provided, see below for usage.

```
$ ./scripts/utils/wifi_ent_cert_installer.py -p␣
↪samples/net/wifi/test_certs/rsa2k_no_des
```

The script will install the certificates to the TLS credentials store in the device over UART and using TLS credentials shell commands.

To initiate a Wi-Fi connection using enterprise security, use one of the following commands depending on the EAP method:

- EAP-TLS

```
uart:~$ wifi connect -s <SSID> -c <channel> -k 7 -w 2 -a <Anonymous identity>␣
↪--key1-pwd <Password EAP phase1> --key2-pwd <Password EAP phase2>
```

- EAP-TTLS-MSCHAPV2

```
uart:~$ wifi connect -s <SSID> -c <channel> -k 14 -K <Private key Password> --
↪eap-id1 <Client Identity> --eap-pwd1 <Client Password> -a <Anonymous␣
↪identity>
```

- EAP-PEAP-MSCHAPV2

```
uart:~$ wifi connect -s <SSID> -c <channel> -k 12 -K <Private key Password> --
↪eap-id1 <Client Identity> --eap-pwd1 <Client Password> -a <Anonymous␣
↪identity>
```

Server certificate is also provided in the same directory for testing purposes. Any AAA server can be used for testing purposes, for example, `FreeRADIUS` or `hostapd`.

Server certificate domain name verification The authentication server's identity is verified by validating the domain name in the X.509 certificate received from the server, using the `Common Name` (CN) field.

- Exact domain match — Verifies that the certificate's CN exactly matches the specified domain.
- Domain suffix match — Allows a certificate whose CN ends with the specified domain suffix.

To initiate a Wi-Fi connection using enterprise security with server certificate validation, use one of the following commands, depending on the desired validation mode:

- Exact domain match

```
wifi connect -s <SSID> -c <channel> -k 12 -K <Private key Password> -e
↪<Domain match>
```

- Domain suffix match

```
wifi connect -s <SSID> -c <channel> -k 12 -K <Private key Password> -x
↪<Domain suffix name>
```

Certificate requirements for EAP methods Different EAP methods have varying client-side certificate requirements, as outlined below:

- **EAP-TLS - Requires both a client certificate (and its private key) and a CA certificate on the client.**
  The client authenticates itself to the server using its certificate.
- **EAP-TTLS-MSCHAPV2 - Requires only the CA certificate on the client.**
  The client authenticates to the server using a username and password <MSCHAPV2> inside the TLS tunnel. No client certificate is needed.
- **EAP-PEAP-MSCHAPV2 - Requires only the CA certificate on the client.**
  Like TTLS, the client uses a username and password <MSCHAPV2> inside the TLS tunnel and does not require a client certificate.

> **Note**
>
> The certificates are for testing purposes only and should not be used in production. They are generated using FreeRADIUS raddb scripts.

> **Note**
>
> When using TLS credentials subsystem, by default the volatile backend i.e., `CONFIG_TLS_CREDENTIALS_BACKEND_VOLATILE` is chosen. When using the volatile backend, the certificates are stored in RAM and are lost on reboot, so the certificates need to be installed again after reboot. As an alternative, the PS (protected storage) backend i.e., `CONFIG_TLS_CREDENTIALS_BACKEND_PROTECTED_STORAGE` can be used to store the certificates in the non-volatile storage.

How to Generate Test Certificates Using FreeRADIUS The test certificates in `samples/net/wifi/test_certs/rsa2k_no_des` are generated using the FreeRADIUS raddb/certs scripts. You can generate your own certificates for testing as follows:

1. **Prerequisites** - Install OpenSSL and GNU Make. - Download the FreeRADIUS raddb/certs directory.
2. **Edit the Makefile** In the `raddb/certs` directory, edit the `Makefile` to add `-nodes` to the OpenSSL commands for server and client keys. This ensures the private keys are not password-protected (Zephyr Wi-Fi shell does not support private key passwords):

```
$(OPENSSL) req -new -out server.csr -keyout server.key -nodes -config ./
↪server.cnf
$(OPENSSL) req -new -out client.csr -keyout client.key -nodes -config ./
↪client.cnf
```

3. **(Optional) Edit the .cnf files** Customize `server.cnf` and `client.cnf` as needed for your environment.
4. **Generate Certificates** Run the following commands in the `raddb/certs` directory:

```
make destroycerts
make server
make client
```

5. **Rename Files for Zephyr** Match the filenames used in Zephyr samples:

FreeRADIUS Output	Zephyr Sample Name
ca.pem server.key server.pem client.key client.pem	ca.pem server-key.pem server.pem client-key.pem client.pem

6. **Copy the files** Place the renamed files in your Zephyr project's certificate directory (e.g., `samples/net/wifi/test_certs/rsa2k_no_des`). Encrypt private keys with AES (PBES2), not DES; see the existing `rsa2k_no_des` keys for reference.

> **Note**
>
> These certificates are for testing only and should not be used in production.

Wi-Fi P2P (Wi-Fi Direct) Wi-Fi P2P or Wi-Fi Direct enables devices to communicate directly with each other without requiring a traditional access point. This feature is particularly useful for device-to-device communication scenarios.

To enable and build with Wi-Fi P2P support:

```
$ west build -p -b <board> samples/net/wifi/shell -- -
↪DCONFIG_WIFI_NM_WPA_SUPPLICANT_P2P=y
```

Wi-Fi NAN (Neighbor Awareness Networking) Wi-Fi NAN (Neighbor Awareness Networking), also known as Wi-Fi Aware, enables devices to discover services and communicate with nearby devices without requiring a traditional access point or Internet connection. This feature uses a publish-subscribe model where devices can publish services or subscribe to discover services from nearby peers.

To enable and build with Wi-Fi NAN support:

```
$ west build -p -b <board> samples/net/wifi/shell -- -
↪DCONFIG_WIFI_NM_WPA_SUPPLICANT_NAN=y
```

API Reference

Wi-Fi Management

> **Related code samples**
>
> - Wi-Fi AP-STA modeConfigure a Wi-Fi board to operate as both an Access Point (AP) and a Station (STA).
> - Wi-Fi provisioning over BLEReceive Wi-Fi credentials over a BLE GATT service and connect to the access point.

Wi-Fi credentials Library

- *Overview*
- *Configuration*
- *Adding credentials*

- *Querying credentials*
- *Removing credentials*
- *Shell commands*
- *Limitations*
- *API documentation*

The Wi-Fi credentials library provides means to load and store Wi-Fi® network credentials.

Overview This library uses either Zephyr's settings subsystem or Platform Security Architecture (PSA) Internal Trusted Storage (ITS) to store credentials. It also holds a list of SSIDs in RAM to provide dictionary-like access using SSIDs as keys.

Configuration To use the Wi-Fi credentials library, enable the `CONFIG_WIFI_CREDENTIALS` Kconfig option.

You can pick the backend using the following options:

- `CONFIG_WIFI_CREDENTIALS_BACKEND_PSA` - Default option for non-secure targets, which includes a TF-M partition (non-minimal TF-M profile type).
- `CONFIG_WIFI_CREDENTIALS_BACKEND_SETTINGS` - Default option for secure targets.

To configure the maximum number of networks, use the `CONFIG_WIFI_CREDENTIALS_MAX_ENTRIES` Kconfig option.

The IEEE 802.11 standard does not specify the maximum length of SAE passwords. To change the default, use the `CONFIG_WIFI_CREDENTIALS_SAE_PASSWORD_LENGTH` Kconfig option.

Adding credentials You can add credentials using the `wifi_credentials_set_personal()` and `wifi_credentials_set_personal_struct()` functions. The former will build the internally used struct from given fields, while the latter takes the struct directly. If you add credentials with the same SSID twice, the older entry will be overwritten.

Querying credentials With an SSID, you can query credentials using the `wifi_credentials_get_by_ssid_personal()` and `wifi_credentials_get_by_ssid_personal_struct()` functions.

You can iterate over all stored credentials with the `wifi_credentials_for_each_ssid()` function. Deleting or overwriting credentials while iterating is allowed, since these operations do not change internal indices.

Removing credentials You can remove credentials using the `wifi_credentials_delete_by_ssid()` function.

Shell commands `wifi cred` is an extension to the Wi-Fi command line. It adds the following subcommands to interact with the Wi-Fi credentials library:

Table 5: Wi-Fi credentials shell subcommands

Subcommands	Description
add	Add a network to the credentials storage with following parameters: <-s –ssid ”<SSID>”>: SSID. [-c –channel]: Channel that needs to be scanned for connection. 0:any channel [-b, –band] 0: any band (2:2.4GHz, 5:5GHz, 6:6GHz) [-p, –passphrase]: Passphrase (valid only for secure SSIDs) [-k, –key-mgmt]: Key management type. 0:None, 1:WPA2-PSK, 2:WPA2-PSK-256, 3:SAE-HNP, 4:SAE-H2E, 5:SAE-AUTO, 6:WAPI,” “ 7:EAP-TLS, 8:WEP, 9: WPA-PSK, 10: WPA-Auto-Personal, 11: DPP [-w, –ieee-80211w]: MFP (optional: needs security type to be specified) : 0:Disable, 1:Optional, 2:Required. [-m, –bssid]: MAC address of the AP (BSSID). [-t, –timeout]: Duration after which connection attempt needs to fail. [-a, –identity]: Identity for enterprise mode. [-K, –key-passwd]: Private key passwd for enterprise mode. [-h, –help]: Print out the help for the connect command.
delete <SSID>	Removes network from credentials storage.
list	Lists networks in credential storage.
auto_connect	Automatically connects to any stored network.

Limitations The library has the following limitations:

- Although permitted by the IEEE 802.11 standard, this library does not support zero-length SSIDs.
- Wi-Fi Protected Access (WPA) Enterprise credentials are only partially supported.
- The number of networks stored is fixed compile time.

API documentation The following section provides an overview and reference for the Wi-Fi credentials API available in Zephyr:

Wi-Fi credentials library

Related code samples

- Wi-Fi provisioning over BLEReceive Wi-Fi credentials over a BLE GATT service and connect to the access point.

Protocols

CoAP

- *Overview*
- *Sample Usage*

  - *CoAP Server*
  - *CoAP Client*
- *Testing*
  - *libcoap*
  - *TTCN3*
- *API Reference*

Overview The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks. It provides a convenient API for RESTful Web services that support CoAP's features. For more information about the protocol itself, see **RFC 7252**.

Zephyr provides a CoAP library which supports client and server roles. The library can be enabled with `CONFIG_COAP` Kconfig option and is configurable as per user needs. The Zephyr CoAP library is implemented using plain buffers. Users of the API create sockets for communication and pass the buffer to the library for parsing and other purposes. The library itself doesn't create any sockets for users.

On top of CoAP, Zephyr has support for LwM2M "Lightweight Machine 2 Machine" protocol, a simple, low-cost remote management and service enablement mechanism. See *Lightweight M2M (LwM2M)* for more information.

Supported RFCs:

- **RFC 7252** - The Constrained Application Protocol (CoAP)
- **RFC 6690** - Constrained RESTful Environments (CoRE) Link Format
- **RFC 7959** - Block-Wise Transfers in the Constrained Application Protocol (CoAP)
- **RFC 7641** - Observing Resources in the Constrained Application Protocol (CoAP)
- **RFC 8613** - Object Security for Constrained RESTful Environments (OSCORE)

> **Note**
>
> Not all parts of these RFCs are supported. Features are supported based on Zephyr requirements.

The Zephyr CoAP library also supports Object Security for Constrained RESTful Environments (OSCORE) as specified in **RFC 8613**. See *OSCORE Support (RFC 8613)* for more information.

Sample Usage

> **Note**
>
> A *CoAP server* subsystem is available, the following is for creating a custom server implementation.

CoAP Server To create a CoAP server, resources for the server need to be defined. The `.well-known/core` resource should be added before all other resources that should be included in the responses of the `.well-known/core` resource.

```
static struct coap_resource resources[] = {
    { .get = well_known_core_get,
      .path = COAP_WELL_KNOWN_CORE_PATH,
    },
    { .get  = sample_get,
      .post = sample_post,
      .del  = sample_del,
      .put  = sample_put,
      .path = sample_path
    },
    { },
};
```

An application reads data from the socket and passes the buffer to the CoAP library to parse the message. If the CoAP message is proper, the library uses the buffer along with resources defined above to call the correct callback function to handle the CoAP request from the client. It's the callback function's responsibility to either reply or act according to CoAP request.

```
coap_packet_parse(&request, data, data_len, options, opt_num);
...
coap_handle_request(&request, resources, options, opt_num,
                    client_addr, client_addr_len);
```

If `CONFIG_COAP_URI_WILDCARD` enabled, server may accept multiple resources using MQTT-like wildcard style:

- the plus symbol represents a single-level wild card in the path;
- the hash symbol represents the multi-level wild card in the path.

```
static const char * const led_set[] = { "led","+","set", NULL };
static const char * const btn_get[] = { "button","#", NULL };
static const char * const no_wc[] = { "test","+1", NULL };
```

It accepts /led/0/set, led/1234/set, led/any/set, /button/door/1, /test/+1, but returns -ENOENT for /led/1, /test/21, /test/1.

This option is enabled by default, disable it to avoid unexpected behaviour with resource path like '/some_resource/+/#'.

> **Note**
>
> A *CoAP client* subsystem is available, the following is for creating a custom client implementation.

CoAP Client If the CoAP client knows about resources in the CoAP server, the client can start prepare CoAP requests and wait for responses. If the client doesn't know about resources in the CoAP server, it can request resources through the `.well-known/core` CoAP message.

```
/* Initialize the CoAP message */
char *path = "test";
struct coap_packet request;
uint8_t data[100];
uint8_t payload[20];

coap_packet_init(&request, data, sizeof(data),
                 1, COAP_TYPE_CON, 8, coap_next_token(),
```

(continues on next page)

(continued from previous page)

```
                 COAP_METHOD_GET, coap_next_id());

/* Append options */
coap_packet_append_option(&request, COAP_OPTION_URI_PATH,
                          path, strlen(path));

/* Append Payload marker if you are going to add payload */
coap_packet_append_payload_marker(&request);

/* Append payload */
coap_packet_append_payload(&request, (uint8_t *)payload,
                           sizeof(payload) - 1);

/* send over sockets */
```

Testing There are various ways to test Zephyr CoAP library.

libcoap libcoap implements a lightweight application-protocol for devices that are resource constrained, such as by computing power, RF range, memory, bandwidth, or network packet sizes. Sources can be found here libcoap. libcoap has a script (`examples/etsi_coaptest.sh`) to test coap-server functionality in Zephyr.

See the net-tools project for more details

The coap-server sample can be built and executed on QEMU as described in *Networking with QEMU*.

Use this command on the host to run the libcoap implementation of the ETSI test cases:

```
sudo ./libcoap/examples/etsi_coaptest.sh -i tap0 2001:db8::1
```

TTCN3 Eclipse has TTCN3 based tests to run against CoAP implementations.

Install eclipse-titan and set symbolic links for titan tools

```
sudo apt-get install eclipse-titan

cd /usr/share/titan

sudo ln -s /usr/bin bin
sudo ln /usr/bin/titanver bin
sudo ln -s /usr/bin/mctr_cli bin
sudo ln -s /usr/include/titan include
sudo ln -s /usr/lib/titan lib

export TTCN3_DIR=/usr/share/titan

git clone https://gitlab.eclipse.org/eclipse/titan/titan.misc.git

cd titan.misc
```

Follow the instruction to setup CoAP test suite from here:

- https://gitlab.eclipse.org/eclipse/titan/titan.misc
- https://gitlab.eclipse.org/eclipse/titan/titan.misc/-/tree/master/CoAP_Conf

After the build is complete, the coap-server sample can be built and executed on QEMU as described in *Networking with QEMU*.

Change the client (test suite) and server (Zephyr coap-server sample) addresses in coap.cfg file as per your setup.

Execute the test cases with following command.

```
ttcn3_start coaptests coap.cfg
```

Sample output of ttcn3 tests looks like this.

```
Verdict statistics: 0 none (0.00 %), 10 pass (100.00 %), 0 inconc (0.00 %), 0 fail␣
↪(0.00 %), 0 error (0.00 %).
Test execution summary: 10 test cases were executed. Overall verdict: pass
```

API Reference

COAP Library

Related code samples

- CoAP clientUse the CoAP library to implement a client that fetches a resource.
- CoAP downloadUse the CoAP client API to download data via a GET request
- CoAP serviceUse the CoAP server subsystem to register CoAP resources.
- CoAP TCP ClientUse the CoAP over TCP client to communicate with a CoAP server.
- CoAP uploadUse the CoAP client API to upload data via a PUT request

CoAP client

- *Overview*
  - *CoAP over TCP*
- *Sample Usage*
- *API Reference*

Overview The CoAP client library allows application to send CoAP requests and parse CoAP responses. The library can be enabled with `CONFIG_COAP_CLIENT` Kconfig option. The application is notified about the response via a callback that is provided to the API in the request. The CoAP client handles the communication over sockets. As the CoAP client doesn't create socket it is using, the application is responsible for creating the socket. Plain UDP or DTLS sockets are supported.

CoAP over TCP CoAP over reliable transports (TCP/TLS) is also supported as specified in **RFC 8323**. Enable it with `CONFIG_COAP_CLIENT_TCP`. The TCP client manages connection setup, CSM (Capabilities and Settings Message) exchange, and signaling (Ping/Pong, Release, Abort) internally. Unlike the UDP client, the application does not create the socket — instead it calls `coap_client_tcp_connect()` with the server address. See coap-client-tcp for a usage example.

Sample Usage The following is an example of a CoAP client initialization and request sending:

```
static struct coap_client client;
struct coap_client_request req = { 0 };
```

(continues on next page)

(continued from previous page)

```
coap_client_init(&client, NULL);

req.method = COAP_METHOD_GET;
req.confirmable = true;
strcpy(req.path, "test");
req.fmt = COAP_CONTENT_FORMAT_TEXT_PLAIN;
req.cb = response_cb;
req.payload = NULL;
req.len = 0;

/* Sock is a file descriptor referencing a socket, address is the net_sockaddr␣
↪struct for the
 * destination address of the request or NULL if the socket is already connected.
 */
ret = coap_client_req(&client, sock, &address, &req, -1);
```

Before any requests can be sent, the CoAP client needs to be initialized. After initialization, the application can send a CoAP request and wait for the response. Currently only one request can be sent for a single CoAP client at a time. There can be multiple CoAP clients.

The callback provided in the callback will be called in following cases:

- There is a response for the request
- The request failed for some reason

The callback contains a flag `last_block`, which indicates if there is more data to come in the response and means that the current response is part of a blockwise transfer. When the `last_block` is set to true, the response is finished and the client is ready for the next request after returning from the callback.

If the server responds to the request, the library provides the response to the application through the response callback registered in the request structure. As the response can be a blockwise transfer and the client calls the callback once per each block, the application should be to process all of the blocks to be able to process the response.

During a blockwise transfer the client compares the ETag option of the received blocks, as required by **RFC 7959**. When the resource representation changes in the middle of the transfer, the transfer is aborted and the callback is invoked with `result_code` set to `-EBADMSG`. Stricter than the RFC minimum, which only mandates comparing ETags the server provides, the transfer is also aborted when the ETag option appears or disappears between blocks, as such a mix of tagged and untagged blocks cannot be verified. The application should discard the partial data it received and may retry the request.

The following is an example of a very simple response handling function:

```
void response_cb(const struct coap_client_response_data *data, void *user_data)
{
    if (data->result_code >= 0) {
            LOG_INF("CoAP response from server %d", data->result_code);
            if (data->last_block) {
                    LOG_INF("Last packet received");
            }
    } else {
            LOG_ERR("Error in sending request %d", data->result_code);
    }
}
```

CoAP options may also be added to the request by the application. The following is an example of the application adding a Block2 option to the initial request, to suggest a maximum block size

to the server for a resource that it expects to be large enough to require a blockwise transfer (see RFC 7959 Figure 3: Block-Wise GET with Early Negotiation).

```
static struct coap_client client;
struct coap_client_request req = { 0 };

coap_client_init(&client, NULL);

req.method = COAP_METHOD_GET;
req.confirmable = true;
strcpy(req.path, "test");
req.fmt = COAP_CONTENT_FORMAT_TEXT_PLAIN;
req.cb = response_cb;
req.options[0] = coap_client_option_initial_block2();
req.num_options = 1;
req.payload = NULL;
req.len = 0;

ret = coap_client_req(&client, sock, &address, &req, -1);
```

Optionally, the application can register a payload callback instead of providing a payload pointer for the CoAP upload. In such cases, the CoAP client library will call this callback when preparing a PUT/POST request, so that the application can provide the payload in blocks, instead of having to provide a single contiguous buffer with the entire payload. An example callback, providing the content of the Lorem Ipsum string can look like this:

```
static int lorem_ipsum_cb(size_t offset, const uint8_t **payload, size_t *len,
                          bool *last_block, void *user_data)
{
    size_t data_left;

    if (offset > LOREM_IPSUM_STRLEN) {
        return -EINVAL;
    }

    *payload = LOREM_IPSUM + offset;

    data_left = LOREM_IPSUM_STRLEN - offset;
    if (data_left <= *len) {
        *len = data_left;
        *last_block = true;
    } else {
        *last_block = false;
    }

    return 0;
}
```

The callback can then be registered for the PUT/POST request instead of a payload pointer:

```
struct coap_client_request req = { 0 };

req.method = COAP_METHOD_PUT;
req.confirmable = true;
strcpy(req.path, "lorem-ipsum");
req.fmt = COAP_CONTENT_FORMAT_TEXT_PLAIN;
req.cb = response_cb;
req.payload_cb = lorem_ipsum_cb,
```

(continues on next page)

(continued from previous page)

```
ret = coap_client_req(&client, sock, &address, &req, -1);
```

API Reference

CoAP client API

CoAP TCP client API

Related code samples

- CoAP TCP ClientUse the CoAP over TCP client to communicate with a CoAP server.

CoAP server

- *Overview*
- *Setup*
- *Sample Usage*
- *Observable resources*
- *CoAP Events*
- *CoRE Link Format*
- *API Reference*

Overview Zephyr comes with a batteries-included CoAP server, which uses services to listen for CoAP requests. The CoAP services handle communication over sockets and pass requests to registered CoAP resources.

Setup Some configuration is required to make sure services can be started using the CoAP server. The `CONFIG_COAP_SERVER` option should be enabled in your project:

Listing 1: `prj.conf`

```
CONFIG_COAP_SERVER=y
```

All services are added to a predefined linker section and all resources for each service also get their respective linker sections. If you would have a service `my_service` it has to be prefixed with `coap_resource_` and added to a linker file:

Listing 2: `sections-ram.ld`

```
#include <zephyr/linker/iterable_sections.h>

ITERABLE_SECTION_RAM(coap_resource_my_service, Z_LINK_ITERABLE_SUBALIGN)
```

Add this linker file to your application using CMake:

Listing 3: CMakeLists.txt

```
# Support LD linker template
zephyr_linker_sources(DATA_SECTIONS sections-ram.ld)

# Support CMake linker generator
zephyr_iterable_section(NAME coap_resource_my_service
                        GROUP DATA_REGION ${XIP_ALIGN_WITH_INPUT})
```

You can now define your service as part of the application:

```
#include <zephyr/net/coap_service.h>

static const uint16_t my_service_port = 5683;

COAP_SERVICE_DEFINE(my_service, "0.0.0.0", &my_service_port,␣
↪COAP_SERVICE_AUTOSTART);
```

Note

Services defined with the COAP_SERVICE_AUTOSTART flag will be started together with the CoAP server thread. Services can be manually started and stopped with coap_service_start and coap_service_stop respectively.

Sample Usage The following is an example of a CoAP resource registered with our service:

```
#include <zephyr/net/coap_service.h>

static int my_get(struct coap_resource *resource, struct coap_packet *request,
                  struct net_sockaddr *addr, socklen_t addr_len)
{
    static const char *msg = "Hello, world!";
    uint8_t data[CONFIG_COAP_SERVER_MESSAGE_SIZE];
    struct coap_packet response;
    uint16_t id;
    uint8_t token[COAP_TOKEN_MAX_LEN];
    uint8_t tkl, type;

    type = coap_header_get_type(request);
    id = coap_header_get_id(request);
    tkl = coap_header_get_token(request, token);

    /* Determine response type */
    type = (type == COAP_TYPE_CON) ? COAP_TYPE_ACK : COAP_TYPE_NON_CON;

    coap_packet_init(&response, data, sizeof(data), COAP_VERSION_1, type, tkl,␣
↪token,
                     COAP_RESPONSE_CODE_CONTENT, id);

    /* Set content format */
    coap_append_option_int(&response, COAP_OPTION_CONTENT_FORMAT,
                           COAP_CONTENT_FORMAT_TEXT_PLAIN);

    /* Append payload */
    coap_packet_append_payload_marker(&response);
    coap_packet_append_payload(&response, (uint8_t *)msg, strlen(msg));
```

(continues on next page)

(continued from previous page)

```
    /* Send to response back to the client */
    return coap_resource_send(resource, &response, addr, addr_len, NULL);
}

static int my_put(struct coap_resource *resource, struct coap_packet *request,
                  struct net_sockaddr *addr, socklen_t addr_len)
{
    /* ... Handle the incoming request ... */

    /* Return a CoAP response code as a shortcut for an empty ACK message */
    return COAP_RESPONSE_CODE_CHANGED;
}

static const char * const my_resource_path[] = { "test", NULL };
COAP_RESOURCE_DEFINE(my_resource, my_service, {
    .path = my_resource_path,
    .get = my_get,
    .put = my_put,
});
```

Note

As demonstrated in the example above, a CoAP resource handler can return response codes to let the server respond with an empty ACK response.

Observable resources The CoAP server provides logic for parsing observe requests and stores these using the runtime data of CoAP services. An example using a temperature sensor can look like:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/sensor.h>
#include <zephyr/net/coap_service.h>

static void notify_observers(struct k_work *work);
K_WORK_DELAYABLE_DEFINE(temp_work, notify_observers);

static int send_temperature(struct coap_resource *resource,
                            const struct net_sockaddr *addr, socklen_t addr_len,
                            uint16_t age, uint16_t id, const uint8_t *token,␣
↪uint8_t tkl,
                            bool is_response)
{
    const struct device *dev = DEVICE_DT_GET(DT_ALIAS(ambient_temp0));
    uint8_t data[CONFIG_COAP_SERVER_MESSAGE_SIZE];
    struct coap_packet response;
    char payload[14];
    struct sensor_value value;
    double temp;
    uint8_t type;

    /* Determine response type */
    type = is_response ? COAP_TYPE_ACK : COAP_TYPE_CON;
```

(continues on next page)

(continued from previous page)

```
    if (!is_response) {
        id = coap_next_id();
    }

    coap_packet_init(&response, data, sizeof(data), COAP_VERSION_1, type, tkl,␣
↪token,
                     COAP_RESPONSE_CODE_CONTENT, id);

    if (age >= 2U) {
        coap_append_option_int(&response, COAP_OPTION_OBSERVE, age);
    }

    /* Set content format */
    coap_append_option_int(&response, COAP_OPTION_CONTENT_FORMAT,
                           COAP_CONTENT_FORMAT_TEXT_PLAIN);

    /* Get the sensor data */
    sensor_sample_fetch_chan(dev, SENSOR_CHAN_AMBIENT_TEMP);
    sensor_channel_get(dev, SENSOR_CHAN_AMBIENT_TEMP, &value);
    temp = sensor_value_to_double(&value);

    snprintk(payload, sizeof(payload), "%0.2f°C", temp);

    /* Append payload */
    coap_packet_append_payload_marker(&response);
    coap_packet_append_payload(&response, (uint8_t *)payload, strlen(payload));

    return coap_resource_send(resource, &response, addr, addr_len, NULL);
}

static int temp_get(struct coap_resource *resource, struct coap_packet *request,
                    struct net_sockaddr *addr, socklen_t addr_len)
{
    uint8_t token[COAP_TOKEN_MAX_LEN];
    uint16_t id;
    uint8_t tkl;
    int r;

    /* Let the CoAP server parse the request and add/remove observers if needed */
    r = coap_resource_parse_observe(resource, request, addr);

    id = coap_header_get_id(request);
    tkl = coap_header_get_token(request, token);

    return send_temperature(resource, addr, addr_len, r == 0 ? resource->age : 0,
                            id, token, tkl, true);
}

static void temp_notify(struct coap_resource *resource, struct coap_observer␣
↪*observer)
{
    send_temperature(resource, net_sad(&observer->addr), sizeof(observer->addr),␣
↪resource->age,
                     0, observer->token, observer->tkl, false);
}
```

(continues on next page)

(continued from previous page)

```
static const char * const temp_resource_path[] = { "sensors", "temp1", NULL };
COAP_RESOURCE_DEFINE(temp_resource, my_service, {
    .path = temp_resource_path,
    .get = temp_get,
    .notify = temp_notify,
});

static void notify_observers(struct k_work *work)
{
    if (sys_slist_is_empty(&temp_resource.observers)) {
        return;
    }

    coap_resource_notify(&temp_resource);
    k_work_reschedule(&temp_work, K_SECONDS(1));
}
```

CoAP Events By enabling `CONFIG_NET_MGMT_EVENT` the user can register for CoAP events. The following example simply prints when an event occurs.

```
#include <zephyr/sys/printk.h>
#include <zephyr/net/coap_mgmt.h>
#include <zephyr/net/coap_service.h>

#define COAP_EVENTS_SET (NET_EVENT_COAP_OBSERVER_ADDED |␣
↪NET_EVENT_COAP_OBSERVER_REMOVED | \
                         NET_EVENT_COAP_SERVICE_STARTED |␣
↪NET_EVENT_COAP_SERVICE_STOPPED)

void coap_event_handler(uint64_t mgmt_event, struct net_if *iface,
                        void *info, size_t info_length, void *user_data)
{
    switch (mgmt_event) {
    case NET_EVENT_COAP_OBSERVER_ADDED:
        printk("CoAP observer added");
        break;
    case NET_EVENT_COAP_OBSERVER_REMOVED:
        printk("CoAP observer removed");
        break;
    case NET_EVENT_COAP_SERVICE_STARTED:
        if (info != NULL && info_length == sizeof(struct net_event_coap_service)) {
            struct net_event_coap_service *net_event = info;

            printk("CoAP service %s started", net_event->service->name);
        } else {
            printk("CoAP service started");
        }
        break;
    case NET_EVENT_COAP_SERVICE_STOPPED:
        if (info != NULL && info_length == sizeof(struct net_event_coap_service)) {
            struct net_event_coap_service *net_event = info;

            printk("CoAP service %s stopped", net_event->service->name);
        } else {
            printk("CoAP service stopped");
```

(continues on next page)

(continued from previous page)

```
        }
        break;
    }
}

NET_MGMT_REGISTER_EVENT_HANDLER(coap_events, COAP_EVENTS_SET, coap_event_handler,␣
↪NULL);
```

CoRE Link Format The `CONFIG_COAP_SERVER_WELL_KNOWN_CORE` option enables handling the `.well-known/core` GET requests by the server. This allows clients to get a list of hypermedia links to other resources hosted in that server.

API Reference

CoAP service API

Related code samples

- CoAP serviceUse the CoAP server subsystem to register CoAP resources.

CoAP Manager Events

OSCORE Support (RFC 8613)

- *Overview*
- *Configuration*
- *Server Usage*
- *Known Limitations*
- *Security Context Derivation*
- *Security Considerations*
  - *Handling OSCORE When Not Supported*
- *API Reference*

Overview The Zephyr CoAP library provides support for Object Security for Constrained RESTful Environments (OSCORE) as specified in **RFC 8613**. OSCORE provides end-to-end protection of CoAP messages using COSE (CBOR Object Signing and Encryption).

OSCORE protects CoAP messages at the application layer, providing:

1. **Confidentiality**: Message payloads and sensitive options are encrypted
2. **Integrity**: Messages are authenticated with a MAC
3. **Replay protection**: Sequence numbers prevent replay attacks
4. **Proxy-friendly**: Outer options remain visible for routing

Unlike DTLS, OSCORE provides end-to-end security that survives proxy translation between different transport protocols (UDP, TCP, HTTP).

Additional OSCORE configuration options:

- `CONFIG_COAP_OSCORE_MAX_CONTEXTS`: Maximum number of OSCORE security contexts

- CONFIG_COAP_OSCORE_EXCHANGE_CACHE_SIZE: Number of OSCORE exchanges to track per service
- **CONFIG_COAP_OSCORE_EXCHANGE_LIFETIME_MS: Lifetime of tracked OSCORE exchanges used to protect**
  deferred (separate) responses
- CONFIG_COAP_OSCORE_CONTEXT_REUSE: Enable OSCORE support for context reuse across reboots
- CONFIG_COAP_OSCORE_MASTER_SECRET_MAX_LEN: Maximum OSCORE Master Secret length in bytes
- CONFIG_COAP_OSCORE_MASTER_SALT_MAX_LEN: Maximum OSCORE Master Salt length in bytes

Configuration Enable OSCORE support with CONFIG_COAP_OSCORE. This option depends on the uoscore-uedhoc module and PSA Crypto support:

```
CONFIG_COAP_OSCORE=y
CONFIG_UOSCORE=y
CONFIG_PSA_CRYPTO=y
```

The uoscore module automatically selects required PSA crypto algorithms (AES-CCM, HKDF-SHA256, etc.).

Server Usage To enable OSCORE on a CoAP service, define the service with COAP_SERVICE_DEFINE_OSCORE (or COAPS_SERVICE_DEFINE_OSCORE for DTLS). The macro statically allocates the per-service OSCORE exchange cache. Security contexts are created separately through the Zephyr OSCORE API and added to a shared pool; applications do not include the underlying uoscore-uedhoc headers directly. Incoming requests are matched to the correct context by their Recipient ID and ID Context, so a single service can serve multiple clients, each with its own context:

```
#include <zephyr/net/coap_oscore.h>
#include <zephyr/net/coap_service.h>

static struct coap_oscore_context *my_oscore_ctx;

static uint16_t my_service_port = 5683;

/* Final argument "true" requires OSCORE for all requests. */
COAP_SERVICE_DEFINE_OSCORE(my_service, NULL, &my_service_port,
                           COAP_SERVICE_AUTOSTART, true);

int my_service_oscore_init(void)
{
    /* coap_oscore_context_add() copies the key material, so these
     * buffers need not outlive the call and can live on the stack.
     */
    const uint8_t master_secret[16] = { /* ... */ };
    const uint8_t master_salt[8] = { /* ... */ };
    const uint8_t sender_id[] = { /* ... */ };
    const uint8_t recipient_id[] = { /* ... */ };

    struct coap_oscore_init_params params = {
        .master_secret = master_secret,
        .master_secret_len = sizeof(master_secret),
        .sender_id = sender_id,
        .sender_id_len = sizeof(sender_id),
```

(continues on next page)

(continued from previous page)

```
        .recipient_id = recipient_id,
        .recipient_id_len = sizeof(recipient_id),
        .master_salt = master_salt,
        .master_salt_len = sizeof(master_salt),
        .aead_alg = COAP_OSCORE_AEAD_AES_CCM_16_64_128,
        .hkdf = COAP_OSCORE_HKDF_SHA_256,
        .fresh_master_secret_salt = true,
    };

    /* Add the context to the shared pool once its key material is
     * available. Call once per client identity (Recipient ID).
     */
    return coap_oscore_context_add(&params, &my_oscore_ctx);
}
```

The number of contexts that can be allocated at once is controlled by `CONFIG_COAP_OSCORE_MAX_CONTEXTS`. Release a context with `coap_oscore_context_remove()`. Contexts are reference-counted: while an in-flight request, an active exchange, or an observer still references a context, `coap_oscore_context_remove()` returns `-EAGAIN` and the context is retained. Retry the removal once those references have been released.

When a service is OSCORE-enabled (created with an `OSCORE` macro and at least one context is added to the pool):

1. **Incoming requests**: The server automatically verifies and decrypts OSCORE-protected requests (**RFC 8613** Section 8.2). Resource handlers receive decrypted CoAP messages with Inner options visible.
2. **Outgoing responses**: The server automatically OSCORE-protects responses and notifications that originate from an OSCORE exchange (**RFC 8613** Section 8.3). Whether a given outgoing message must be protected is decided as follows:
   - **Synchronous responses, produced while the request is being handled, are matched** against the per-service exchange cache and protected when a matching entry is found. These entries expire after `CONFIG_COAP_OSCORE_EXCHANGE_LIFETIME_MS`. On a mixed service (OSCORE and non-OSCORE clients), sending a synchronous response after the exchange cache entry has expired will result in a plaintext response.
   - **Observe notifications are protected based on the observer's stored OSCORE** state, which lives for the duration of the observation.
   - **Deferred (separate) responses, produced after the request handler has** returned, are matched against the per-service exchange cache and protected when a matching entry is found. These entries expire after `CONFIG_COAP_OSCORE_EXCHANGE_LIFETIME_MS`. On a mixed service (OSCORE and non-OSCORE clients), sending a deferred response after the exchange cache entry has expired will result in a plaintext response.
3. **Error handling: OSCORE verification errors are sent as simple CoAP responses without** OSCORE processing (**RFC 8613** Section 8.2): - COSE decode failure 4.02 Bad Option - Security context not found 4.01 Unauthorized - Decryption failure 4.00 Bad Request
4. **Required OSCORE: If the service is defined with the `_oscore_required` argument** set to true, unprotected requests are rejected with 4.01 Unauthorized.
5. **Fail-closed behavior**: If OSCORE protection of a response fails, the server does not fall back to sending a plaintext response. On an OSCORE-required service, an outgoing response that cannot be matched to any OSCORE state is also dropped rather than sent in the clear. Observe notifications are never downgraded. On a mixed service, any response (synchronous

or deferred) whose exchange cache entry has expired can no longer be matched and is sent unprotected. In practice this affects deferred (separate) responses, because a synchronous response is matched against an entry that was just created while the request was handled (see item 2 and `CONFIG_COAP_OSCORE_EXCHANGE_LIFETIME_MS`).

Known Limitations

1. **Mixed-Service Expired-Exchange Plaintext**: On a mixed service (one that serves both OSCORE and non-OSCORE clients), a response whose exchange cache entry has expired can no longer be matched to its OSCORE state and is sent as plaintext. This affects both synchronous and deferred (separate) responses (see `CONFIG_COAP_OSCORE_EXCHANGE_LIFETIME_MS`). For any service that carries sensitive data, define it as OSCORE-required (pass the `_oscore_required` argument as true, e.g. via `COAP_SERVICE_DEFINE_OSCORE`) instead of running a mixed service. An OSCORE-required service rejects unprotected requests and drops responses that cannot be protected, so it never downgrades to plaintext.

Security Context Derivation OSCORE security contexts are derived from a small set of parameters (**RFC 8613** Section 3):

Required parameters:

- **Master Secret**: Shared secret (typically 16 bytes for AES-CCM-16-64-128)
- **Sender ID**: Unique identifier for the sender
- **Recipient ID**: Unique identifier for the recipient

Optional parameters:

- **Master Salt**: Additional entropy (recommended, typically 8 bytes)
- **ID Context**: Additional context identifier
- **AEAD Algorithm**: Defaults to AES-CCM-16-64-128
- **KDF**: Defaults to HKDF-SHA-256

These parameters are typically established through:

1. **Pre-shared keys**: Configured at device provisioning
2. **EDHOC**: Ephemeral Diffie-Hellman Over COSE (see uoscore-uedhoc module)

Security Considerations

1. **Sequence number overflow**: The sender sequence number (SSN) must not exceed 2^23-1 for AES-CCM-16-64-128. The uoscore library enforces this limit.
2. **Master secret protection**: Master secrets must be stored securely (e.g., in secure storage or derived from EDHOC).
3. **Persistence across reboots**: If the same master secret is reused after a reboot (i.e., master secrets are not re-derived, e.g., via EDHOC), the sender sequence number must be persisted to non-volatile memory to prevent nonce reuse, which would break confidentiality and integrity. The receiver's replay window does not need to be persisted: it is kept in memory and, after a reboot, is re-synchronized using the Echo option as described in **RFC 8613** Appendix B.1.2.

Handling OSCORE When Not Supported When OSCORE support is not enabled (`CONFIG_COAP_OSCORE` is not set), the Zephyr CoAP stack implements fail-closed behavior for the OSCORE option per **RFC 7252** Section 5.4.1:

Server behavior (when `CONFIG_COAP_OSCORE=n`):

- **CON requests** with OSCORE option: Returns **4.02 (Bad Option)** response
- **NON requests** with OSCORE option: Silently rejects (drops) the message
- **Responses** with OSCORE option: Sends RST for CON, silently drops NON/ACK

API Reference
CoAP OSCORE API

FTP client

- *Overview*
- *Protocols*
- *Limitations*
- *API Reference*

Overview The FTP client library can be used to download or upload files to FTP server.

The FTP client library reports FTP control message and download data with two separate callback functions `ftp_client.ctrl_callback` and `ftp_client.data_callback`. The library can be configured to automatically send KEEPALIVE message to the server through a timer if `CONFIG_FTP_CLIENT_KEEPALIVE_TIME` is not zero. The KEEPALIVE message is sent periodically at the completion of the time interval indicated by the value of `CONFIG_FTP_CLIENT_KEEPALIVE_TIME`.

Protocols The library is implemented in accordance with the **RFC 959** specification.

Limitations The library implements only a minimal set of commands as of now. However, new command support can be easily added.

Due to the differences in the implementation of FTP servers, the library might need customization to work with a specific server.

API Reference
FTP client library API

Related code samples

- FTP client shell sampleFTP client sample interacting with FTP server over Zephyr shell.

HTTP Client

- *Overview*
- *Sample Usage*
- *API Reference*

Overview The HTTP client library allows you to send HTTP requests and parse HTTP responses. The library communicates over the sockets API but it does not create sockets on its own. It can be enabled with `CONFIG_HTTP_CLIENT` Kconfig option.

The application must be responsible for creating a socket and passing it to the library. Therefore, depending on the application's needs, the library can communicate over either a plain TCP socket (HTTP) or a TLS socket (HTTPS).

Sample Usage The API of the HTTP client library has a single function.

The following is an example of a request structure created correctly:

```
struct http_request req = { 0 };
static uint8_t recv_buf[512];

req.method = HTTP_GET;
req.url = "/";
req.host = "localhost";
req.protocol = "HTTP/1.1";
req.response = response_cb;
req.recv_buf = recv_buf;
req.recv_buf_len = sizeof(recv_buf);

/* sock is a file descriptor referencing a socket that has been connected
 * to the HTTP server.
 */
ret = http_client_req(sock, &req, 5000, NULL);
```

If the server responds to the request, the library provides the response to the application through the response callback registered in the request structure. As the library can provide the response in chunks, the application must be able to process these.

Together with the structure containing the response data, the callback function also provides information about whether the library expects to receive more data.

The following is an example of a very simple response handling function:

```
static int response_cb(struct http_response *rsp,
                       enum http_final_call final_data,
                       void *user_data)
{
    if (final_data == HTTP_DATA_MORE) {
        LOG_INF("Partial data received (%zd bytes)", rsp->data_len);
    } else if (final_data == HTTP_DATA_FINAL) {
        LOG_INF("All the data received (%zd bytes)", rsp->data_len);
    }

    LOG_INF("Response status %s", rsp->http_status);

    return 0;
}
```

See HTTP client sample application for more information about the library usage.

API Reference

HTTP client API

Related code samples

- HTTP ClientImplement an HTTP(S) client that issues a variety of HTTP requests.
- TagoIO HTTP PostSend random temperature values to TagoIO IoT Cloud Platform using HTTP.

HTTP Server

- *Overview*
- *Server Setup*
- *Sample Usage*
  - *Services*
  - *Per-service configuration*
  - *Fallback Resources*
  - *Static resources*
  - *Static filesystem resources*
  - *Dynamic resources*
  - *Websocket resources*
  - *Accessing request headers*
- *API Reference*

Overview Zephyr provides an HTTP server library, which allows to register HTTP services and HTTP resources associated with those services. The server creates a listening socket for every registered service, and handles incoming client connections. It's possible to communicate over a plain TCP socket (HTTP) or a TLS socket (HTTPS). HTTP/1.1 (**RFC 2616**), HTTP/2 (**RFC 9113**) and HTTP/3 (**RFC 9114**) protocol versions are supported.

The server operation is generally transparent for the application, running in a background thread. The application can control the server activity with respective API functions.

Certain resource types (for example dynamic resource) provide resource-specific application callbacks, allowing the server to interact with the application (for instance provide resource content, or process request payload).

Currently, the following resource types are supported:

- Static resources - content defined compile-time, cannot be modified at runtime (`HTTP_RESOURCE_TYPE_STATIC`).
- Static file system resources - the path at which the filesystem is mounted, and the URL at which the filesystem is made available are fixed at build time, but the content within the filesystem can be changed dynamically. This means that the files can be created, modified or deleted by some other code outside the HTTP server (`HTTP_RESOURCE_TYPE_STATIC_FS`).
- Dynamic resources - content provided at runtime by respective application callback (`HTTP_RESOURCE_TYPE_DYNAMIC`).
- Websocket resources - allowing to establish Websocket connections with the server (`HTTP_RESOURCE_TYPE_WEBSOCKET`).

Zephyr provides a sample demonstrating HTTP(s) server operation and various resource types usage. See sockets-http-server for more information.

Server Setup A few prerequisites are needed in order to enable HTTP server functionality in the application.

First of all, the HTTP server has to be enabled in applications configuration file with `CONFIG_HTTP_SERVER` Kconfig option:

Listing 4: `prj.conf`

```
CONFIG_HTTP_SERVER=y
```

All HTTP services and HTTP resources are placed in a dedicated linker section. The linker section for services is predefined locally, however the application is responsible for defining linker sections for resources associated with respective services. Linker section names for resources should be prefixed with `http_resource_desc_`, appended with the service name.

Linker sections for resources should be defined in a linker file. For example, for a service named `my_service`, the linker section shall be defined as follows:

Listing 5: `sections-rom.ld`

```
#include <zephyr/linker/iterable_sections.h>

ITERABLE_SECTION_ROM(http_resource_desc_my_service, Z_LINK_ITERABLE_SUBALIGN)
```

Finally, the linker file and linker section have to be added to your application using CMake:

Listing 6: `CMakeLists.txt`

```
zephyr_linker_sources(SECTIONS sections-rom.ld)
zephyr_linker_section(NAME http_resource_desc_my_service
                      KVMA RAM_REGION GROUP RODATA_REGION)
```

> **Note**
>
> You need to define a separate linker section for each HTTP service registered in the system.

Sample Usage

Services The application needs to define an HTTP service (or multiple services), with the same name as used for the linker section with `HTTP_SERVICE_DEFINE` macro:

```
#include <zephyr/net/http/service.h>

static uint16_t http_service_port = 80;

HTTP_SERVICE_DEFINE(my_service, "0.0.0.0", &http_service_port, 1, 10, NULL, NULL,␣
↪NULL);
```

Alternatively, an HTTPS service can be defined with `HTTPS_SERVICE_DEFINE`:

```
#include <zephyr/net/http/service.h>
#include <zephyr/net/tls_credentials.h>

#define HTTP_SERVER_CERTIFICATE_TAG 1

static uint16_t https_service_port = 443;
static const sec_tag_t sec_tag_list[] = {
    HTTP_SERVER_CERTIFICATE_TAG,
```

(continues on next page)

(continued from previous page)

```
};

HTTPS_SERVICE_DEFINE(my_service, "0.0.0.0", &https_service_port, 1, 10,
                     NULL, NULL, NULL, sec_tag_list, sizeof(sec_tag_list));
```

Per-service configuration HTTP services support individual service configuration, for now it includes only socket creation through the `http_service_config` structure. This allows applications to customize socket creation behavior, for example to set specific socket options or use custom socket types.

To use custom socket creation:

```
static int my_socket_create(const struct http_service_desc *svc, int af, int proto)
{
    int fd;

    /* Create socket with custom parameters */
    fd = zsock_socket(af, SOCK_STREAM, proto);
    if (fd < 0) {
        return fd;
    }

    /* Set custom socket options */
    /* Add any other custom socket configuration */

    return fd;
}

static const struct http_service_config my_service_config = {
    .socket_create = my_socket_create,
};

static uint16_t http_service_port = 80;

HTTP_SERVICE_DEFINE(my_service, "0.0.0.0", &http_service_port, 1, 10,
                    NULL, NULL, &my_service_config);
```

The custom socket creation function receives: - `svc`: Pointer to the service descriptor - `af`: Address family (NET_AF_INET or NET_AF_INET6) - `proto`: Protocol (NET_IPPROTO_TCP or NET_IPPROTO_TLS_1_2 for HTTPS)

The function should return the socket file descriptor on success, or a negative error code on failure.

If no custom configuration is needed, simply pass `NULL` for the config parameter:

```
HTTP_SERVICE_DEFINE(my_service, "0.0.0.0", &http_service_port, 1, 10,
                    NULL, NULL, NULL);
```

Fallback Resources The `_res_fallback` parameter can be used when defining an HTTP/HTTPS service to specify a fallback resource which will be used if no other resource matches the URL. This can be used for example to serve an index page for all unknown paths (useful for a single-page app which handles routing in the frontend), or for a customised 404 response.

```
static int default_handler(struct http_client_ctx *client, enum␣
↪http_transaction_status status,
```

(continues on next page)

(continued from previous page)

```
                    const struct http_request_ctx *request_ctx,
                    struct http_response_ctx *response_ctx, void *user_data)
{
    static const char response_404[] = "Oops, page not found!";

    if (status == HTTP_SERVER_REQUEST_DATA_FINAL) {
        response_ctx->status = 404;
        response_ctx->body = response_404;
        response_ctx->body_len = sizeof(response_404) - 1;
        response_ctx->final_chunk = true;
    }

    return 0;
}

static struct http_resource_detail_dynamic default_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_DYNAMIC,
        .bitmask_of_supported_http_methods = BIT(HTTP_GET),
    },
    .cb = default_handler,
    .user_data = NULL,
};

/* Register a fallback resource to handle any unknown path */
HTTP_SERVICE_DEFINE(my_service, "0.0.0.0", &http_service_port, 1, 10, NULL, &
↪default_detail, NULL);
```

Note

HTTPS services rely on TLS credentials being registered in the system. See *TLS credentials subsystem* for information on how to configure TLS credentials in the system.

Once HTTP(s) service is defined, resources can be registered for it with `HTTP_RESOURCE_DEFINE` macro.

Application can enable resource wildcard support by enabling `CONFIG_HTTP_SERVER_RESOURCE_WILDCARD` option. When this option is set, then it is possible to match several incoming HTTP requests with just one resource handler. The fnmatch() POSIX API function is used to match the pattern in the URL paths.

Example:

```
HTTP_RESOURCE_DEFINE(my_resource, my_service, "/foo*", &resource_detail);
```

This would match all URLs that start with a string `foo`. See POSIX.2 chapter 2.13 for pattern matching syntax description.

Static resources Static resource content is defined build-time and is immutable. The following example shows how gzip compressed webpage can be defined as a static resource in the application:

```
static const uint8_t index_html_gz[] = {
    #include "index.html.gz.inc"
};
```

(continues on next page)

(continued from previous page)

```
struct http_resource_detail_static index_html_gz_resource_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_STATIC,
        .bitmask_of_supported_http_methods = BIT(HTTP_GET),
        .content_encoding = "gzip",
    },
    .static_data = index_html_gz,
    .static_data_len = sizeof(index_html_gz),
};

HTTP_RESOURCE_DEFINE(index_html_gz_resource, my_service, "/",
                     &index_html_gz_resource_detail);
```

The resource content and content encoding is application specific. For the above example, a gzip compressed webpage can be generated during build, by adding the following code to the application's `CMakeLists.txt` file:

Listing 7: `CMakeLists.txt`

```
set(gen_dir ${ZEPHYR_BINARY_DIR}/include/generated/)
set(source_file_index src/index.html)
generate_inc_file_for_target(app ${source_file_index} ${gen_dir}/index.html.gz.inc␣
↪--gzip)
```

where `src/index.html` is the location of the webpage to be compressed.

Static filesystem resources Static filesystem resource content is defined build-time and is immutable. Note that only `GET` operation is supported, user is not able to upload files to the filesystem. The following example shows how the path can be defined as a static resource in the application:

```
struct http_resource_detail_static_fs static_fs_resource_detail = {
    .common = {
        .type                              = HTTP_RESOURCE_TYPE_STATIC_FS,
        .bitmask_of_supported_http_methods = BIT(HTTP_GET),
    },
    .fs_path = "/lfs1/www",
};

HTTP_RESOURCE_DEFINE(static_fs_resource, my_service, "*", &
↪static_fs_resource_detail);
```

All files located in /lfs1/www are made available to the client. If a file is gzipped, .gz must be appended to the file name (e.g. index.html.gz), then the server delivers index.html.gz when the client requests index.html and adds gzip content-encoding to the HTTP header.

The content type is evaluated based on the file extension. The server supports .html, .js, .css, .jpg, .png and .svg. More content types can be provided with the `HTTP_SERVER_CONTENT_TYPE` macro. All other files are provided with the content type text/html.

```
HTTP_SERVER_CONTENT_TYPE(json, "application/json")
```

When serving files from a static filesystem, the response chunk size can be configured using the `CONFIG_HTTP_SERVER_STATIC_FS_RESPONSE_SIZE` Kconfig option. This determines the size of individual chunks when transmitting file content to clients.

Dynamic resources For dynamic resource, a resource callback is registered to exchange data between the server and the application.

The following example code shows how to register a dynamic resource with a simple resource handler, which echoes received data back to the client:

```
static int dyn_handler(struct http_client_ctx *client, enum␣
↪http_transaction_status status,
                       const struct http_request_ctx *request_ctx,
                       struct http_response_ctx *response_ctx, void *user_data)
{
#define MAX_TEMP_PRINT_LEN 32
    static char print_str[MAX_TEMP_PRINT_LEN];
    enum http_method method = client->method;
    static size_t processed;

    __ASSERT_NO_MSG(request_ctx->data != NULL);

    if (status == HTTP_SERVER_TRANSACTION_ABORTED ||
        status == HTTP_SERVER_TRANSACTION_COMPLETE) {
        if (status == HTTP_SERVER_TRANSACTION_ABORTED) {
            LOG_DBG("Transaction aborted after %zd bytes.", processed);
        }
        processed = 0;
        return 0;
    }

    processed += request_ctx->data_len;

    snprintf(print_str, sizeof(print_str), "%s received (%zd bytes)",
             http_method_str(method), request_ctx->data_len);
    LOG_HEXDUMP_DBG(request_ctx->data, request_ctx->data_len, print_str);

    if (status == HTTP_SERVER_REQUEST_DATA_FINAL) {
        LOG_DBG("All data received (%zd bytes).", processed);
        processed = 0;
    }

    /* Echo data back to client */
    response_ctx->body = request_ctx->data;
    response_ctx->body_len = request_ctx->data_len;
    response_ctx->final_chunk = (status == HTTP_SERVER_REQUEST_DATA_FINAL);

    return 0;
}

struct http_resource_detail_dynamic dyn_resource_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_DYNAMIC,
        .bitmask_of_supported_http_methods =
            BIT(HTTP_GET) | BIT(HTTP_POST),
    },
    .cb = dyn_handler,
    .user_data = NULL,
};

HTTP_RESOURCE_DEFINE(dyn_resource, my_service, "/dynamic",
                     &dyn_resource_detail);
```

The resource callback may be called multiple times for a single request, hence the application should be able to keep track of the received data progress.

The `status` field informs the application about the progress in passing request payload from the server to the application. As long as the status reports `HTTP_SERVER_REQUEST_DATA_MORE`, the application should expect more data to be provided in a consecutive callback calls. Once all request payload has been passed to the application, the server reports `HTTP_SERVER_REQUEST_DATA_FINAL` status. In case of communication errors during request processing (for example client closed the connection before complete payload has been received), the server reports `HTTP_SERVER_TRANSACTION_ABORTED`. When the response has been sent completely to the client, the server reports `HTTP_SERVER_TRANSACTION_COMPLETE` status. Either of the two events indicate that the request processing is finished, and the application shall reset any progress recorded for the resource, and await a new request to come. The server guarantees that the resource can only be accessed by single client at a time.

The `request_ctx` parameter is used to pass request data to the application:

- The `data` and `data_len` fields pass request data to the application.
- The `headers`, `header_count` and `headers_status` fields pass request headers to the application, if `CONFIG_HTTP_SERVER_CAPTURE_HEADERS` is enabled. These fields are only populated in the first callback of a request; see *Accessing request headers* for details.

The `response_ctx` field is used by the application to pass response data to the HTTP server:

- The `status` field allows the application to send an HTTP response code. If not populated, the response code will be 200 by default.
- The `headers` and `header_count` fields can be used for the application to send any arbitrary HTTP headers. If not populated, only Transfer-Encoding and Content-Type are sent by default. The callback may override the Content-Type if desired.
- The `body` and `body_len` fields are used to send body data.
- The `final_chunk` field is used to indicate that the application has no more response data to send.

Headers and/or response codes may only be sent in the first populated `response_ctx`, after which only further body data is allowed in subsequent callbacks.

The server will call the resource callback until it provided all request data to the application, and the application reports there is no more data to include in the reply.

Websocket resources Websocket resources register an application callback, which is called when a Websocket connection upgrade takes place. The callback is provided with a socket descriptor corresponding to the underlying TCP/TLS connection. Once called, the application takes full control over the socket, i. e. is responsible to release it when done.

```
static int ws_socket;
static uint8_t ws_recv_buffer[1024];

int ws_setup(int sock, struct http_request_ctx *request_ctx, void *user_data)
{
    ws_socket = sock;
    return 0;
}

struct http_resource_detail_websocket ws_resource_detail = {
    .common = {
        .type = HTTP_RESOURCE_TYPE_WEBSOCKET,
        /* We need HTTP/1.1 Get method for upgrading */
        .bitmask_of_supported_http_methods = BIT(HTTP_GET),
    },
    .cb = ws_setup,
    .data_buffer = ws_recv_buffer,
```

(continues on next page)

(continued from previous page)

```
    .data_buffer_len = sizeof(ws_recv_buffer),
    .user_data = NULL, /* Fill this for any user specific data */
};

HTTP_RESOURCE_DEFINE(ws_resource, my_service, "/", &ws_resource_detail);
```

The above minimalistic example shows how to register a Websocket resource with a simple callback, used only to store the socket descriptor provided. Further processing of the Websocket connection is application-specific, hence outside of scope of this guide. See sockets-http-server for an example Websocket-based echo service implementation.

Accessing request headers The application can register an interest in any specific HTTP request headers. These headers are then stored for each incoming request, and can be accessed from within a dynamic resource callback.

> **Important**
>
> Captured request headers are delivered to the application **only in the first callback** for a given request. In any subsequent callback for the same request the `headers`, `header_count` and `headers_status` fields of `request_ctx` are cleared (`headers_status` becomes `HTTP_HEADER_STATUS_NONE`). Applications must copy out any header value they need during the first callback rather than reading it later.
>
> Note that the first callback may or may not also carry request body data (this depends on the HTTP method and on transport framing). Do not condition header handling on the presence of body data: always inspect `headers_status` and stash the headers when it is not `HTTP_HEADER_STATUS_NONE`.

This feature must first be enabled with `CONFIG_HTTP_SERVER_CAPTURE_HEADERS` Kconfig option.

Then the application can register headers to be captured, and read the values from within the dynamic resource callback. The recommended pattern is to copy out the headers of interest in the first callback, then act on them once the full request body has been received:

```
HTTP_SERVER_REGISTER_HEADER_CAPTURE(capture_user_agent, "User-Agent");

static char user_agent[64];

static int dyn_handler(struct http_client_ctx *client, enum␣
↪http_transaction_status status,
                       const struct http_request_ctx *request_ctx,
                       struct http_response_ctx *response_ctx, void *user_data)
{
    /* Request headers are only present in the first callback, so copy out
     * any values needed later before they are gone.
     */
    if (request_ctx->headers_status != HTTP_HEADER_STATUS_NONE) {
        for (size_t i = 0; i < request_ctx->header_count; i++) {
            const struct http_header *hdr = &request_ctx->headers[i];

            LOG_INF("Captured header: '%s: %s'", hdr->name, hdr->value);

            if (strcasecmp(hdr->name, "User-Agent") == 0) {
                strncpy(user_agent, hdr->value, sizeof(user_agent) - 1);
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }

    /* Process request body data (may be empty in the first callback). */

    if (status == HTTP_SERVER_REQUEST_DATA_FINAL) {
        /* Full request received: act on the body together with the header
         * values stashed above (e.g. user_agent).
         */
    }

    return 0;
}
```

API Reference

HTTP service API

Related code samples

- HTTP ServerImplement an HTTP(s) Server demonstrating various resource types.
- Prometheus SampleImplement a Prometheus Metric Server demonstrating various metric types.

HTTP server API

Related code samples

- HTTP ServerImplement an HTTP(s) Server demonstrating various resource types.
- Prometheus SampleImplement a Prometheus Metric Server demonstrating various metric types.

Lightweight M2M (LwM2M)

- *Overview*
- *Example LwM2M object and resources: Device*
- *Sample usage*
- *LwM2M security modes*
- *Multi-thread usage*
- *Support for time series data*
  - *Enabling and configuring*
  - *Read and Write operations*
  - *Limitations*
- *Send scheduler helper objects*
  - *Enabling and wiring*
  - *Scheduler Control object (10523)*

- *Sampling Rules object (10524)*
- *LwM2M engine and application events*
- *Configuring lifetime and activity period*
- *LwM2M shell*
- *API Reference*

Overview Lightweight Machine to Machine (LwM2M) is an application layer protocol designed with device management, data reporting and device actuation in mind. Based on CoAP/UDP, LwM2M is a standard defined by the Open Mobile Alliance and suitable for constrained devices by its use of CoAP packet-size optimization and a simple, stateless flow that supports a REST API.

One of the key differences between LwM2M and CoAP is that an LwM2M client initiates the connection to an LwM2M server. The server can then use the REST API to manage various interfaces with the client.

LwM2M uses a simple resource model with the core set of objects and resources defined in the specification.

The LwM2M library can be enabled with `CONFIG_LWM2M` Kconfig option.

Example LwM2M object and resources: Device *Object definition*

Object ID	Name	Instance	Mandatory
3	Device	Single	Mandatory

Resource definitions

`* R=Read, W=Write, E=Execute`

ID	Name	OP*	Instance	Mandatory	Type
0	Manufacturer	R	Single	Optional	String
1	Model	R	Single	Optional	String
2	Serial number	R	Single	Optional	String
3	Firmware version	R	Single	Optional	String
4	Reboot	E	Single	Mandatory	
5	Factory Reset	E	Single	Optional	
6	Available Power Sources	R	Multiple	Optional	Integer 0-7
7	Power Source Voltage (mV)	R	Multiple	Optional	Integer
8	Power Source Current (mA)	R	Multiple	Optional	Integer
9	Battery Level %	R	Single	Optional	Integer
10	Memory Free (Kb)	R	Single	Optional	Integer
11	Error Code	R	Multiple	Optional	Integer 0-8
12	Reset Error	E	Single	Optional	
13	Current Time	RW	Single	Optional	Time
14	UTC Offset	RW	Single	Optional	String
15	Timezone	RW	Single	Optional	String
16	Supported Binding	R	Single	Mandatory	String
17	Device Type	R	Single	Optional	String
18	Hardware Version	R	Single	Optional	String
19	Software Version	R	Single	Optional	String
20	Battery Status	R	Single	Optional	Integer 0-6
21	Memory Total (Kb)	R	Single	Optional	Integer
22	ExtDevInfo	R	Multiple	Optional	ObjLnk

The server could query the `Manufacturer` resource for `Device` object instance 0 (the default and only instance) by sending a `READ 3/0/0` operation to the client.

The full list of registered objects and resource IDs can be found in the OMA LwM2M registries.

Zephyr's LwM2M library lives in the subsys/net/lib/lwm2m, with a client sample in samples/net/lwm2m_client. For more information about the provided sample see: lwm2m-client. The sample can be configured to use normal insecure network sockets or sockets secured via DTLS.

The Zephyr LwM2M library implements the following items:

- engine to process networking events and core functions
- RD client which performs BOOTSTRAP and REGISTRATION functions
- SenML CBOR, SenML JSON, CBOR, TLV, JSON, and plain text formatting functions
- LwM2M Technical Specification Enabler objects such as Security, Server, Device, Firmware Update, etc.
- Extended IPSO objects such as Light Control, Temperature Sensor, and Timer

By default, the library implements LwM2M specification 1.0.2 and can be set to LwM2M specification 1.1.1 with a Kconfig option.

For more information about LwM2M specification releases visit the OMA LwM2M releases page.

Sample usage To use the LwM2M library, start by creating an LwM2M client context `lwm2m_ctx` structure:

```
/* LwM2M client context */
static struct lwm2m_ctx client;
```

Create callback functions for LwM2M resource executions:

```
static int device_reboot_cb(uint16_t obj_inst_id, uint8_t *args,
                            uint16_t args_len)
{
        LOG_INF("Device rebooting.");
        LOG_PANIC();
        sys_reboot(0);
        return 0; /* won't reach this */
}
```

The LwM2M RD client can send events back to the sample. To receive those events, setup a callback function:

```
static void rd_client_event(struct lwm2m_ctx *client,
                            enum lwm2m_rd_client_event client_event)
{
        switch (client_event) {

        case LWM2M_RD_CLIENT_EVENT_NONE:
                /* do nothing */
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_FAILURE:
                LOG_DBG("Bootstrap registration failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_REG_COMPLETE:
                LOG_DBG("Bootstrap registration complete");
```

(continues on next page)

(continued from previous page)

```
                break;

        case LWM2M_RD_CLIENT_EVENT_BOOTSTRAP_TRANSFER_COMPLETE:
                LOG_DBG("Bootstrap transfer complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_REGISTRATION_FAILURE:
                LOG_DBG("Registration failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE:
                LOG_DBG("Registration complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_TIMEOUT:
                LOG_DBG("Registration timeout!");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE:
                LOG_DBG("Registration update complete");
                break;

        case LWM2M_RD_CLIENT_EVENT_DEREGISTER_FAILURE:
                LOG_DBG("Deregister failure!");
                break;

        case LWM2M_RD_CLIENT_EVENT_DISCONNECT:
                LOG_DBG("Disconnected");
                break;

        case LWM2M_RD_CLIENT_EVENT_REG_UPDATE:
                LOG_DBG("Registration update");
                break;

        case LWM2M_RD_CLIENT_EVENT_DEREGISTER:
                LOG_DBG("Deregistration client");
                break;

        case LWM2M_RD_CLIENT_EVENT_SERVER_DISABLED:
                LOG_DBG("LwM2M server disabled");
          break;
        }
}
```

Next we assign `Security` resource values to let the client know where and how to connect as well as set the `Manufacturer` and `Reboot` resources in the `Device` object with some data and the callback we defined above:

```
/*
 * Server URL of default Security object = 0/0/0
 * Use leshan.eclipse.org server IP (5.39.83.206) for connection
 */
lwm2m_set_string(&LWM2M_OBJ(0, 0, 0), "coap://5.39.83.206");

/*
 * Security Mode of default Security object = 0/0/2
```

(continues on next page)

(continued from previous page)

```
 * 3 = NoSec mode (no security beware!)
 */
lwm2m_set_u8(&LWM2M_OBJ(0, 0, 2), 3);

#define CLIENT_MANUFACTURER "Zephyr Manufacturer"

/*
 * Manufacturer resource of Device object = 3/0/0
 * We use lwm2m_set_res_data() function to set a pointer to the
 * CLIENT_MANUFACTURER string.
 * Note the LWM2M_RES_DATA_FLAG_RO flag which stops the engine from
 * trying to assign a new value to the buffer.
 */
lwm2m_set_res_data(&LWM2M_OBJ(3, 0, 0), CLIENT_MANUFACTURER,
                   sizeof(CLIENT_MANUFACTURER),
                   LWM2M_RES_DATA_FLAG_RO);

/* Reboot resource of Device object = 3/0/4 */
lwm2m_register_exec_callback(&LWM2M_OBJ(3, 0, 4), device_reboot_cb);
```

Lastly, we start the LwM2M RD client (which in turn starts the LwM2M engine). The second parameter of `lwm2m_rd_client_start()` is the client endpoint name. This is important as it needs to be unique per LwM2M server:

```
(void)memset(&client, 0x0, sizeof(client));
lwm2m_rd_client_start(&client, "unique-endpoint-name", 0, rd_client_event);
```

LwM2M security modes The Zephyr LwM2M library can be used either without security or use DTLS to secure the communication channel. When using DTLS with the LwM2M engine, PSK (Pre-Shared Key) and X.509 certificates are the security modes that can be used to secure the communication. The engine uses LwM2M Security object (Id 0) to read the stored credentials and feed keys from the security object into the TLS credential subsystem, see *secure sockets documentation*. Enable the `CONFIG_LWM2M_DTLS_SUPPORT` Kconfig option to use the security.

Depending on the selected mode, the security object must contain following data:

PSK
: Security Mode (Resource ID 2) set to zero (Pre-Shared Key mode). Identity (Resource ID 3) contains PSK ID in binary form. Secret key (Resource ID 5) contains the PSK key in binary form. If the key or identity is provided as a hex string, it must be converted to binary before storing into the security object.

X509
: When X509 certificates are used, set Security Mode (ID 2) to 2 (Certificate mode). Identity (ID 3) is used to store the client certificate and Secret key (ID 5) must have a private key associated with the certificate. Server Public Key resource (ID 4) must contain a server certificate or CA certificate used to sign the certificate chain. If the `CONFIG_MBEDTLS_PEM_PARSE_C` / `CONFIG_MBEDTLS_PEM_WRITE_C` Kconfig options are enabled, certificates and private keys can be entered in PEM format. Otherwise, they must be in binary DER format.

NoSec
: When no security is used, set Security Mode (Resource ID 2) to 3 (NoSec).

In all modes, Server URI resource (ID 0) must contain the full URI for the target server. When DNS names are used, the DNS resolver must be enabled.

When DTLS is used, following options are recommended to reduce DTLS handshake traffic when connection is re-established:

- CONFIG_LWM2M_DTLS_CID enables DTLS Connection Identifier support. When server supports it, this completely removes the handshake when device resumes operation after long idle period. Greatly helps when NAT mappings have timed out.
- CONFIG_LWM2M_TLS_SESSION_CACHING uses session cache when before falling back to full DTLS handshake. Reduces few packets from handshake, when session is still cached on server side. Most significant effect is to avoid full registration.

LwM2M stack provides callbacks in the lwm2m_ctx structure. They are used to feed keys from the LwM2M security object into the TLS credential subsystem. By default, these callbacks can be left as NULL pointers, in which case default callbacks are used. When an external TLS stack, or non-default socket options are required, you can overwrite the lwm2m_ctx.load_credentials() or lwm2m_ctx.set_socketoptions() callbacks.

An example of setting up the security object for PSK mode:

```
/* "000102030405060708090a0b0c0d0e0f" */
static unsigned char client_psk[] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
};

static const char client_identity[] = "Client_identity";

lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 0),
↪"coaps://lwm2m.example.com");
lwm2m_set_u8(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 2), LWM2M_SECURITY_PSK);
/* Set the client identity as a string, but this could be binary as well */
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 3), client_identity);
/* Set the client pre-shared key (PSK) */
lwm2m_set_opaque(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 5), client_psk,␣
↪sizeof(client_psk));
```

An example of setting up the security object for X509 certificate mode:

```
static const char certificate[] = "-----BEGIN CERTIFICATE-----\nMIIB6jCCAY+gAw...";
static const char key[] = "-----BEGIN EC PRIVATE KEY-----\nMHcCAQ...";
static const char root_ca[] = "-----BEGIN CERTIFICATE-----\nMIIBaz...";

lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 0),
↪"coaps://lwm2m.example.com");
lwm2m_set_u8(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 2), LWM2M_SECURITY_CERT);
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 3), certificate);
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 5), key);
lwm2m_set_string(&LWM2M_OBJ(LWM2M_OBJECT_SECURITY_ID, 0, 4), root_ca);
```

Before calling lwm2m_rd_client_start() assign the tls_tag # where the LwM2M library should store the DTLS information prior to connection (normally a value of 1 is ok here).

```
(void)memset(&client, 0x0, sizeof(client));
client.tls_tag = 1; /* <---- */
lwm2m_rd_client_start(&client, "endpoint-name", 0, rd_client_event);
```

For a more detailed LwM2M client sample see: lwm2m-client.

Multi-thread usage Writing a value to a resource can be done using functions like lwm2m_set_u8. When writing to multiple resources, the function lwm2m_registry_lock will ensure that the client halts until all writing operations are finished:

```
lwm2m_registry_lock();
lwm2m_set_u32(&LWM2M_OBJ(1, 0, 1), 60);
lwm2m_set_u8(&LWM2M_OBJ(5, 0, 3), 0);
lwm2m_set_f64(&LWM2M_OBJ(3303, 0, 5700), value);
lwm2m_registry_unlock();
```

This is especially useful if the server is composite-observing the resources being written to. Locking will then ensure that the client only updates and sends notifications to the server after all operations are done, resulting in fewer messages in general.

Support for time series data LwM2M version 1.1 adds support for SenML CBOR and SenML JSON data formats. These data formats add support for time series data. Time series formats can be used for READ, NOTIFY and SEND operations. When data cache is enabled for a resource, each write will create a timestamped entry in a cache, and its content is then returned as a content in READ, NOTIFY or SEND operation for a given resource.

Data cache is only supported for resources with a fixed data size.

Supported resource types:

- Signed and unsigned 8-64-bit integers
- Float
- Boolean

Enabling and configuring Enable data cache by selecting `CONFIG_LWM2M_RESOURCE_DATA_CACHE_SUPPORT`. Application needs to allocate an array of `lwm2m_time_series_elem` structures and then enable the cache by calling `lwm2m_enable_cache()` for a given resource. Each resource must be enabled separately and each resource needs their own storage.

```
/* Allocate data cache storage */
static struct lwm2m_time_series_elem temperature_cache[10];
/* Enable data cache */
lwm2m_enable_cache(LWM2M_OBJ(IPSO_OBJECT_TEMP_SENSOR_ID, 0, SENSOR_VALUE_RID),
        temperature_cache, ARRAY_SIZE(temperature_cache));
```

Applications can inspect the available space in a cached resource with `lwm2m_cache_free_slots_get()`, which returns the number of samples that can still be stored or a negative errno value if the query fails.

LwM2M engine have room for four resources that have cache enabled. Limit can be increased by changing `CONFIG_LWM2M_MAX_CACHED_RESOURCES`. This affects a static memory usage of engine.

Data caches depends on one of the SenML data formats `CONFIG_LWM2M_RW_SENML_CBOR_SUPPORT` or `CONFIG_LWM2M_RW_SENML_JSON_SUPPORT` and needs `CONFIG_POSIX_TIMERS` so it can request a timestamp from the system.

Read and Write operations Full content of data cache is written into a payload when any READ, SEND or NOTIFY operation internally reads the content of a given resource. This has a side effect that any read callbacks registered for a that resource are ignored when cache is enabled. Data is written into a cache when any of the `lwm2m_set_*` functions are called. Applications can register a cache filter callback with `lwm2m_set_cache_filter()` to drop otherwise valid samples based on application-specific rules.

Limitations Cache size should be manually set so small that the content can fit normal packets sizes. When cache is full, new values are dropped.

Send scheduler helper objects The optional SEND scheduler extension exposes two objects (Send scheduler Control `10523` and Sampling Rules `10524`) that sit on top of cached resources to decide when samples should be kept and when the client should trigger a LwM2M SEND.

Enabling and wiring

- Select `CONFIG_LWM2M_SEND_SCHEDULER` (requires LwM2M 1.1 SEND support and `CONFIG_LWM2M_RESOURCE_DATA_CACHE_SUPPORT`).
- Send-scheduler objects register automatically; ensure caches are configured before starting the RD client.
- Attach `lwm2m_send_sched_cache_filter()` to every cached resource that should be governed by the scheduler using `lwm2m_set_cache_filter()`.
- Call `lwm2m_send_sched_handle_registration_event()` from the RD client callback when registration or registration-update completes (on `LWM2M_RD_CLIENT_EVENT_REGISTRATION_COMPLETE` and `LWM2M_RD_CLIENT_EVENT_REG_UPDATE_COMPLETE`) so cached samples are sent right after the registration exchange.

Scheduler Control object (10523) The single-instance control object configures global behaviour:

- `0: paused` – stop accepting samples in cache.
- `1: max-samples` – upper limit for cached samples across all resources; a SEND is forced when the limit is reached (`0` disables).
- `2: max-age` – maximum age in seconds of the oldest cached sample before a SEND is triggered (`0` disables).
- `3: flush` – Execute resource that immediately triggers a SEND for all configured rule paths.
- `4: flush-on-update` – when enabled (default), a successful registration or registration-update event triggers a SEND of cached resources.

Sampling Rules object (10524) Each instance describes one cached resource to watch. `/10524/X/0` holds the resource path and `/10524/X/1` contains up to four rule strings. Supported attributes:

- `gt=<float>` – trigger when the sample crosses above the threshold.
- `lt=<float>` – trigger when the sample crosses below the threshold.
- `st=<float>` – trigger when the absolute delta from the last reported value is greater than or equal to the threshold.
- `pmin=<int>` – minimum seconds between accepted samples.
- `pmax=<int>` – force a cached sample to be kept at least every `pmax` seconds, even without changes.

When no rules are configured for an instance, every incoming sample is cached. The scheduler also forces a SEND when a cache for a controlled resource runs out of space or when the global `max-samples`/`max-age` limits are reached.

LwM2M engine and application events The Zephyr LwM2M engine defines events that can be sent back to the application through callback functions. The engine state machine shows when the events are spawned. Events depicted in the diagram are listed in the table. The events are prefixed with `LWM2M_RD_CLIENT_EVENT_`.

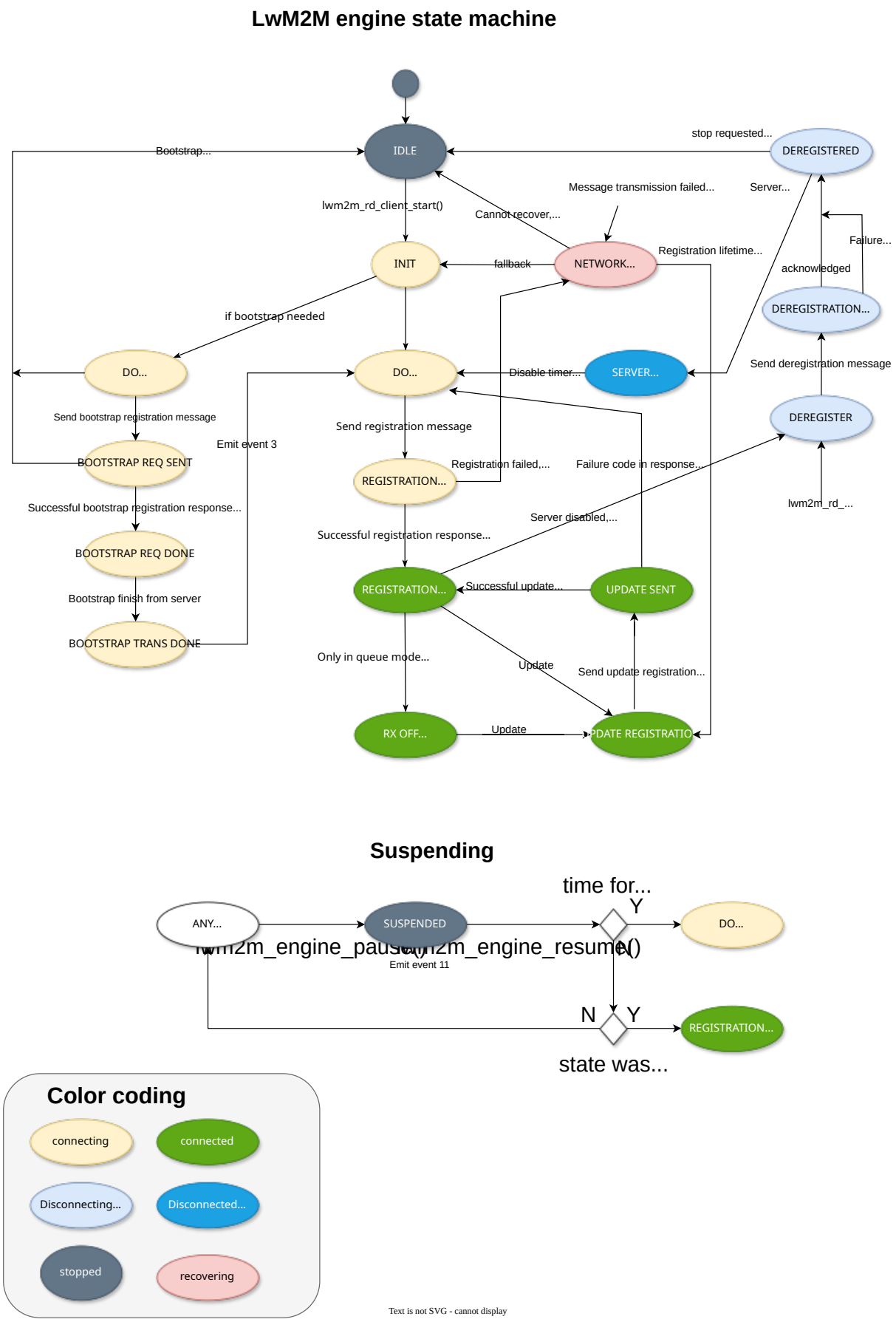


Fig. 6: State machine for the LwM2M engine

Table 6: LwM2M RD Client events

Ever ID	Event Name	Description
0	NONE	No event
1	BOOT-STRAP_REG_F	Bootstrap registration failed. Occurs if there is a timeout or failure in bootstrap registration.
2	BOOT-STRAP_REG_C	Bootstrap registration complete. Occurs after successful bootstrap registration.
3	BOOT-STRAP_TRANS	Bootstrap finish command received from the server.
4	REGISTRA-TION_FAILUR	Registration to LwM2M server failed. Occurs if server rejects the registration attempt.
5	REGISTRA-TION_COMPL	Registration to LwM2M server successful. Occurs after a successful registration reply from the LwM2M server or when session resumption is used.
6	REG_TIMEOU	Registration status lost. Occurs if there is socket errors or message timeouts. Client have lost connection to the server.
7	REG_UPDATE_	Registration update completed. Occurs after successful registration update reply from the LwM2M server.
8	DEREGIS-TER_FAILURE	Deregistration to LwM2M server failed. Occurs if there is a timeout or failure in the deregistration.
9	DISCON-NECT	LwM2M client have de-registered from server and is now stopped. Triggered only if the application have requested the client to stop.
10	QUEUE_MODI	Used only in queue mode, not actively listening for incoming packets. In queue mode the client is not required to actively listen for the incoming packets after a configured time period.
11	EN-GINE_SUSPEN	Indicate that client has now paused as a result of calling `lwm2m_engine_pause()`. State machine is no longer running and the handler thread is suspended. All timers are stopped so notifications are not triggered.
12	SERVER_DISA	Server have executed the disable command. Client will deregister and stay idle for the disable period.
13	NET-WORK_ERROI	Sending messages to the network failed too many times. Client cannot reach any servers or fallback to bootstrap. LwM2M engine cannot recover and have stopped.

The LwM2M client engine handles most of the state transitions automatically. The application needs to handle only the events that indicate that the client have stopped or is in a state where it cannot recover.

Table 7: How application should react to events

Event Name	How application should react
NONE	Ignore the event.
BOOTSTRAP_REG_FAIL	Try to recover network connection. Then restart the client by calling `lwm2m_rd_client_start()`. This might also indicate configuration issue.
BOOTSTRAP_REG_COM	No actions needed
BOOTSTRAP_TRANSFE	No actions needed
REGISTRATION_FAILURE	No actions needed. Client proceeds re-registration automatically. Might need a bootstrap or configuration fix. Cannot send or receive data.
REGISTRATION_COMPLETE	No actions needed. Application can send or receive data.
REG_TIMEOUT	No actions needed. Client proceeds to re-registration automatically. Cannot send or receive data.
REG_UPDATE_CO	No actions needed Application can send or receive data.
DEREGISTER_FAILURE	No actions needed, client proceeds to idle state automatically. Cannot send or receive data.
DISCONNECT	Engine have stopped as a result of calling `lwm2m_rd_client_stop()`. If connection is required, the application should restart the client by calling `lwm2m_rd_client_start()`.
QUEUE_MODE_R	No actions needed. Application can send but cannot receive data. Any data transmission will trigger a registration update.
ENGINE_SUSPENDE	Engine can be resumed by calling `lwm2m_engine_resume()`. Cannot send or receive data.
SERVER_DISABLI	No actions needed, client will re-register once the disable period is over. Cannot send or receive data.
NETWORK_ERROR	Try to recover network connection. Then restart the client by calling `lwm2m_rd_client_start()`. This might also indicate configuration issue.

Sending of data in the table above refers to calling `lwm2m_send_cb()` or by writing into one of the observed resources where observation would trigger a notify message. Receiving of data refers to receiving read, write or execute operations from the server. Application can register callbacks for these operations.

Configuring lifetime and activity period In LwM2M engine, there are three Kconfig options and one runtime value that configures how often the client will send LwM2M Update message.

Table 8: Update period variables

Variable	Effect
LwM2M registration lifetime	The lifetime parameter in LwM2M specifies how long a device's registration with an LwM2M server remains valid. Device is expected to send LwM2M Update message before the lifetime exprires.
`CONFIG_LWM2M_ENGIN`	Default lifetime value, unless set by the bootstrap server. Also defines lower limit that client accepts as a lifetime.
`CONFIG_LWM2M_UPDAT`	How long the client can stay idle before sending a next update.
`CONFIG_LWM2M_SECON`	Minimum time margin to send the update message before the registration lifetime expires.

By default, the client uses `CONFIG_LWM2M_SECONDS_TO_UPDATE_EARLY` to calculate how many seconds before the expiration of lifetime it is going to send the registration update. The problem with default mode is when the server changes the lifetime of the registration. This is then affecting the period of updates the client is doing. If this is used with the QUEUE mode, which is

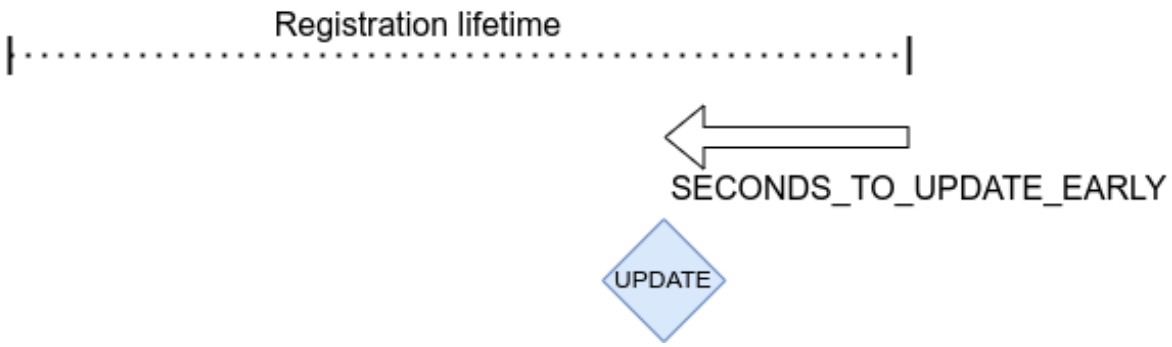


Fig. 7: Default way of calculating when to update registration.

typical in IPv4 networks, it is also affecting the period of when the device is reachable from the server.

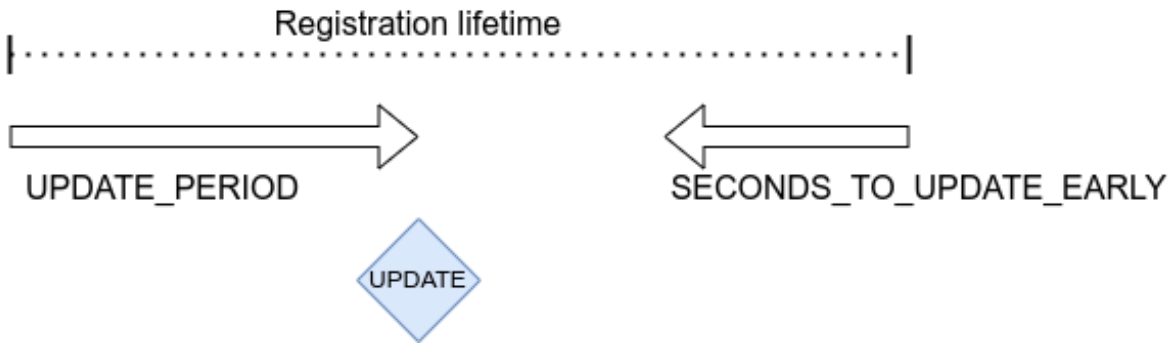


Fig. 8: Update time is controlled by UPDATE_PERIOD.

When also the `CONFIG_LWM2M_UPDATE_PERIOD` is set, time to send the update message is the earliest when any of these values expire. This allows setting long lifetime for the registration and configure the period accurately, even if server changes the lifetime parameter.

In runtime, the update frequency is limited to once in 15 seconds to avoid flooding.

LwM2M shell For testing the client it is possible to enable Zephyr's shell and LwM2M specific commands which support changing the state of the client. Operations supported are read, write and execute resources. Client start, stop, pause and resume are also available. The feature is enabled by selecting `CONFIG_LWM2M_SHELL`. The shell is meant for testing so productions systems should not enable it.

One imaginable scenario, where to use the shell, would be executing client side actions over UART when a server side tests would require those. It is assumed that not all tests are able to trigger required actions from the server side.

```
uart:~$ lwm2m
lwm2m - LwM2M commands
Subcommands:
  send    :send PATHS
          LwM2M SEND operation

  exec    :exec PATH [PARAM]
          Execute a resource

  read    :read PATH [OPTIONS]
          Read value from LwM2M resource
```

(continues on next page)

(continued from previous page)

```
        -x   Read value as hex stream (default)
        -s   Read value as string
        -b   Read value as bool (1/0)
        -uX  Read value as uintX_t
        -sX  Read value as intX_t
        -f   Read value as float
        -t   Read value as time_t

write   :write PATH [OPTIONS] VALUE
        Write into LwM2M resource
        -s   Write value as string (default)
        -b   Write value as bool
        -uX  Write value as uintX_t
        -sX  Write value as intX_t
        -f   Write value as float
        -t   Write value as time_t

create  :create PATH
        Create object or resource instance

delete  :delete PATH
        Delete object or resource instance

cache   :cache PATH NUM
        Enable data cache for resource
        PATH is LwM2M path
        NUM how many elements to cache

start   :start EP_NAME [BOOTSTRAP FLAG]
        Start the LwM2M RD (Registration / Discovery) Client
        -b   Set the bootstrap flag (default 0)

stop    :stop [OPTIONS]
        Stop the LwM2M RD (De-register) Client
        -f   Force close the connection

update  :Trigger Registration Update of the LwM2M RD Client

pause   :LwM2M engine thread pause
resume  :LwM2M engine thread resume
lock    :Lock the LwM2M registry
unlock  :Unlock the LwM2M registry
obs     : List observations
ls      : ls [PATH]
        List objects, instances, resources
```

API Reference

LwM2M high-level API

Related code samples

- LwM2M clientImplement a LwM2M client that connects to a LwM2M server.

MCP Server

- *Overview*
- *Architecture*
  - *HTTP Transport and Asynchronous Responses*
- *Configuration*
- *Usage*
  - *Server Setup*
  - *Tool Registration*
  - *Tool Callback Patterns*
  - *Tool Removal*
- *Limitations*
- *Testing*
- *Sample*
- *API Reference*

Overview The Model Context Protocol (MCP) is an open standard for connecting AI applications to external data sources and tools. It defines a JSON-RPC-based communication protocol between MCP clients (AI agents) and MCP servers (capability providers).

The Zephyr MCP Server library implements the server role of the MCP specification (version 2025-11-25). It allows networked Zephyr devices to expose tools that AI agents can discover and invoke over HTTP. The library builds on existing Zephyr subsystems: the HTTP server library for transport and the JSON library for serialization.

> **Note**
>
> This library is marked *experimental*. Only the tool service with text responses is implemented. Additional features, like more content types, session management, SSE streaming, authorization and other services are planned for future releases.

Architecture

Application
Registers tools and implements their callbacks. Tool execution results are submitted back to the core via `mcp_server_submit_tool_message()`.

MCP Server Core
Implements the MCP protocol state machines. Manages client connections, a tool registry, execution tracking, and a configurable worker thread pool. A health monitor thread enforces timeouts and triggers cancellations.

JSON Processing
Serializes outgoing responses and deserializes incoming JSON-RPC requests using the Zephyr JSON library.

Transport Layer
Abstracts the network protocol. The included HTTP transport uses the Zephyr HTTP server library. A mock transport is available for unit testing.

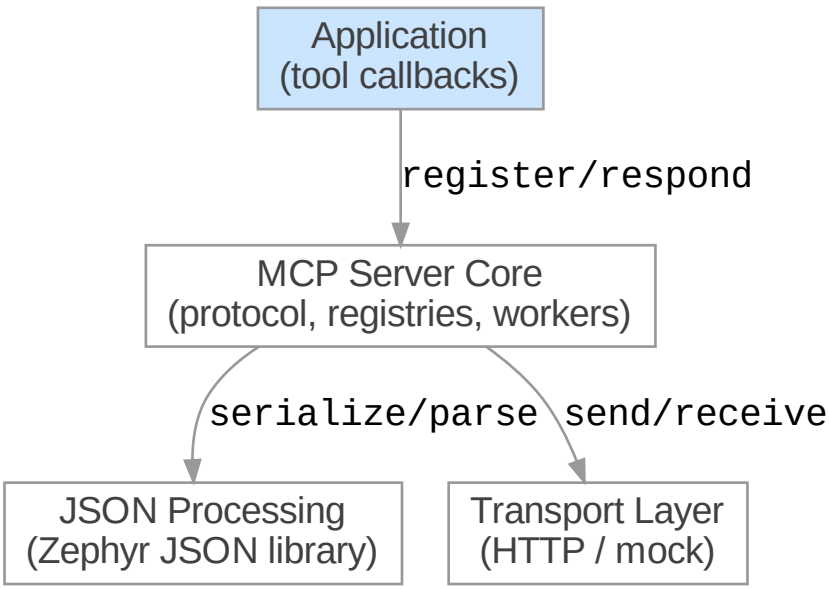


Fig. 9: MCP Server layered architecture

HTTP Transport and Asynchronous Responses Zephyr's HTTP server runs in a single thread and processes one request at a time across all connections. A resource callback must return before the server can handle the next request from any client. Because tool execution may take longer than acceptable for blocking the entire server, the MCP HTTP transport implements a polling-to-SSE fallback:

1. On a POST request, the transport polls internally for a tool response up to `CONFIG_MCP_HTTP_TIMEOUT_MS` (checked every `CONFIG_MCP_HTTP_POLL_INTERVAL_MS`).
2. If a response is ready within this window, it is returned directly as `application/json`.
3. If the timeout expires, the transport switches to SSE mode: it returns a `text/event-stream` response containing only an event ID (no data). This signals the client that the response is pending and it should start polling with periodic GET requests (interval controlled by `CONFIG_MCP_HTTP_SSE_RETRY_MS`).
4. The client then issues the periodic GET requests with a `Last-Event-Id` header. If a result is ready, the server sends the response and ends the SSE stream. If the result is not ready, the server sends a retry response again.

> **Note**
>
> This is not full SSE streaming. The mechanism only delivers deferred responses for requests that could not complete within the initial HTTP timeout. Server-initiated notifications and streaming tool output are not supported in this phase.

Configuration Enable the library with `CONFIG_MCP_SERVER`. The transport is selected through `CONFIG_MCP_TRANSPORT_HTTP` (default) or `CONFIG_MCP_TRANSPORT_MOCK` (testing only).

Minimal `prj.conf`:

```
CONFIG_NETWORKING=y
CONFIG_NET_TCP=y
CONFIG_HTTP_SERVER=y
CONFIG_MCP_SERVER=y
CONFIG_MCP_TRANSPORT_HTTP=y
```

Key configuration groups:

Scalability
: CONFIG_MCP_MAX_CLIENTS, CONFIG_MCP_MAX_CLIENT_REQUESTS, CONFIG_MCP_MAX_TOOLS, CONFIG_MCP_REQUEST_WORKERS

Timeouts
: CONFIG_MCP_TOOL_EXEC_TIMEOUT_MS, CONFIG_MCP_TOOL_IDLE_TIMEOUT_MS, CONFIG_MCP_TOOL_CANCEL_TIMEOUT_MS, CONFIG_MCP_CLIENT_TIMEOUT_MS CONFIG_MCP_HTTP_TIMEOUT_MS

Memory allocation
: CONFIG_MCP_ALLOC_SLAB (default) provides deterministic, fragmentation-free allocation using pre-allocated slabs. CONFIG_MCP_ALLOC_HEAP uses k_malloc/k_free for on-demand allocation at the cost of possible fragmentation.

Usage

Server Setup

```c
#include <zephyr/net/mcp/mcp_server.h>
#include <zephyr/net/mcp/mcp_server_http.h>

static mcp_server_ctx_t server;

int main(void)
{
    server = mcp_server_init();
    mcp_server_http_init(server);

    /* Register tools here */

    mcp_server_start(server);
    mcp_server_http_start(server);
    return 0;
}
```

Tool Registration

```c
static int my_tool_cb(enum mcp_tool_event_type event,
                      const char *arguments,
                      const char *execution_token)
{
    if (event == MCP_TOOL_CANCEL_REQUEST) {
        struct mcp_tool_message ack = {
            .type = MCP_USR_TOOL_CANCEL_ACK,
        };

        mcp_server_submit_tool_message(server, &ack, execution_token);

        /* Handle cancellation here */
    }
```

(continues on next page)

(continued from previous page)

```
    struct mcp_tool_message resp = {
        .type = MCP_USR_TOOL_RESPONSE,
        .data = "Tool execution result",
        .length = strlen("Tool execution result"),
        .is_error = false,
    };
    return mcp_server_submit_tool_message(server, &resp, execution_token);
}

static const struct mcp_tool_record my_tool = {
    .metadata = {
        .name = "my_tool",
        .input_schema = "{\"type\":\"object\",\"properties\":{}}",
    },
    .callback = my_tool_cb,
};

mcp_server_add_tool(server, &my_tool);
```

The `.data` field accepts a plain text string. The server wraps it into an MCP-compliant `"text"` content item automatically. Maximum length is `CONFIG_MCP_TOOL_RESULT_MAX_LEN`.

Tool Callback Patterns

Blocking
: Short-running tools execute directly in the worker thread and call `mcp_server_submit_tool_message()` before returning. The worker stack size is `CONFIG_MCP_REQUEST_WORKER_STACK_SIZE`.

Asynchronous
: Long-running tools should spawn a dedicated thread, return immediately from the callback, and submit the response later using the provided execution token. Periodic pings (`MCP_USR_TOOL_PING`) prevent the health monitor from cancelling idle executions.

Cancellation
: When the health monitor or a client requests cancellation, the callback is invoked with `MCP_TOOL_CANCEL_REQUEST`. The tool should stop work and submit `MCP_USR_TOOL_CANCEL_ACK`.

Tool Removal Tools can be removed at runtime with `mcp_server_remove_tool()`. The call returns `-EBUSY` if the tool is currently executing; retry later.

Limitations The following MCP features are not yet implemented:

- Resources, prompts, sampling, roots, and session management
- Server-initiated notifications and streaming tool output
- Image and embedded-resource content types (only `"text"` is supported)
- Full SSE transport (only deferred response delivery is supported)

Testing Unit tests are available under tests/net/lib/mcp/. They use the mock transport (`CONFIG_MCP_TRANSPORT_MOCK`) to exercise protocol logic without a network stack.

Sample See mcp-server-hello-world for a working example that registers multiple tools including GPIO-based LED control.

API Reference
MCP Server API

Related code samples

- MCP Server Hello WorldThis code sample shows two very basic tool definitions to introduce users to the Zephyr MCP Server library.

MQTT

- *Overview*
- *Sample usage*
- *Using MQTT with TLS*
- *API Reference*

Overview MQTT (Message Queuing Telemetry Transport) is an application layer protocol which works on top of the TCP/IP stack. It is a lightweight publish/subscribe messaging transport for machine-to-machine communication. For more information about the protocol itself, see https://mqtt.org/.

Zephyr provides an MQTT client library built on top of BSD sockets API. The library can be enabled with `CONFIG_MQTT_LIB` Kconfig option and is configurable at a per-client basis, with support for MQTT versions 3.1.0, 3.1.1 and 5.0. The Zephyr MQTT implementation can be used with either plain sockets communicating over TCP, or with secure sockets communicating over TLS. See *BSD Sockets* for more information about Zephyr sockets.

MQTT clients require an MQTT server to connect to. Such a server, called an MQTT Broker, is responsible for managing client subscriptions and distributing messages published by clients. There are many implementations of MQTT brokers, one of them being Eclipse Mosquitto. See https://mosquitto.org/ for more information about the Eclipse Mosquitto project.

Sample usage To create an MQTT client, a client context structure and buffers need to be defined:

```
/* Buffers for MQTT client. */
static uint8_t rx_buffer[256];
static uint8_t tx_buffer[256];

/* MQTT client context */
static struct mqtt_client client_ctx;
```

Multiple MQTT client instances can be created in the application and managed independently. Additionally, a structure for MQTT Broker address information is needed. This structure must be accessible throughout the lifespan of the MQTT client and can be shared among MQTT clients:

```
/* MQTT Broker address information. */
static struct net_sockaddr_storage broker;
```

An MQTT client library will notify MQTT events to the application through a callback function created to handle respective events:

```
void mqtt_evt_handler(struct mqtt_client *client,
                      const struct mqtt_evt *evt)
```

(continues on next page)

(continued from previous page)

```
{
   switch (evt->type) {
      /* Handle events here. */
   }
}
```

For a list of possible events, see *API Reference*.

The client context structure needs to be initialized and set up before it can be used. An example configuration for TCP transport is shown below:

```
mqtt_client_init(&client_ctx);

/* MQTT client configuration */
client_ctx.broker = &broker;
client_ctx.evt_cb = mqtt_evt_handler;
client_ctx.client_id.utf8 = (uint8_t *)"zephyr_mqtt_client";
client_ctx.client_id.size = sizeof("zephyr_mqtt_client") - 1;
client_ctx.password = NULL;
client_ctx.user_name = NULL;
client_ctx.protocol_version = MQTT_VERSION_3_1_1;
client_ctx.transport.type = MQTT_TRANSPORT_NON_SECURE;

/* MQTT buffers configuration */
client_ctx.rx_buf = rx_buffer;
client_ctx.rx_buf_size = sizeof(rx_buffer);
client_ctx.tx_buf = tx_buffer;
client_ctx.tx_buf_size = sizeof(tx_buffer);
```

After the configuration is set up, the MQTT client can connect to the MQTT broker. Call the `mqtt_connect` function, which will create the appropriate socket, establish a TCP/TLS connection, and send an `MQTT CONNECT` message. When notified, the application should call the `mqtt_input` function to process the response received. Note, that `mqtt_input` is a non-blocking function, therefore the application should use socket `poll` to wait for the response. If the connection was successful, `MQTT_EVT_CONNACK` will be notified to the application through the callback function.

```
rc = mqtt_connect(&client_ctx);
if (rc != 0) {
   return rc;
}

fds[0].fd = client_ctx.transport.tcp.sock;
fds[0].events = ZSOCK_POLLIN;
poll(fds, 1, 5000);

mqtt_input(&client_ctx);

if (!connected) {
   mqtt_abort(&client_ctx);
}
```

In the above code snippet, the MQTT callback function should set the `connected` flag upon a successful connection. If the connection fails at the MQTT level or a timeout occurs, the connection will be aborted, and the underlying socket closed.

After the connection is established, an application needs to call `mqtt_input` and `mqtt_live` functions periodically to process incoming data and upkeep the connection. If an MQTT message is received, an MQTT callback function will be called and an appropriate event notified.

The connection can be closed by calling the `mqtt_disconnect` function.

Zephyr provides sample code utilizing the MQTT client API. See mqtt-publisher for more information.

Using MQTT with TLS The Zephyr MQTT library can be used with TLS transport for secure communication by selecting a secure transport type (`MQTT_TRANSPORT_SECURE`) and some additional configuration information:

```
client_ctx.transport.type = MQTT_TRANSPORT_SECURE;

struct mqtt_sec_config *tls_config = &client_ctx.transport.tls.config;

tls_config->peer_verify = TLS_PEER_VERIFY_REQUIRED;
tls_config->cipher_list = NULL;
tls_config->sec_tag_list = m_sec_tags;
tls_config->sec_tag_count = ARRAY_SIZE(m_sec_tags);
tls_config->hostname = MQTT_BROKER_HOSTNAME;
tls_config->set_native_tls = true;
```

In this sample code, the `m_sec_tags` array holds a list of tags, referencing TLS credentials that the MQTT library should use for authentication. We do not specify `cipher_list`, to allow the use of all cipher suites available in the system. We set `hostname` field to broker hostname, which is required for server authentication. Finally, we enforce peer certificate verification by setting the `peer_verify` field.

Note, that TLS credentials referenced by the `m_sec_tags` array must be registered in the system first. For more information on how to do that, refer to *secure sockets documentation*.

Finally, `set_native_tls` can be optionally set to enable native TLS support instead of offloading TLS operations to an offloaded socket.

An example of how to use TLS with MQTT is also present in mqtt-publisher sample application.

API Reference

MQTT Client library

Related code samples

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- Microsoft Azure IoT Hub MQTTConnect to Azure IoT Hub and publish messages using MQTT.
- MQTT publisherSend MQTT PUBLISH messages to an MQTT server.
- Secure MQTT Sensor/ActuatorImplement an MQTT-based IoT sensor/actuator device

MQTT-SN

- *Overview*
- *Sample usage*
- *Deviations from the standard*
- *API Reference*

Overview MQTT-SN is a variant of the well-known MQTT protocol - see *MQTT*.

In contrast to MQTT, MQTT-SN does not require a TCP transport, but is designed to be used over any message-based transport. Originally, it was mainly created with ZigBee in mind, but others like Bluetooth, UDP or even a UART can be used just as well.

Zephyr provides an MQTT-SN client library built on top of BSD sockets API. The library can be enabled with `CONFIG_MQTT_SN_LIB` Kconfig option and is configurable at a per-client basis, with support for MQTT-SN version 1.2. The Zephyr MQTT-SN implementation can be used with any message-based transport, but support for UDP is already built-in.

MQTT-SN clients require an MQTT-SN gateway to connect to. These gateways translate between MQTT-SN and MQTT. The Eclipse Paho project offers an implementation of a MQTT-SN gateway, but others are available too. https://www.eclipse.org/paho/index.php?page=components/mqtt-sn-transparent-gateway/index.php

The MQTT-SN spec v1.2 can be found here: https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf

Sample usage To create an MQTT-SN client, a client context structure and buffers need to be defined:

```
/* Buffers for MQTT client. */
static uint8_t rx_buffer[256];
static uint8_t tx_buffer[256];

/* MQTT-SN client context */
static struct mqtt_sn_client client;
```

Multiple MQTT-SN client instances can be created in the application and managed independently. Additionally, a structure for the transport is needed as well. The library already comes with an example implementation for UDP.

```
/* MQTT Broker address information. */
static struct mqtt_sn_transport tp;
```

The MQTT-SN library will inform clients about certain events using a callback.

```
static void evt_cb(struct mqtt_sn_client *client,
                   const struct mqtt_sn_evt *evt)
{
   switch(evt->type) {
   {
      /* Handle events here. */
   }
}
```

For a list of possible events, see *API Reference*.

The client context structure needs to be initialized and set up before it can be used. An example configuration for UDP transport is shown below:

```
struct mqtt_sn_data client_id = MQTT_SN_DATA_STRING_LITERAL("ZEPHYR");
struct net_sockaddr_in gateway = {0};

uint8_t tx_buf[256];
uint8_t rx_buf[256];

mqtt_sn_transport_udp_init(&tp, (struct net_sockaddr*)&gateway, sizeof((gateway)));
```

(continues on next page)

(continued from previous page)

```
mqtt_sn_client_init(&client, &client_id, &tp.tp, evt_cb, tx_buf, sizeof(tx_buf),␣
↪rx_buf, sizeof(rx_buf));
```

After the configuration is set up, the network address for the gateway to connect to must be defined. The MQTT-SN protocol offers functionality to discover gateways through an advertisement or a search mechanism. A user should do at least one of the following steps to define a Gateway for the library:

- Call the `mqtt_sn_add_gw()` function to manually define a Gateway address.
- Wait for an `MQTT_SN_EVT_ADVERTISE`.
- Call the `mqtt_sn_search()` function and wait for an `MQTT_SN_EVT_GWINFO` callback. Make sure to call the `mqtt_sn_input()` function periodically to process incoming messages.

Example `mqtt_sn_search()` function call:

```
err = mqtt_sn_search(&mqtt_client, 1);
k_sleep(K_SECONDS(10));
err = mqtt_sn_input(&mqtt_client);
__ASSERT(err == 0, "mqtt_sn_search() failed %d", err);
```

After the Gateway address has been defined or found, the MQTT-SN client can connect to the gateway. Call the `mqtt_sn_connect()` function, which will send a `CONNECT` MQTT-SN message. The application should periodically call the `mqtt_sn_input()` function to process the response received. The application does not have to call `mqtt_sn_input()` if it knows that no data has been received (e.g. when using Bluetooth). Note that `mqtt_sn_input()` is a non-blocking function, if the transport struct contains a `poll()` compatible function pointer. If the connection was successful, `MQTT_SN_EVT_CONNECTED` will be notified to the application through the callback function.

```
err = mqtt_sn_connect(&client, false, true);
__ASSERT(err == 0, "mqtt_sn_connect() failed %d", err);

while (1) {
        mqtt_sn_input(&client);
        if (connected) {
                mqtt_sn_publish(&client, MQTT_SN_QOS_0, &topic_p, false, &pubdata);
        }
        k_sleep(K_MSEC(500));
}
```

In the above code snippet, the gateway is connected to before publishing messages. If the connection fails at the MQTT level or a timeout occurs, the connection will be aborted and an error returned.

After the connection is established, an application needs to call `mqtt_input()` function periodically to process incoming data. Connection upkeep, on the other hand, is done automatically using a k_work item. If a MQTT message is received, an MQTT callback function will be called and an appropriate event notified.

The connection can be closed by calling the `mqtt_sn_disconnect()` function. This has no effect on the transport, however. If you want to close the transport (e.g. the socket), call `mqtt_sn_client_deinit()`, which will deinit the transport as well.

Zephyr provides sample code utilizing the MQTT-SN client API. See mqtt-sn-publisher for more information.

Deviations from the standard Certain parts of the protocol are not yet supported in the library.

- Forwarder Encapsulation

API Reference
MQTT-SN Client library

Related code samples

- MQTT-SN publisherSend MQTT-SN PUBLISH messages to an MQTT-SN gateway.

Open Charge Point Protocol (OCPP)

- *Overview*
- *Sample usage*
- *API Reference*

Overview Open Charge Point Protocol (OCPP) is an application protocol for communication between Charge Points (Electric vehicle (EV) charging stations) and a central management system, also known as a charging station network. OCPP is a standard defined by The Open Charge Alliance and goal is to offer a uniform solution for the method of communication between charge point and central system. With this protocol it is possible to connect any central system with any charge point, regardless of the vendor.

Zephyr provides an OCPP Charge Point (CP) library built on top of websocket API with payload in json format. The library can be enabled with `CONFIG_OCPP` Kconfig option. Currently OCPP 1.6 with basic core profile is supported.

OCPP charge point (CP) require a Central System (CS) server to connect, an open source SteVe server shall be setup locally for development purpose, See SteVe server for more information about the setup.

The Zephyr OCPP CP library implements the following items:

- engine to process socket connectivity and events
- OCPP core functions to frame/parse payload, user notification for OCPP events, heartbeat notification

Sample usage Init ocpp library with overall CP and CS information. Prior to init an OCPP library, a network interface should be ready using ethernet or wifi or modem. A filled CP, CS structure and user callback needs to be passed in ocpp_init.

```
static int user_notify_cb(ocpp_notify_reason_t reason,
                          ocpp_io_value_t *io,
                          void *user_data)
{

     switch (reason) {
     case OCPP_USR_GET_METER_VALUE:
             ...
             break;

     case OCPP_USR_START_CHARGING:
             ...
             break;

             ...
```

(continues on next page)

(continued from previous page)

```
            ...
}

/* OCPP configuration */
ocpp_cp_info_t cpi = { "basic", "zephyr", .num_of_con = 1};
ocpp_cs_info_t csi =  {"192.168.1.3",   /* ip address */
                       "/steve/websocket/CentralSystemService/zephyr",
                       8180,
                       AF_INET};

ret = ocpp_init(&cpi, &csi, user_notify_cb, NULL);
```

A unique session must open for each physical connector before any ocpp transaction API call.

```
ocpp_session_handle_t sh = NULL;
ret = ocpp_session_open(&sh);
```

idtag is EV user's authentication token which should match with list on CS. Authorize request must call to ensure validity of idtag (if charging request originate from local CP) before start energy transfer to EV.

```
ocpp_auth_status_t status;
ret = ocpp_authorize(sh, idtag, &status, 500);
```

On successful, authorization status is available in status.

Apart from local CP, charging request may originate from CS is notified to user in callback with OCPP_USR_START_CHARGING, here authorize request call is optional. When the CS is ready to provide power to EV, a start transaction is notified to CS with meter reading and connector id using ocpp_start_transaction.

```
const int idcon = 1;
const int mval = 25; //meter reading in wh
ret = ocpp_start_transaction(sh, mval, idcon, 200);
```

Once the start transaction is success, user callback is invoked to get meter readings from the library. callback should be not be hold for longer time.

API Reference

OCPP library

Related code samples

- OCPP charge pointImplement an OCPP charge point that connects to a Central System server and simulates the meter readings.

RTP

- *Overview*
- *Samples*
- *API Reference*

Overview The Real-time Transport Protocol (RTP, **RFC 3550**) carries media streams such as audio over UDP. Zephyr implements the RTP data plane under `CONFIG_RTP`: session management, packet serialization and deserialization, and two transport backends. RTCP is not implemented.

An application declares a session with `RTP_SESSION_DEFINE`, configures it with `rtp_session_init()` as a source, sink, or both, and starts it with `rtp_session_start()`. Payloads are transmitted with `rtp_session_send()`; received packets are delivered through a callback invoked from the network receive context.

Two transports are available:

- **Socket** (`CONFIG_RTP_TRANSPORT_SOCKET`): BSD sockets with a background socket service for polling. The default.
- **net_pkt** (`CONFIG_RTP_TRANSPORT_NET_PKT`): builds raw `net_pkt` buffers, bypassing the socket layer for minimal overhead.

Note

RTP support is **experimental**: APIs and Kconfig options may change.

Samples

- net-rtp

API Reference

RTP (Real-time Transport Protocol)

Related code samples

- RTPSend or receive an RTP audio stream over a UDP multicast.

TFTP Zephyr provides a simple TFTP client library that can enabled with `CONFIG_TFTP_LIB` Kconfig option.

See TFTP client sample application for more information about the library usage.

API Reference

TFTP Client library

Related code samples

- TFTP clientUse the TFTP client library to get/put files from/to a TFTP server.

Latmon Network Service

- *Overview*
- *API Reference*
- *Features*
- *Workflow*
  - *Socket Creation*

- *Connection Handling*
- *Monitoring Start*
- *Monitoring Status*
- *Thread Management*

- *Enabling the Latmon Service*
- *Example Usage*

Overview Provides the functionality required for network-based latency monitoring, including socket management, client-server communication, and data exchange with the Latmus service running on the System Under Test (SUT).

The Latmon network service is responsible for establishing and managing network communication between the Latmon application (running on a Zephyr-based board) and the Latmus service (running on the SUT).

It uses TCP sockets for reliable communication and UDP sockets for broadcasting the IP address of the Latmon device.

API Reference

Latency Monitor

Related code samples

- Latmon ClientMeasures delta time between GPIO events and reports the latency metrics via latmon to the latmus service executing on the SUT.

Features

- **Socket Management**: Creates and manages TCP and UDP sockets for communication.
- **Client-Server Communication**: Handles incoming connections from the Latmus service.
- **Data Exchange**: Sends latency metrics and histogram data to the Latmus service.
- **IP Address Broadcasting**: Broadcasts the IP address of the Latmon device to facilitate discovery by the Latmus service.
- **Thread-Safe Design**: Uses Zephyr's kernel primitives (e.g., message queues and semaphores) for synchronization.

Workflow

Socket Creation The `net_latmon_get_socket()` function is called to create and configure a TCP socket to communicate with the Latmus service. A connection address can be specified as a parameter to bind the socket to a specific interface and port.

Connection Handling The `net_latmon_connect()` function waits for a connection from the Latmus service. If no connection is received within the timeout period, the service broadcasts its IP address using UDP and returns `-EAGAIN`. If the broadcast request cannot be sent, the function returns `-1`, and the client should quit.

Monitoring Start Once a connection is established, the `net_latmon_start()` function is called to start the monitoring process. This function uses a callback to calculate latency deltas and sends the data to the Latmus service.

Monitoring Status The `net_latmon_running()` function can be used to check if the monitoring process is active.

Thread Management The service uses Zephyr threads to handle incoming connections and manage the monitoring process.

Enabling the Latmon Service The following configuration option must be enabled in the `prj.conf` file.

- `CONFIG_NET_LATMON`

The following options may be configured to customize the Latmon service:

- `CONFIG_NET_LATMON_PORT` - Port number for the Latmon service.
- `CONFIG_NET_LATMON_XFER_THREAD_STACK_SIZE`
- `CONFIG_NET_LATMON_XFER_THREAD_PRIORITY`
- `CONFIG_NET_LATMON_THREAD_STACK_SIZE`
- `CONFIG_NET_LATMON_THREAD_PRIORITY`
- `CONFIG_NET_LATMON_MONITOR_THREAD_STACK_SIZE`
- `CONFIG_NET_LATMON_MONITOR_THREAD_PRIORITY`

Example Usage

```
#include <zephyr/net/latmon.h>
#include <zephyr/net/socket.h>

void main(void)
{
    struct in_addr ip;
    int server_socket, client_socket;

    /* Create and configure the server socket */
    server_socket = net_latmon_get_socket(NULL);

    while (1) {
        /* Wait for a connection from the Latmus service */
        client_socket = net_latmon_connect(server_socket, &ip);
        if (client_socket < 0) {
            if (client_socket == -EAGAIN) {
                continue;
            }
            goto out;
        }

        /* Start the latency monitoring process */
        net_latmon_start(client_socket, measure_latency_cycles);
    }
out:
    close(server_socket);
}
```

QUIC Transport Interface

- *Overview*
- *Architecture & Concepts*
- *Application Workflow*
  - *Client Application*
  - *Server Application*
- *TLS & Security Configuration*
  - *Managing Certificates*
  - *Applying Credentials to QUIC*
  - *Applying ALPN to QUIC*
- *Session Resumption and 0-RTT*
- *Path MTU Discovery (DPLPMTUD)*
  - *Behavior*
  - *Interaction with loss recovery*
- *Configure Options*
  - *Connection and Stream Counts*
  - *QUIC Transport Parameters (Flow Control)*
  - *Buffer Sizes and RAM Allocation*
  - *Timeout Options*
  - *Service Thread Options*
- *Memory Model*
- *Kconfig Optimizer*
  - *Prerequisites*
  - *Usage: Interactive Mode*
  - *Usage: Command-Line Mode*
  - *Input Parameters*
  - *Output Sections*
  - *Example Output*
  - *Minimal IoT sensor (client, IPv4 only, tight RAM budget)*
  - *Dual-stack server (IPv4 + IPv6, multiple clients)*
  - *High-throughput WAN link with packet loss*
- *Tuning Guidelines*
- *API Reference*

Overview QUIC is a general-purpose, multiplexed transport protocol standardised in **RFC 9000**. It operates over UDP and provides ordered, reliable byte-stream delivery with integrated TLS 1.3 security (**RFC 9001**), stream multiplexing, and connection migration. Zephyr's QUIC implementation is available under `CONFIG_QUIC`.

Key properties relevant to embedded use:

- **Stream multiplexing**: Bidirectional and unidirectional streams share a single UDP socket, avoiding head-of-line blocking at the transport layer.
- **Integrated TLS 1.3**: The handshake is built into the connection establishment; no separate TLS layer is required.
- **Flow control**: Per-stream and per-connection credit-based flow control prevents fast senders from overwhelming constrained receivers.
- **Loss recovery**: A Probe Timeout (PTO) mechanism retransmits data without relying on ICMP or TCP-style ACK clocks.
- **Path MTU discovery**: Datagram Packetization Layer PMTU Discovery (DPLPMTUD, RFC 9000 Section 14.3) probes the path after the handshake and raises the send size only after probes are acknowledged.
- **Socket Integration**: Uses standard Zephyr socket calls like `zsock_send`, `zsock_recv`, `zsock_recvmsg`, `zsock_sendmsg`, `zsock_close` for data transfer.
- **Dual stack**: Supports both IPv4 and IPv6 connections.

> **Note**
>
> QUIC support is currently **experimental** in Zephyr. Enable it with `CONFIG_QUIC=y` and be aware that APIs and Kconfig options may change between releases.

Architecture & Concepts To effectively use the library, it is helpful to understand the relationship between **Connections** and **Streams**.

- **QUIC Connection Socket**: Represents the "tunnel" to the peer. It handles the TLS handshake, congestion control, and connection termination. You typically do not send application data directly over this socket; you use it to create streams.
- **QUIC Stream Socket**: Represents a lightweight data channel inside the connection. This is where the actual application data (`send` or `recv`) flows.

Application Workflow The following sections outline how to write a generic Client and Server application using the library.

Client Application A client initiates a connection to a remote server, opens a stream, sends a request, and reads the response.

Step 1: Setup Addresses

```
struct net_sockaddr_in remote_addr = {
    .sin_family = NET_AF_INET,
    .sin_port = net_htons(4422),
};

zsock_inet_pton(NET_AF_INET, "192.0.2.1", &remote_addr.sin_addr);
```

Step 2: Open QUIC Connection

```
int conn_sock = quic_connection_open((struct net_sockaddr *)&remote_addr, NULL);
if (conn_sock < 0) {
    /* Handle error */
}

/* The TLS handshake occurs automatically upon the first stream creation. */
```

Step 3: Open a Stream

```
/* Create a bidirectional stream initiated by the client */
int stream_sock = quic_stream_open(conn_sock,
                                   QUIC_STREAM_CLIENT,
                                   QUIC_STREAM_BIDIRECTIONAL,
                                   0);
```

Step 4: Transfer Data

Use standard socket calls on the **stream socket**, not the connection socket.

```
zsock_send(stream_sock, "Hello Server", 12, 0);

char buffer[64];
zsock_recv(stream_sock, buffer, sizeof(buffer), 0);
```

Step 5: Cleanup

```
quic_stream_close(stream_sock);
quic_connection_close(conn_sock);
```

Server Application A server binds to a local port, waits for incoming connections, accepts streams, and processes data.

Step 1: Bind Connection Socket

```
struct net_sockaddr_in local_addr = {
   .sin_family = NET_AF_INET,
   .sin_port = net_htons(4422),
   .sin_addr.s_addr = NET_INADDR_ANY,
};

/* Create the listening context */
int conn_sock = quic_connection_open(NULL, (struct net_sockaddr *)&local_addr);
```

Step 2: Accept Incoming Streams

The `quic_connection_open()` socket acts as a "parent". Use `zsock_accept()` on the **connection socket** to retrieve a file descriptor for a new stream initiated by a peer.

```
struct net_sockaddr_in peer_addr;
net_socklen_t addrlen = sizeof(peer_addr);

/* Block until a client opens a stream */
int stream_sock = zsock_accept(conn_sock, (struct net_sockaddr *)&peer_addr, &
↪addrlen);

if (stream_sock >= 0) {
    /* Handle the new stream (read/write data) */
    char buf[128];
    int len = zsock_recv(stream_sock, buf, sizeof(buf), 0);

    /* Echo back */
    zsock_send(stream_sock, buf, len, 0);

    /* Close the stream when done, also zsock_close() can be used */
    quic_stream_close(stream_sock);
}
```

TLS & Security Configuration The QUIC transport uses Mbed TLS and PSA APIs for cryptographic operations. Security credentials (certificates, keys) are managed via the Zephyr **TLS Credentials** subsystem.

Managing Certificates Before opening a connection, you must register your certificates using `tls_credential_add()`.

1. **CA Certificate**: Required for clients to verify servers.
2. **Server Certificate & Private Key**: Required for servers.

Example: Loading Credentials

```
#include <zephyr/net/tls_credentials.h>

/* Tag ID to reference credentials later */
#define MY_SEC_TAG 1

static const char server_cert[] = ...; /* PEM or DER data */
static const char priv_key[] = ...;    /* PEM or DER data */

void setup_credentials(void) {
    tls_credential_add(MY_SEC_TAG, TLS_CREDENTIAL_PUBLIC_CERTIFICATE,
                       server_cert, sizeof(server_cert));
    tls_credential_add(MY_SEC_TAG, TLS_CREDENTIAL_PRIVATE_KEY,
                       priv_key, sizeof(priv_key));
}
```

Applying Credentials to QUIC You apply credentials to the QUIC socket using `zsock_setsockopt()` on the **connection socket** immediately after creation. The credentials must be set before the stream is created.

Peer certificate verification follows the same default policy as Zephyr TLS sockets: clients require successful peer verification by default, while servers default to not verifying client certificates unless `ZSOCK_TLS_PEER_VERIFY` is explicitly enabled. A client that does not load a CA certificate therefore fails the handshake by default; applications that deliberately skip server authentication must opt out with `ZSOCK_TLS_PEER_VERIFY = MBEDTLS_SSL_VERIFY_NONE`.

```
sec_tag_t sec_tag_list[] = { MY_SEC_TAG };

zsock_setsockopt(conn_sock, ZSOCK_SOL_TLS, ZSOCK_TLS_SEC_TAG_LIST,
                 sec_tag_list, sizeof(sec_tag_list));
```

Applying ALPN to QUIC The Application-Layer Protocol Negotiation (ALPN) is mandatory in QUIC. The ALPN is used to negotiate the application protocol that is run on top of two QUIC endpoints. You apply ALPN list to the QUIC socket using `zsock_setsockopt()` on the **connection socket** immediately after creation. The ALPN list must be set before the stream is created. Note that the list items must be constants, and they cannot be variables.

```
const char * const alpn_list[] = {
    "echo-quic",
    NULL
};

ret = zsock_setsockopt(quic_sock, ZSOCK_SOL_TLS, ZSOCK_TLS_ALPN_LIST,
                       alpn_list, sizeof(alpn_list));
if (ret < 0) {
```

(continues on next page)

(continued from previous page)

```
    LOG_ERR("Failed to set ALPN (%d)", -errno);
}
```

Session Resumption and 0-RTT After a full handshake a server may issue a TLS 1.3 *NewSessionTicket*. A client can store the resulting session state and present it on a later connection to **resume** without a full handshake, and – when the server permits it – to send **0-RTT early data** in the very first flight, before the handshake completes.

0-RTT support is compiled in with `CONFIG_QUIC_0RTT` (**disabled by default**). Session resumption at 1-RTT works without it; enabling it adds the early-data offer/accept/replay handling.

> **Warning**
>
> 0-RTT early data is **replayable** by a network attacker (RFC 9001 Section 9.2). Only send data for idempotent / replay-safe operations over 0-RTT, and be prepared for the peer to reject it (the data must then be retried after the handshake completes). The stack makes issued tickets single-use and applies an age-freshness window, but this anti-replay state is in-memory and per-instance only (it does not survive a restart and is not shared across server instances).

Client: export and reuse session state

After the handshake, read the resumable state from the connection socket and keep it. On a new connection, import it before opening the first stream. When 0-RTT is armed, the first stream data is carried as early data.

```
struct quic_session_state state;
net_socklen_t optlen = sizeof(state);

/* After the first connection's handshake completes: */
zsock_getsockopt(conn_sock, ZSOCK_SOL_QUIC, ZSOCK_QUIC_SO_SESSION_STATE,
                 &state, &optlen);

/* On a later connection, before the first stream is opened: */
zsock_setsockopt(new_conn_sock, ZSOCK_SOL_QUIC, ZSOCK_QUIC_SO_SESSION_STATE,
                 &state, sizeof(state));
```

Since `struct quic_session_state` carries the resumption secret, treat it as sensitive and store it securely. Its layout is versioned with `QUIC_SESSION_STATE_VERSION`.

Server: issue tickets and permit 0-RTT

Set these options on the listening (or server-side connection) socket before the handshake:

- `ZSOCK_QUIC_SO_SESSION_TICKET_ENABLE` (pointer to `int`) – non-zero makes the server send a NewSessionTicket after the handshake, enabling 1-RTT resumption.
- `ZSOCK_QUIC_SO_MAX_EARLY_DATA_SIZE` (pointer to `uint32_t`) – any non-zero value enables 0-RTT on newly issued tickets (and requires `CONFIG_QUIC_0RTT`, else `setsockopt()` fails with `ENOTSUP`). Per RFC 9001 Section 4.6.1 the ticket always advertises the fixed `0xffffffff` sentinel; the amount of early data a client may actually send is bounded by the connection's flow-control (transport-parameter) limits, so this option is an enable switch, not a byte budget.

```
int enable = 1;
uint32_t early_data = 1; /* non-zero enables 0-RTT for issued tickets */

zsock_setsockopt(quic_sock, ZSOCK_SOL_QUIC, ZSOCK_QUIC_SO_SESSION_TICKET_ENABLE,
                 &enable, sizeof(enable));
```

(continues on next page)

(continued from previous page)

```
zsock_setsockopt(quic_sock, ZSOCK_SOL_QUIC, ZSOCK_QUIC_SO_MAX_EARLY_DATA_SIZE,
                 &early_data, sizeof(early_data));
```

Per-stream early-data status

On a server, query whether a given stream carried accepted 0-RTT data with `ZSOCK_QUIC_SO_STREAM_EARLY_DATA` (pointer to `int`) on the **stream** socket. This lets a protocol layer apply replay protection (for example, HTTP/3 answers a non-idempotent early request with `425 Too Early`).

Path MTU Discovery (DPLPMTUD) Zephyr's QUIC stack performs **Datagram Packetization Layer Path MTU Discovery** (DPLPMTUD) as described in **RFC 9000** Section 14.3. The RFC 8899 search state lives in the generic *Datagram PLPMTUD API* subsystem; QUIC supplies probe packets (PING plus padding), don't-fragment UDP sends, and ACK/loss confirmation through its recovery machinery.

The goal is to find the largest UDP payload size that can traverse the path without IP-layer fragmentation, then use that size for application data packets.

Behavior After the TLS handshake completes, the stack may send **probe datagrams** containing a PING frame plus padding. Probes are sized to exercise larger path MTUs and are transmitted with **don't fragment** enabled on the underlying UDP socket (see `ZSOCK_IP_DONTFRAG` / `ZSOCK_IPV6_DONTFRAG` in *IPv4 and IPv6 socket options*). This is handled internally; applications do not need to configure the UDP socket used by QUIC.

The stack tracks three related limits:

- **Peer "max_udp_payload_size" transport parameter** - upper bound advertised by the remote endpoint during the handshake.
- **Local ceiling** - derived from the interface or socket MTU minus IP and UDP header overhead.
- **Validated send size** - the largest probe size confirmed by an ACK. This is what governs how large outgoing QUIC packets may be.

Send sizing starts at the **1200-byte minimum UDP payload** required by QUIC. When the validated size is below the local and peer ceilings, the stack performs a binary search for a larger working size. A probe ACK raises the validated limit; repeated probe loss narrows the search range. Stream frames are sized to fit within the current validated limit, so throughput on high-MTU paths improves automatically once probing succeeds.

Interaction with loss recovery DPLPMTUD probes are ack-eliciting and participate in the same loss-recovery machinery as stream data. On Probe Timeout (PTO), **in-flight stream frames are retransmitted first**; a DPLPMTUD probe is sent only when there is no stream data to retransmit (the same situation where a bare PING probe would be sent).

There is currently no application-facing QUIC API to enable, disable, or tune DPLPMTUD. Probing begins automatically after handshake completion whenever a larger path MTU may be available. Transports implementing their own UDP protocols can reuse the same generic API documented in *Datagram PLPMTUD API*.

Configure Options All options below live inside the `if QUIC` block and are only visible when `CONFIG_QUIC` is enabled.

Connection and Stream Counts

Option	Default	Description
`CONFIG_QUIC_MAX_CONTEXTS`	1	Maximum number of simultaneous QUIC connections in the system. Each context carries its own TLS state, flow-control windows, and sent-packet history. Memory scales linearly with this value.
`CONFIG_QUIC_MAX_STREAMS_BIDI`	3-5	Maximum bidirectional streams **per connection**. Streams are allocated at build time; unused stream slots still consume their TX/RX buffer allocation.
`CONFIG_QUIC_MAX_STREAMS_UNI`	0	Maximum unidirectional streams per connection. Set to `0` to disable unidirectional streams entirely and recover the associated buffer RAM.
`CONFIG_QUIC_MAX_ENDPOINTS`	2-3	Number of UDP endpoint sockets. A client connecting to a single server over one IP version needs 2; dual-stack server roles need 3. Consider enabling `CONFIG_QUIC_ENDPOINT_USE_IPV4_MAPPING_TO_IPV6` to share a single socket for both IPv4 and IPv6.
`CONFIG_QUIC_MAX_PEER_CIDS`	4	Number of peer Connection IDs tracked per connection. Reduce to 1 on very constrained devices if connection migration is not needed.

QUIC Transport Parameters (Flow Control) These values are advertised to the remote peer during the handshake (see **RFC 9000** Section 18.2). They govern how much data the **peer is permitted to send** before receiving a window update. Each value should match the corresponding local receive buffer size; advertising a window larger than the receive buffer causes buffer overflows, while advertising a window smaller than the buffer under-utilises available memory.

Option	Default	Description
CONFIG_QUIC_INITIAL_MAX_DATA	16384	Connection-level receive credit in bytes. Must be greater than or equal to any single stream's INITIAL_MAX_STREAM_DATA value, otherwise the connection window becomes the bottleneck. A good starting point is streams_bidi x INITIAL_MAX_STREAM_DATA_BIDI_LOCAL.
CONFIG_QUIC_INITIAL_MAX_STREAM_DATA_	16384	Initial receive window for locally initiated bidirectional streams. Set equal to CONFIG_QUIC_STREAM_RX_BUFFER_SIZE.
CONFIG_QUIC_INITIAL_MAX_STREAM_DATA_	16384	Initial receive window for peer-initiated bidirectional streams. Set equal to CONFIG_QUIC_STREAM_RX_BUFFER_SIZE.
CONFIG_QUIC_INITIAL_MAX_STREAM_DATA_	16384	Initial receive window for unidirectional streams opened by the peer. Only relevant when CONFIG_QUIC_MAX_STREAMS_UNI > 0.
CONFIG_QUIC_INITIAL_MAX_STREAMS_BIDI	QUIC_MAX	Maximum number of bidirectional streams the peer may open before receiving a MAX_STREAMS frame. Defaults to the local build-time limit.
CONFIG_QUIC_INITIAL_MAX_STREAMS_UNI	QUIC_MAX	Maximum number of unidirectional streams the peer may open.
CONFIG_QUIC_STREAM_RX_WINDOW_UPDATE_	25	Percentage of stream receive buffer consumed before a MAX_STREAM_DATA frame is sent to refill the peer's window. Lower values (e.g. 15 %) send updates more frequently, which improves throughput on high-latency links at the cost of additional control frames.

Buffer Sizes and RAM Allocation The following options directly control how much RAM the QUIC subsystem allocates at initialisation. See *Memory Model* for a detailed breakdown of how these interact.

Option	Default	Description
CONFIG_QUIC_ENDPOINT_PENDING_DATA_LE	1500	Receive staging buffer per endpoint in bytes. Should equal the network MTU. Combined with CONFIG_QUIC_PKT_COUNT, total endpoint RX staging = endpoints x pkt_count x this value.
CONFIG_QUIC_TX_BUFFER_SIZE	1500	Output packet assembly buffer per endpoint. Must be at least the network MTU (minimum 1280 for IPv6 compliance, maximum 1500). DPLPMTUD may limit the **validated** UDP payload below this value until a larger path MTU is confirmed; see *Path MTU Discovery (DPLPMTUD)*.
CONFIG_QUIC_CRYPTO_RX_BUFFER_SIZE	4096	Shared CRYPTO frame reassembly buffer per endpoint, used during the TLS handshake. Since QUIC progresses sequentially through encryption levels (Initial -> Handshake -> Application), a single buffer is reused across all levels. 4096 bytes is the minimum for interoperability with browsers: Chrome and Firefox may split a ClientHello across 10 or more CRYPTO frame fragments totalling up to ~4 KiB. Reduce to 2048 for embedded-to-embedded deployments with small certificates. Range: 1024-8192.
CONFIG_QUIC_CRYPTO_OOO_SLOTS	8	Number of out-of-order CRYPTO frame metadata slots per endpoint. Each slot (approximately 8 bytes) records the offset and length of one out-of-order fragment until the gap before it is filled. 8 slots are needed for browser interoperability; reduce to 4 for embedded-only deployments. Range: 4-16.
CONFIG_QUIC_STREAM_TX_BUFFER_SIZE	8192	Per-stream buffer holding sent but unacknowledged data. The QUIC sender cannot advance beyond the unacknowledged window, so this is the primary throughput-limiting parameter on high-latency links. Size to at least the bandwidth-delay product: throughput_bytes_per_ms x rtt_ms.
CONFIG_QUIC_STREAM_RX_BUFFER_SIZE	QUIC_STR	Per-stream receive buffer. Defaults to the TX buffer size, which is optimal for symmetric request/response patterns. Can be reduced independently for asymmetric workloads (e.g. download-only streams).
CONFIG_QUIC_STREAM_OOO_SLOTS	4	Number of out-of-order STREAM frame slots per stream. Each slot holds up to CONFIG_QUIC_STREAM_OOO_SEG_SIZE bytes. Set to 2 on loss-free local networks; increase to 8 or more on WAN links with significant reordering.
CONFIG_QUIC_STREAM_OOO_SEG_SIZE	1280	Maximum size of one buffered out-of-order segment. Setting this to the MTU avoids fragmentation of incoming frames. Can be reduced to 512 bytes when OOO delivery is uncommon.
CONFIG_QUIC_SENT_PKT_HISTORY_SIZE	64	Ring buffer depth of recently sent

Timeout Options

Option	Default	Description
`CONFIG_QUIC_MAX_IDLE_TIMEOUT`	30000	Milliseconds of inactivity after which the connection is silently closed. Both endpoints advertise their value; the smaller of the two takes effect. Set to 0 to disable idle timeout entirely. Recommended values: 5000 ms (loopback), 10000 ms (LAN), 30000 ms (WAN).
`CONFIG_QUIC_CONNECT_TIMEOUT`	3000	Milliseconds to wait for the handshake to complete before the `connect()` call fails. Should be at least 10x the expected RTT to accommodate packet loss during the handshake.
`CONFIG_QUIC_MAX_PTO_TIMEOUT_MS`	10000	Upper bound on the exponential PTO backoff in milliseconds. The PTO timer doubles on each consecutive probe timeout: `PTO_n = PTO_base x 2^n`. This cap prevents excessively long retransmission delays. Should not exceed `CONFIG_QUIC_MAX_IDLE_TIMEOUT`.

Service Thread Options

Option	Default	Description
`CONFIG_QUIC_SERVICE_THREAD_PRIO`	NUM_PRE	Thread priority of the QUIC service dispatcher. Values >= 0 are preemptive (0 = highest); values < 0 are cooperative.
`CONFIG_QUIC_SERVICE_STACK_SIZE`	4096	Stack size in bytes for the QUIC service thread. 4096 bytes is the default and is sufficient for Mbed TLS handshake operations. Reduce only if RAM is extremely constrained and profiling confirms the stack headroom is not needed.
`CONFIG_QUIC_PKT_COUNT`	QUIC_MA	Number of simultaneous pending packet receive operations. Defaults to the number of endpoints so that one packet per endpoint can be in flight concurrently. Increase (e.g. 2x endpoints) for higher-throughput scenarios.

Memory Model The table below shows how each RAM region scales with the configuration parameters. All sizes are in bytes unless noted.

Memory Region	Formula
Endpoint RX staging	QUIC_MAX_ENDPOINTS x QUIC_PKT_COUNT x QUIC_ENDPOINT_PENDING_DATA_LEN
Endpoint TX buffers	QUIC_MAX_ENDPOINTS x QUIC_TX_BUFFER_SIZE
Crypto RX buffers	QUIC_MAX_ENDPOINTS x QUIC_CRYPTO_RX_BUFFER_SIZE
Crypto OOO slots	QUIC_MAX_ENDPOINTS x QUIC_CRYPTO_OOO_SLOTS x 8
Stream TX buffers	(QUIC_MAX_CONTEXTS x (QUIC_MAX_STREAMS_BIDI + QUIC_MAX_STREAMS_UNI)) x QUIC_STREAM_TX_BUFFER_SIZE
Stream RX buffers	(QUIC_MAX_CONTEXTS x (QUIC_MAX_STREAMS_BIDI + QUIC_MAX_STREAMS_UNI)) x QUIC_STREAM_RX_BUFFER_SIZE
Stream OOO buffers	total_streams x QUIC_STREAM_OOO_SLOTS x QUIC_STREAM_OOO_SEG_SIZE
Sent-packet history	QUIC_MAX_CONTEXTS x QUIC_SENT_PKT_HISTORY_SIZE x 24
TLS transcript buffers	QUIC_MAX_CONTEXTS x QUIC_TLS_TRANSCRIPT_BUF_LEN
TLS context (Mbed TLS)	~8192 x QUIC_MAX_CONTEXTS (estimated; depends on ciphersuites)
Connection state	~512 x QUIC_MAX_CONTEXTS
Stream state	~128 x total_streams

Total estimated QUIC RAM (rough minimum for 1 connection, 3 bidi streams, 1500-byte stream buffers, LAN client defaults):

```
Endpoint RX staging :  2 x 1 x 1500 = 3 000 B
Endpoint TX         :  2 x 1500      = 3 000 B
Crypto RX buffers   :  2 x 2048      = 4 096 B  (embedded client, LAN)
Crypto OOO slots    :  2 x 4 x 8     =   256 B
Stream TX           :  3 x 1500      = 4 500 B
Stream RX           :  3 x 1500      = 4 500 B
OOO buffers         :  3 x 2 x 512   = 3 072 B
Sent-pkt history    :  1 x 16 x 24   =   384 B
TLS transcript      :  1 x 4096      = 4 096 B
TLS context         :  1 x 8192      = 8 192 B
State overhead      :  512 + 3x128   =   896 B
----------------------------------------------
Approximate total                   ~ 35 992 B (~35 KiB)
```

The dominant cost at low stream counts is the Mbed TLS context per connection. At higher stream counts, the stream TX/RX buffers become dominant.

Kconfig Optimizer Choosing buffer sizes that are both memory-efficient and able to sustain target throughput requires balancing several interdependent parameters. To assist with this, Zephyr provides a Python optimizer script:

```
scripts/net/quic-kconfig-optimizer.py
```

The script accepts a description of the deployment scenario – connection counts, MTU, network type, expected RTT, and a RAM budget – and outputs a complete set of recommended CONFIG_ values together with an estimated RAM breakdown and a ready-to-paste prj.conf snippet.

Prerequisites The script requires Python 3.8 or later with no additional dependencies beyond the standard library.

```
python3 --version   # must be 3.8+
```

Usage: Interactive Mode Run the script without arguments to enter the guided interactive mode. Each prompt shows its default in square brackets; press Enter to accept it.

```
python3 scripts/net/quic-kconfig-optimizer.py
```

Example session:

```
============================================================
  Zephyr QUIC Kconfig Optimizer -- Interactive Mode
============================================================
  Press Enter to accept [default] values.

Max QUIC connections (QUIC_MAX_CONTEXTS) [1]: 2
Bidirectional streams per connection [3]:
Unidirectional streams per connection [0]:
Network MTU in bytes [1500]:
RAM budget in bytes (0 = no limit) [0]: 131072
Device role [client] (client/server/both):
IP version(s) [ipv4] (ipv4/ipv6/both):
Expected RTT in ms [10]: 50
Expected packet loss rate (%) [0.0]:
Typical application message size (bytes) [1024]: 4096
Expect out-of-order delivery? [no] (yes/no):
Network type [lan] (lan/wan/loopback):
```

Usage: Command-Line Mode All parameters can be supplied on the command line. When every required parameter is present, the script skips interactive prompts entirely, which makes it suitable for CI pipelines and scripted board bringup.

```
python3 scripts/net/quic-kconfig-optimizer.py \
    --contexts        2          \
    --streams-bidi    3          \
    --streams-uni     0          \
    --mtu             1500       \
    --ram-budget      131072     \
    --role            client     \
    --ip-version      ipv4       \
    --rtt-ms          50         \
    --loss-rate       0.0        \
    --app-message-size 4096      \
    --ooo-expected    no         \
    --network-type    lan
```

If only some parameters are provided, the script falls back to interactive mode for the missing ones.

```
# Pre-fill the constraints, fill the rest interactively
python3 scripts/net/quic-kconfig-optimizer.py \
    --contexts 1 --streams-bidi 3 --ram-budget 65536
```

Input Parameters

Parameter	CLI flag	Default	Description
Max connections	`--context`	1	Maps directly to `CONFIG_QUIC_MAX_CONTEXTS`. Memory for TLS contexts, connection state, and sent-packet history scales linearly with this value.
Bidi streams	`--streams`	3	Bidirectional streams per connection. Stream TX/RX buffers are allocated for every stream across every connection, so `contexts x streams_bidi x 2 x stream_buf` bytes are reserved at boot.
Uni streams	`--streams`	0	Unidirectional streams per connection. Set to 0 if not needed to avoid allocating the associated receive buffers.
MTU	`--mtu`	1500	Network maximum transmission unit in bytes. Sets the floor for `CONFIG_QUIC_TX_BUFFER_SIZE` and `CONFIG_QUIC_ENDPOINT_PENDING_DATA_LEN`. Use 1280 for networks where IPv6 minimum MTU compliance is required and no Path MTU Discovery is available.
RAM budget	`--ram-bud`	0 (unlimited)	Hard upper bound on estimated QUIC RAM usage in bytes. The script aborts with an error if the computed allocation exceeds this value. Set to 0 to skip the budget check and simply report the estimate.
Device role	`--role`	`client`	`client`, `server`, or `both`. Affects the recommended number of endpoints: servers and dual-stack devices typically need one extra endpoint.
IP version	`--ip-vers`	`ipv4`	`ipv4`, `ipv6`, or `both`. When `both` is selected and `CONFIG_QUIC_ENDPOINT_USE_IPV4_MAPPING_TO_IPV6` is not enabled, an additional endpoint is needed.
Expected RTT	`--rtt-ms`	10	Round-trip time in milliseconds between the device and the remote peer. Used to size `CONFIG_QUIC_CONNECT_TIMEOUT`, `CONFIG_QUIC_MAX_PTO_TIMEOUT_MS`, and the window-update threshold.
Packet loss rate	`--loss-ra`	0.0	Expected packet loss as a percentage (0-100). Loss >= 2 % doubles stream buffer sizes to keep the pipeline full; loss >= 5 % forces the sent-packet history to at least 128 entries for effective loss detection.
Message size	`--app-mes`	1024	Typical application payload in bytes. Stream buffers are sized to hold at least 2x this value so that one complete message can be in transit while the previous one is being acknowledged.
OOO expected	`--ooo-exp`	`no`	Whether out-of-order packet delivery is expected (`yes`/`no`). When `no`, OOO slots are reduced to 2 and segment size to 512 B, saving `total_streams x (default_slots - 2) x seg_size` bytes.
Network type	`--network`	`lan`	`lan`, `wan`, or `loopback`. Controls idle timeout, PTO ceiling, and whether stream buffers are doubled to fill the larger bandwidth-delay product of a WAN link.

Output Sections The optimizer prints five sections.

Input Parameters – echoes all supplied values for verification.

Recommended Kconfig Values – the computed `CONFIG_*` values grouped by category (connection counts, transport parameters, buffer sizes, timeouts).

Estimated RAM Usage – a table showing each memory region, its size, and a proportional ASCII bar chart. If a RAM budget was given, the utilisation percentage is shown.

prj.conf Snippet – a copy-pasteable block of Kconfig options ready to drop into a `prj.conf` or board-specific `.conf` overlay.

Additional Notes – deployment-specific warnings and suggestions derived from the input, for example if the service stack is likely to be too small, or if the connection-level flow-control window may become a bottleneck.

Example Output The following shows the output for a minimal client configuration: one connection, three bidirectional streams, Ethernet LAN, 10 ms RTT, 1 KiB messages, no OOO.

```
========================================================================
         QUIC Kconfig Optimizer -- Recommended Settings
========================================================================

------------------------------------------------------------------------
  INPUT PARAMETERS
------------------------------------------------------------------------
  Max connections (QUIC_MAX_CONTEXTS)            1
  Bidi streams per connection                    3
  Uni  streams per connection                    0
  MTU (bytes)                                    1500
  RAM budget                                     unlimited
  Device role                                    client
  IP version                                     ipv4
  Expected RTT (ms)                              10
  Expected packet loss (%)                       0.0
  Typical message size (bytes)                   1024
  Out-of-order delivery expected                 no
  Network type                                   lan

------------------------------------------------------------------------
  RECOMMENDED Kconfig VALUES
------------------------------------------------------------------------

  [Connection / stream counts]
    CONFIG_QUIC_MAX_CONTEXTS                              1
    CONFIG_QUIC_MAX_STREAMS_BIDI                         3
    CONFIG_QUIC_MAX_STREAMS_UNI                          0
    CONFIG_QUIC_MAX_ENDPOINTS                            2
    CONFIG_QUIC_PKT_COUNT                                2

  [QUIC transport parameters (flow control)]
    CONFIG_QUIC_INITIAL_MAX_DATA                         6144
    CONFIG_QUIC_INITIAL_MAX_STREAM_DATA_BIDI_LOCAL       2048
    CONFIG_QUIC_INITIAL_MAX_STREAM_DATA_BIDI_REMOTE      2048
    CONFIG_QUIC_INITIAL_MAX_STREAM_DATA_UNI              16384
    CONFIG_QUIC_INITIAL_MAX_STREAMS_BIDI                 3
    CONFIG_QUIC_INITIAL_MAX_STREAMS_UNI                  0
    CONFIG_QUIC_STREAM_RX_WINDOW_UPDATE_THRESHOLD        25
```

(continues on next page)

(continued from previous page)

```
  [Buffer sizes (RAM)]
    CONFIG_QUIC_ENDPOINT_PENDING_DATA_LEN                1500
    CONFIG_QUIC_TX_BUFFER_SIZE                           1500
    CONFIG_QUIC_CRYPTO_RX_BUFFER_SIZE                    2048
    CONFIG_QUIC_CRYPTO_OOO_SLOTS                         4
    CONFIG_QUIC_STREAM_TX_BUFFER_SIZE                    2048
    CONFIG_QUIC_STREAM_RX_BUFFER_SIZE                    2048
    CONFIG_QUIC_STREAM_OOO_SLOTS                         2
    CONFIG_QUIC_STREAM_OOO_SEG_SIZE                      512
    CONFIG_QUIC_SENT_PKT_HISTORY_SIZE                    16
    CONFIG_QUIC_TLS_TRANSCRIPT_BUF_LEN                   4096

  [Timeouts]
    CONFIG_QUIC_MAX_IDLE_TIMEOUT                         10000
    CONFIG_QUIC_CONNECT_TIMEOUT                          3000
    CONFIG_QUIC_MAX_PTO_TIMEOUT_MS                       5000

----------------------------------------------------------------------
  ESTIMATED RAM USAGE
----------------------------------------------------------------------
  Endpoint RX staging              9 KiB  ######
  Endpoint TX buffers              3 KiB  ##
  Crypto RX buffers                4 KiB  ###
  Crypto OOO slots                  64 B
  Stream TX buffers                6 KiB  ####
  Stream RX buffers                6 KiB  ####
  Stream OOO buffers               2 KiB  #
  Stream state overhead            0 KiB
  Sent-packet history              0 KiB
  TLS context overhead             8 KiB  #####
  Connection state overhead        0 KiB
  TLS transcript buffers           4 KiB  ##
  TOTAL                           42 KiB  ######################### <-- TOTAL

----------------------------------------------------------------------
  prj.conf SNIPPET
----------------------------------------------------------------------

CONFIG_QUIC_MAX_CONTEXTS=1
CONFIG_QUIC_MAX_STREAMS_BIDI=3
...
```

Minimal IoT sensor (client, IPv4 only, tight RAM budget) A sensor that opens one QUIC connection to a cloud endpoint, uses a single bidirectional stream to report telemetry, and has 48 KiB available for QUIC:

```
python3 scripts/net/quic-kconfig-optimizer.py \
    --contexts 1 --streams-bidi 1 --streams-uni 0 \
    --mtu 1280 --ram-budget 49152 \
    --role client --ip-version ipv4 \
    --rtt-ms 80 --loss-rate 0.5 \
    --app-message-size 256 \
    --ooo-expected no --network-type wan
```

The script will size stream buffers to 1500 B (minimum, since `2 x 256 = 512 < 1500`), set OOO slots to 2, and keep the sent-packet history at 16 entries – all chosen to minimise RAM while still

handling the expected 0.5 % loss on the WAN link.

Dual-stack server (IPv4 + IPv6, multiple clients) An edge gateway accepting up to four simultaneous client connections over both IPv4 and IPv6, with 256 KiB available for QUIC:

```
python3 scripts/net/quic-kconfig-optimizer.py \
    --contexts 4 --streams-bidi 3 --streams-uni 1 \
    --mtu 1500 --ram-budget 262144 \
    --role server --ip-version both \
    --rtt-ms 5 --loss-rate 0.0 \
    --app-message-size 8192 \
    --ooo-expected no --network-type lan
```

The script will recommend three endpoints (two for the listening sockets, one for each client family), and size stream buffers to 16 KiB (to hold 2 x 8192 B messages), which with 4 contexts and 4 streams each will dominate the RAM budget.

High-throughput WAN link with packet loss A data aggregator connected over a cellular WAN with measurable packet loss, requiring sustained throughput for firmware-update payloads:

```
python3 scripts/net/quic-kconfig-optimizer.py \
    --contexts 1 --streams-bidi 2 --streams-uni 0 \
    --mtu 1400 --ram-budget 0 \
    --role client --ip-version ipv4 \
    --rtt-ms 150 --loss-rate 3.0 \
    --app-message-size 32768 \
    --ooo-expected yes --network-type wan
```

Here the script doubles stream buffers (loss >= 2 %) and selects a window update threshold of 15 % (high-latency WAN) to keep the peer's send window open, maximising pipeline utilisation. It also sets OOO slots to 8 with 1280-byte segments to handle cellular reordering.

Tuning Guidelines **Start with the optimizer defaults, then measure.** The script's heuristics are conservative. Use the Zephyr network statistics (`CONFIG_NET_STATISTICS`) to observe actual stream window stalls and OOO events before deciding to increase buffer sizes.

Stream buffers are the biggest RAM cost. On multi-connection devices, `contexts x streams x 2 x stream_buf` dominates. Halving the stream buffer size saves proportionally. Only increase beyond the BDP if profiling shows persistent sender stalls.

The connection-level window must not be the bottleneck. `CONFIG_QUIC_INITIAL_MAX_DATA` must be greater than or equal to the largest per-stream window; otherwise the connection cap limits throughput even when individual stream windows are open.

CRYPTO buffers scale with endpoints, not connections. `CONFIG_QUIC_CRYPTO_RX_BUFFER_SIZE` and `CONFIG_QUIC_CRYPTO_OOO_SLOTS` are allocated once per endpoint, so their RAM cost is `QUIC_MAX_ENDPOINTS x buffer_size`. The 4096-byte default is required for browser interoperability (Chrome and Firefox fragment ClientHello into many small CRYPTO frames); reduce to 2048 for embedded-only deployments where the peer is also a Zephyr device.

IPv4-mapped IPv6. When both IPv4 and IPv6 are needed, enabling `CONFIG_QUIC_ENDPOINT_USE_IPV4_MAPPING_TO_IPV6` allows one UDP socket to serve both address families, reducing `CONFIG_QUIC_MAX_ENDPOINTS` by one and saving the associated buffer RAM.

Disable unused stream directions. If the application only uses client-initiated streams, set `CONFIG_QUIC_MAX_STREAMS_UNI` to 0. The optimizer accounts for this automatically.

OOO memory is per-stream. `QUIC_STREAM_OOO_SLOTS x QUIC_STREAM_OOO_SEG_SIZE x total_streams` can be significant. On a reliable LAN, setting `CONFIG_QUIC_STREAM_OOO_SLOTS` to 2 and `CONFIG_QUIC_STREAM_OOO_SEG_SIZE` to 512 is safe and saves several KiB per connection.

API Reference

QUIC Library

Related code samples

- Echo client (QUIC)Implements a simple IPv4/IPv6 echo client using QUIC API and socket service API.
- Echo server (QUIC)Implements a simple IPv4/IPv6 echo server using QUIC API and socket service API.

Network System Management

Network Configuration Library

- *Overview*
- *SNTP clock initialization*
- *Sample usage*
- *API Reference*

Overview The network configuration library sets up networking devices in a semi-automatic way during the system boot, based on user-supplied Kconfig options.

The following Kconfig options affect how configuration library will setup the system:

Table 9: Kconfig options for network configuration library

Option name	Description
CONFIG_NET_CONFIG_SETTINGS	This option controls whether the network system is configured or initialized at all. If not set, then the config library is not used for initialization and the application needs to do all the network related configuration itself. If this option is set, then the user can optionally configure static IP addresses to be set to the first network interface in the system. Typically setting static IP addresses is only usable in testing and should not be used in production code. See the config library Kconfig file subsys/net/lib/config/Kconfig for specific options to set the static IP addresses.
CONFIG_NET_CONFIG_AUTO_INIT	The networking system is automatically configured when the device is started.
CONFIG_NET_CONFIG_INIT_TIMEOUT	This tells how long to wait for the networking to be ready and available. If for example IPv4 address from DHCPv4 is not received within this limit, then a call to net_config_init() will return error during the device startup.
CONFIG_NET_CONFIG_NEED_IPV4	The network application needs IPv4 support to function properly. This option makes sure the network application is initialized properly in order to use IPv4. If CONFIG_NET_IPV4 is not enabled, then setting this option will automatically enable IPv4.
CONFIG_NET_CONFIG_NEED_IPV6	The network application needs IPv6 support to function properly. This option makes sure the network application is initialized properly in order to use IPv6. If CONFIG_NET_IPV6 is not enabled, then setting this option will automatically enable IPv6.
CONFIG_NET_CONFIG_NEED_IPV6_ROUTER	If IPv6 is enabled, then this option tells that the network application needs IPv6 router to exists before continuing. This means in practice that the application wants to wait until it receives IPv6 router advertisement message before continuing.
CONFIG_NET_CONFIG_MY_IPV6_ADDR	Local static IPv6 address assigned to the default network interface.
CONFIG_NET_CONFIG_PEER_IPV6_ADDR	Peer static IPv6 address. This is mainly useful in testing setups where the application can connect to a pre-defined host.
CONFIG_NET_CONFIG_MY_IPV4_ADDR	Local static IPv4 address assigned to the default network interface.
CONFIG_NET_CONFIG_MY_IPV4_NETMASK	Static IPv4 netmask assigned to the IPv4 address.
CONFIG_NET_CONFIG_MY_IPV4_GW	Static IPv4 gateway address assigned to the default network interface.
CONFIG_NET_CONFIG_PEER_IPV4_ADDR	Peer static IPv4 address. This is mainly useful in testing setups where the application can connect to a pre-defined host.

SNTP clock initialization If CONFIG_NET_CONFIG_CLOCK_SNTP_INIT is set, the library also initializes the system clock from an SNTP server, and optionally keeps it in sync with CONFIG_NET_CONFIG_SNTP_INIT_RESYNC.

The server to query is looked up in the following order:

1. The server set at runtime with net_config_sntp_set_server(). This requires CON-

FIG_NET_CONFIG_SNTP_INIT_SERVER_RUNTIME and is meant for applications that only learn which time server to use while running, for example after provisioning.

2. The server received via the DHCPv4 NTP server option, if CONFIG_NET_CONFIG_SNTP_INIT_SERVER_USE_DHCPV4_OPTION is set.
3. CONFIG_NET_CONFIG_SNTP_INIT_SERVER, which must be set if CONFIG_NET_CONFIG_CLOCK_SNTP_INIT is set but neither CONFIG_NET_CONFIG_SNTP_INIT_SERVER_RUNTIME nor CONFIG_NET_CONFIG_SNTP_INIT_SERVER_USE_DHCPV4_OPT is enabled.

Sample usage If CONFIG_NET_CONFIG_AUTO_INIT is set, then the configuration library is automatically enabled and run during the device boot. In this case, the library will call net_config_init() automatically and the application does not need to do any network configuration.

If you want to use the network configuration library but without automatic initialization, you can call net_config_init() manually. The flags parameter can be used to give hints to the library about what kind of functionality the application wishes to have before the actual application starts.

API Reference
Network Configuration Library

Related code samples

- Legacy USB device netusb sampleUse USB CDC EEM/ECM or RNDIS implementation
- zperf: Network Traffic GeneratorUse the zperf shell utility to evaluate network bandwidth.

DHCPv4

- *Overview*
- *Sample usage*
- *API Reference*

Overview The Dynamic Host Configuration Protocol (DHCP) is a network management protocol used on IPv4 networks. A DHCPv4 server dynamically assigns an IPv4 address and other network configuration parameters to each device on a network so they can communicate with other IP networks. See this DHCP Wikipedia article for a detailed overview of how DHCP works.

Note that Zephyr supports both DHCPv4 client and server functionality.

Sample usage See dhcpv4-client sample application for details.

API Reference
DHCPv4

Related code samples

- DHCPv4 clientStart a DHCPv4 client to obtain an IPv4 address from a DHCPv4 server.

DHCPv6

- *Overview*
- *Prefix delegation*
- *Limitations*
- *API Reference*

Overview The Dynamic Host Configuration Protocol (DHCP) for IPv6 is a network management protocol used on IPv6 based networks. A DHCPv6 server dynamically assigns an IPv6 address and other network configuration parameters to each device on a network so they can communicate with other IP networks. See this DHCPv6 Wikipedia article for a detailed overview of how DHCPv6 works.

Zephyr supports both DHCPv6 client and server functionality, including IPv6 prefix delegation (IA_PD) as specified in RFC 8415.

Prefix delegation The DHCPv6 client can act as a *requesting router*: in addition to requesting a non-temporary address (IA_NA), it can request a delegated prefix (IA_PD) by setting `net_dhcpv6_params.request_prefix`. The delegated prefix is installed on the requesting interface.

To turn the node into a full requesting router that sub-delegates the prefix to one or more downstream links, list the downstream (LAN) interface indices in `net_dhcpv6_params.downstream_ifaces` and set `net_dhcpv6_params.downstream_count` to the number of entries. Each downstream interface is assigned a distinct `/64` carved from the delegated prefix (the Nth interface gets the Nth `/64`), which is then advertised on that interface via Router Advertisements, allowing downstream hosts to auto-configure addresses using SLAAC. The array holds at most `CONFIG_NET_DHCPV6_MAX_DOWNSTREAM` entries; a `net_dhcpv6_params.downstream_count` of 0 means the delegated prefix is only installed on the requesting interface. This requires `CONFIG_NET_IPV6_ND_RA_TX` (Router Advertisement transmit / router role) and, for traffic to be forwarded between the upstream and downstream interfaces, `CONFIG_NET_IPV6_FORWARDING`.

The delegation has to be short enough to contain a distinct `/64` for every downstream interface, that is, a delegated prefix of length `L` can serve at most `2^(64 - L)` downstream links. A delegation of exactly a `/64` can therefore serve a single link, and one longer than a `/64` cannot be sub-delegated at all. Downstream interfaces that cannot be given a distinct `/64` are skipped with a warning.

The DHCPv6 server (`CONFIG_NET_DHCPV6_SERVER`) implements the *delegating router* role, handing out addresses and delegated prefixes from configured pools. See dhcpv6-pd for a complete example of both roles.

Limitations The implementation deliberately leaves out a few things that RFC 8415 and RFC 4861 allow to be simplified:

- A Release sent when the client is stopped (`CONFIG_NET_DHCPV6_SEND_RELEASE_ON_STOP`) is best effort, it is transmitted once and is not retransmitted `REL_MAX_RC` times as described in **RFC 8415 Section 18.2.7**. A lease that is not released is reclaimed by the server once it expires.
- Unsolicited Router Advertisements are sent at the fixed interval configured with `CONFIG_NET_IPV6_ND_RA_TX_INTERVAL` instead of at a random interval between `MinRtrAdvInterval` and `MaxRtrAdvInterval`, and no initial burst of `MAX_INITIAL_RTR_ADVERTISEMENTS` is sent when an interface takes the router role. Solicited advertisements are rate limited and randomly delayed as required by **RFC 4861 Section 6.2.6**.

- The server assigns addresses and prefixes by varying a single byte of the configured pool base, so pools are limited to `CONFIG_NET_DHCPV6_SERVER_MAX_LEASES` entries and IA_PD pool prefix lengths have to be byte aligned.

API Reference

DHCPv6

Related code samples

- DHCPv6 prefix delegationObtain and delegate an IPv6 prefix using DHCPv6 (RFC 8415).

DHCPv6 server

Related code samples

- DHCPv6 prefix delegationObtain and delegate an IPv6 prefix using DHCPv6 (RFC 8415).

Hostname Configuration

- *Overview*
- *API Reference*

Overview A networked device might need a hostname, for example, if the device is configured to be a mDNS responder (see *DNS Resolve* for details) and needs to respond to `<hostname>.local` DNS queries.

The `CONFIG_NET_HOSTNAME_ENABLE` must be set in order to store the hostname and enable the relevant APIs. If the option is enabled, then the default hostname is set to be `zephyr` by `CONFIG_NET_HOSTNAME` option.

If the same firmware image is used to flash multiple boards, then it is not practical to use the same hostname in all of the boards. In that case, one can enable `CONFIG_NET_HOSTNAME_UNIQUE` which will add a unique postfix to the hostname. By default the link local address of the first network interface is used as a postfix. In Ethernet networks, the link local address refers to MAC address. For example, if the link local address is `01:02:03:04:05:06`, then the unique hostname could be `zephyr010203040506`. If you want to set the prefix yourself, then call `net_hostname_set_postfix_str()` before the network interfaces are created. Alternatively, if you prefer a hexadecimal conversion for the prefix, then call `net_hostname_set_postfix()`. For example for the Ethernet networks, the initialization priority is set by `CONFIG_ETH_INIT_PRIORITY` so you would need to set the postfix before that. The postfix can be set only once.

API Reference

Network Hostname Library

Network Core Helpers

- *Overview*
- *API Reference*

Overview The network subsystem contains two functions for sending and receiving data from the network. The `net_recv_data()` is typically used by network device driver when the received network data needs to be pushed up in the network stack for further processing. All the data is received via a network interface which is typically created by the device driver.

For sending, the `net_send_data()` can be used. Typically applications do not call this function directly as there is the *BSD Sockets* API for sending and receiving network data.

API Reference
Network Core Library

Related code samples

- mDNS responderListen and respond to mDNS queries.
- Telnet consoleAccess Zephyr shell over telnet.

Network Interface

- *Overview*
- *Network interface state management*
- *API Reference*

Overview The network interface is a nexus that ties the network device drivers and the upper part of the network stack together. All the sent and received data is transferred via a network interface. The network interfaces cannot be created at runtime. A special linker section will contain information about them and that section is populated at linking time.

Network interfaces are created by `NET_DEVICE_INIT()` macro. For Ethernet network, a macro called `ETH_NET_DEVICE_INIT()` should be used instead as it will create VLAN interfaces automatically if `CONFIG_NET_VLAN` is enabled. These macros are typically used in network device driver source code.

The network interface can be turned ON by calling `net_if_up()` and OFF by calling `net_if_down()`. When the device is powered ON, the network interface is also turned ON by default.

The network interfaces can be referenced either by a `struct net_if *` pointer or by a network interface index. The network interface can be resolved from its index by calling `net_if_get_by_index()` and from interface pointer by calling `net_if_get_by_iface()`.

The IP address for network devices must be set for them to be connectable. In a typical dynamic network environment, IP addresses are set automatically by DHCPv4, for example. If needed though, the application can set a device's IP address manually. See the API documentation below for functions such as `net_if_ipv4_addr_add()` that do that.

The `net_if_get_default()` returns a *default* network interface. What this default interface means can be configured via options like `CONFIG_NET_DEFAULT_IF_FIRST` and `CONFIG_NET_DEFAULT_IF_ETHERNET`. See Kconfig file subsys/net/ip/Kconfig what options are available for selecting the default network interface.

The transmitted and received network packets can be classified via a network packet priority. This is typically done in Ethernet networks when virtual LANs (VLANs) are used. Higher priority packets can be sent or received earlier than lower priority packets. The traffic class setup can be configured by `CONFIG_NET_TC_TX_COUNT` and `CONFIG_NET_TC_RX_COUNT` options.

If the CONFIG_NET_PROMISCUOUS_MODE is enabled and if the underlying network technology supports promiscuous mode, then it is possible to receive all the network packets that the network device driver is able to receive. See *Promiscuous Mode* API for more details.

Network interface state management Zephyr distinguishes between two interface states: administrative state and operational state, as described in RFC 2863. The administrative state indicate whether an interface is turned ON or OFF. This state is represented by NET_IF_UP flag and is controlled by the application. It can be changed by calling net_if_up() or net_if_down() functions. Network drivers or L2 implementations should not change administrative state on their own.

Bringing an interface up however not always means that the interface is ready to transmit packets. Because of that, operational state, which represents the internal interface status, was implemented. The operational state is updated whenever one of the following conditions take place:

- The interface is brought up/down by the application (administrative state changes).
- The interface is notified by the driver/L2 that PHY status has changed.
- The interface is notified by the driver/L2 that it joined/left a network.

The PHY status is represented with NET_IF_LOWER_UP flag and can be changed with net_if_carrier_on() and net_if_carrier_off(). By default, the flag is set on a newly initialized interface. An example of an event that changes the carrier state is Ethernet cable being plugged in or out.

The network association status is represented with NET_IF_DORMANT flag and can be changed with net_if_dormant_on() and net_if_dormant_off(). By default, the flag is cleared on a newly initialized interface. An example of an event that changes the dormant state is a Wi-Fi driver successfully connecting to an access point. In this scenario, driver should set the dormant state to ON during initialization, and once it detects that it connected to a Wi-Fi network, the dormant state should be set to OFF.

The operational state of an interface is updated as follows:

- `!net_if_is_admin_up()`

  Interface is in NET_IF_OPER_DOWN.

- `net_if_is_admin_up() && !net_if_is_carrier_ok()`

  Interface is in NET_IF_OPER_DOWN or NET_IF_OPER_LOWERLAYERDOWN if the interface is stacked (virtual).

- `net_if_is_admin_up() && net_if_is_carrier_ok() && net_if_is_dormant()`

  Interface is in NET_IF_OPER_DORMANT.

- `net_if_is_admin_up() && net_if_is_carrier_ok() && !net_if_is_dormant()`

  Interface is in NET_IF_OPER_UP.

Only after an interface enters NET_IF_OPER_UP state the NET_IF_RUNNING flag is set on the interface indicating that the interface is ready to be used by the application.

API Reference

Network Interface abstraction layer

Related code samples

- bridgeTest and debug Ethernet bridge
- IPv4 autoconf clientPerform IPv4 autoconfiguration and self-assign a random IPv4 address

- Network management socketListen to network management events using a network management socket.
- Telnet consoleAccess Zephyr shell over telnet.
- Virtual LANSetup two virtual LAN networks and use net-shell to view the networks' settings.

L2 Layer Management

- *Overview*
- *L2 layer API*
- *Network Device drivers*
  - *Ethernet device driver*
  - *IEEE 802.15.4 device driver*
- *API Reference*

Overview The L2 stack is designed to hide the whole networking link-layer part and the related device drivers from the upper network stack. This is made through a `net_if` declared in include/zephyr/net/net_if.h.

The upper layers are unaware of implementation details beyond the net_if object and the generic API provided by the L2 layer in include/zephyr/net/net_l2.h as `net_l2`.

Only the L2 layer can talk to the device driver, linked to the net_if object. The L2 layer dictates the API provided by the device driver, specific for that device, and optimized for working together.

Currently, there are L2 layers for *Ethernet*, *IEEE 802.15.4 Soft-MAC*, *CANBUS*, *OpenThread*, Wi-Fi, and a dummy layer example that can be used as a template for writing a new one.

L2 layer API In order to create an L2 layer, or a driver for a specific L2 layer, one needs to understand how the L3 layer interacts with it and how the L2 layer is supposed to behave. See also *network stack architecture* for more details. The generic L2 API has these functions:

- `recv()`: All device drivers, once they receive a packet which they put into a `net_pkt`, will push this buffer to the network stack via `net_recv_data()`. At this point, the network stack does not know what to do with it. Instead, it passes the buffer along to the L2 stack's `recv()` function for handling. The L2 stack does what it needs to do with the packet, for example, parsing the link layer header, or handling link-layer only packets. The `recv()` function will return `NET_DROP` in case of an erroneous packet, `NET_OK` if the packet was fully consumed by the L2, or `NET_CONTINUE` if the network stack should then handle it.
- `send()`: Similar to receive function, the network stack will call this function to actually send a network packet. All relevant link-layer content will be generated and added by this function. The `send()` function returns the number of bytes sent, or a negative error code if there was a failure sending the network packet.
- `enable()`: This function is used to enable/disable traffic over a network interface. The function returns `<0` if error and `>=0` if no error.
- `get_flags()`: This function will return the capabilities of an L2 driver, for example whether the L2 supports multicast or promiscuous mode.

Network Device drivers Network device drivers fully follows Zephyr device driver model as a basis. Please refer to *Device Driver Model*.

There are, however, two differences:

- The driver_api pointer must point to a valid `net_if_api` pointer.
- The network device driver must use `NET_DEVICE_INIT_INSTANCE()` or `ETH_NET_DEVICE_INIT()` for Ethernet devices. These macros will call the `DEVICE_DEFINE()` macro, and also instantiate a unique `net_if` related to the created device driver instance.

Implementing a network device driver depends on the L2 stack it belongs to: *Ethernet*, *IEEE 802.15.4*, etc. In the next section, we will describe how a device driver should behave when receiving or sending a network packet. The rest is hardware dependent and is not detailed here.

Ethernet device driver On reception, it is up to the device driver to fill-in the network packet with as many data buffers as required. The network packet itself is a `net_pkt` and should be allocated through `net_pkt_rx_alloc_with_buffer()`. Then all data buffers will be automatically allocated and filled by `net_pkt_write()`.

After all the network data has been received, the device driver needs to call `net_recv_data()`. If that call fails, it will be up to the device driver to unreference the buffer via `net_pkt_unref()`.

On sending, the device driver send function will be called, and it is up to the device driver to send the network packet all at once, with all the buffers.

Each Ethernet device driver will need, in the end, to call `ETH_NET_DEVICE_INIT()` like this:

```
ETH_NET_DEVICE_INIT(..., CONFIG_ETH_INIT_PRIORITY,
                    &the_valid_net_if_api_instance, 1500);
```

IEEE 802.15.4 device driver Device drivers for IEEE 802.15.4 L2 work basically the same as for Ethernet. What has been described above, especially for `recv()`, applies here as well. There are two specific differences however:

- It requires a dedicated device driver API: `ieee802154_radio_api`, which overloads `net_if_api`. This is because 802.15.4 L2 needs more from the device driver than just `send()` and `recv()` functions. This dedicated API is declared in include/zephyr/net/ieee802154_radio.h. Each and every IEEE 802.15.4 device driver must provide a valid pointer on such relevantly filled-in API structure.
- Sending a packet is slightly different than in Ethernet. Most IEEE 802.15.4 PHYs support relatively small frames only, 127 bytes all inclusive: frame header, payload and frame checksum. Buffers to be sent over the radio will often not fit this frame size limitation, e.g. a buffer containing an IPv6 packet will often have to be split into several fragments and IP6 packet headers and fragments need to be compressed using a protocol like 6LoWPAN before being passed on to the radio driver. Additionally the IEEE 802.15.4 standard defines medium access (e.g. CSMA/CA), frame retransmission, encryption and other pre-processing procedures (e.g. addition of information elements) that individual radio drivers should not have to care about. This is why the `ieee802154_radio_api` requires a tx function pointer which differs from the `net_if_api` send function pointer. Zephyr's native IEEE 802.15.4 L2 implementation provides a generic `ieee802154_send()` instead, meant to be given as `net_if` send function. The implementation of `ieee802154_send()` takes care of IEEE 802.15.4 standard packet preparation procedures, splitting the packet into possibly compressed, encrypted and otherwise pre-processed fragment buffers, sending one buffer at a time through `ieee802154_radio_api` tx function and unreferencing the network packet only when the transmission as a whole was either successful or failed.

Interaction between IEEE 802.15.4 radio device drivers and L2 is bidirectional:

- L2 -> L1: Methods as `ieee802154_send()` and several IEEE 802.15.4 net management calls will call into the driver, e.g. to send a packet over the radio link or re-configure

the driver at runtime. These incoming calls will all be handled by the methods in the `ieee802154_radio_api`.

- L1 -> L2: There are several situations in which the driver needs to initiate calls into the L2/MAC layer. Zephyr's IEEE 802.15.4 L1 -> L2 adaptation API employs an "inversion-of-control" pattern in such cases avoids duplication of complex logic across independent driver implementations and ensures implementation agnostic loose coupling and clean separation of concerns between MAC (L2) and PHY (L1) whenever reverse information transfer or close co-operation between hardware and L2 is required. During driver initialization, for example, the driver calls `ieee802154_init()` to pass the interface's MAC address as well as other hardware-related configuration to L2. Similarly, drivers may indicate performance or timing critical radio events to L2 that require close integration with the hardware (e.g. `ieee802154_handle_ack()`). Calls from L1 into L2 are not implemented as methods in `ieee802154_radio_api` but are standalone functions declared and documented as such in include/zephyr/net/ieee802154_radio.h. The API documentation will clearly state which functions must be implemented by all L2 stacks as part of the L1 -> L2 "inversion-of-control" adaptation API.

Note: Standalone functions in include/zephyr/net/ieee802154_radio.h that are not explicitly documented as callbacks are considered to be helper functions within the PHY (L1) layer implemented independently of any specific L2 stack, see for example `ieee802154_is_ar_flag_set()`.

As all net interfaces, IEEE 802.15.4 device driver implementations will have to call `NET_DEVICE_INIT_INSTANCE()` in the end:

```
NET_DEVICE_INIT_INSTANCE(...,
                         the_device_init_prio,
                         &the_valid_ieee802154_radio_api_instance,
                         IEEE802154_L2,
                         NET_L2_GET_CTX_TYPE(IEEE802154_L2), 125);
```

API Reference

Network L2 Abstraction Layer

Related code samples

- Link Layer Discovery Protocol (LLDP)Enable LLDP support and setup VLANs.
- Virtual LANSetup two virtual LAN networks and use net-shell to view the networks' settings.

Network Traffic Offloading

- *Network Offloading*
  - *Overview*
  - *API Reference*
- *Socket Offloading*
  - *Overview*

Network Offloading

Overview The network offloading API provides hooks that a device vendor can use to provide an alternate implementation for an IP stack. This means that the actual network connection

creation, data transfer, etc., is done in the vendor HAL instead of the Zephyr network stack.

API Reference
Network Offloading Interface

Socket Offloading

Overview In addition to the network offloading API, Zephyr allows offloading of networking functionality at the socket API level. With this approach, vendors who provide an alternate implementation of the networking stack, exposing socket API for their networking devices, can easily integrate it with Zephyr.

See drivers/wifi/simplelink/simplelink_sockets.c for a sample implementation on how to integrate network offloading at socket level.

Link Layer Address Handling

- *Overview*
- *API Reference*

Overview The link layer addresses are set for network interfaces so that L2 connectivity works correctly in the network stack. Typically the link layer addresses are 6 bytes long like in Ethernet but for IEEE 802.15.4 the link layer address length is 8 bytes.

API Reference
Network Link Address Library

Ethernet Management

- *Overview*
- *API Reference*

Overview Ethernet management API provides functions to manage the Ethernet network interface low level status. The caller of these functions can:

- raise `carrier ON` or `carrier OFF` management events
- raise `VLAN enabled` or `VLAN disabled` management events

Typically the `carrier OFF` event would be generated by the Ethernet device driver when it notices that the Ethernet cable is disconnected. The `carrier ON` event would be generated if the Ethernet device driver notices that the Ethernet cable is re-connected.

Currently the VLAN events are generated by the Ethernet L2 layer when a specific VLAN tag is either enabled or disabled.

The user application can monitor these events if it needs to act when the corresponding status changes.

API Reference
Ethernet Library

Related code samples

- TSN SwitchTest and debug TSN switch

Traffic Classification

Overview Traffic classification is an automated process that categorizes computer network traffic according to various parameters. For Zephyr, the VLAN priority code point (PCP) is used to classify both received and sent network packets. See more information about VLAN priority at IEEE 802.1Q.

By default, all network traffic is treated equal in Zephyr. If desired, the option `CONFIG_NET_TC_TX_COUNT` can be used to set the number of transmit queues. The option `CONFIG_NET_TC_RX_COUNT` can be used to set the number of receive queues. Each traffic class queue corresponds to a specific kernel work queue. Each kernel work queue has a priority. The VLAN priority is mapped to a certain traffic class according to rules specified in IEEE 802.1Q spec chapter I.3, chapter 8.6.6 table 8-4, and chapter 34.5 table 34-1. Each traffic class is in turn mapped to a certain kernel work queue. The maximum number of traffic classes for both Rx and Tx is 8.

See subsys/net/ip/net_tc.c for details of how various mappings are done.

Network Packet Filtering

- *Overview*
- *Examples*
- *API Reference*

Overview The Network Packet Filtering facility provides the infrastructure to construct custom rules for accepting and/or denying packet transmission and reception. It also allows to modify the priority of incoming network packets. This can be used to create a basic firewall, control network traffic, etc.

The `CONFIG_NET_PKT_FILTER` must be set in order to enable the relevant APIs.

Both the transmission and reception paths may have a list of filter rules. Each rule is made of a set of conditions and a packet outcome. Every packet is subjected to the conditions attached to a rule. When all the conditions for a given rule are true then the packet outcome is immediately determined as specified by the current rule and no more rules are considered. If one condition is false then the next rule in the list is considered.

Packet outcome is either `NET_OK` to accept the packet, `NET_DROP` to drop it or `NET_CONTINUE` to modify its priority on the fly.

When the outcome is `NET_CONTINUE` the priority is updated but the final outcome is not yet determined and processing continues. If all conditions of multiple rules are true, then the packet gets the priority of the rule last considered.

A rule is represented by a `npf_rule` object. It can be inserted to, appended to or removed from a rule list contained in a `npf_rule_list` object using `npf_insert_rule()`, `npf_append_rule()`, and `npf_remove_rule()`.

There are different sets of rules for different layers in the network stack. Some of the rules are applied for L2 layer like Ethernet, and some for L3 layer where IPv4 or IPv6 protocol is handled.

The `local_in` rules are for matching incoming protocol types like UDP or TCP packets that are run on top of IPv4 or IPv6. The rule support for different layers can be controlled by relevant Kconfig options mentioned below.

- `npf_send_rules` is a rule list applied to outgoing packets in L2 layer
- `npf_recv_rules` is a rule list applied to incoming packets in L2 layer
- `npf_ipv4_recv_rules` is a rule list applied for incoming IPv4 packets. Can be enabled or disabled by `CONFIG_NET_PKT_FILTER_IPV4_HOOK` option.
- `npf_ipv6_recv_rules` is a rule list applied for incoming IPv6 packets. Can be enabled or disabled by `CONFIG_NET_PKT_FILTER_IPV6_HOOK` option.
- `npf_local_in_recv_rules` is a rule list applied for incoming UDP or TCP packets. Can be enabled or disabled by `CONFIG_NET_PKT_FILTER_LOCAL_IN_HOOK` option.

If a filter rule list is empty then `NET_OK` is assumed. If a non-empty rule list runs to the end then `NET_DROP` is assumed. However it is recommended to always terminate a non-empty rule list with an explicit default termination rule, either `npf_default_ok` or `npf_default_drop`.

Rule conditions are represented by a `npf_test`. This structure can be embedded into a larger structure when a specific condition requires extra test data. It is up to the test function for such conditions to retrieve the outer structure from the provided `npf_test` structure pointer.

Convenience macros are provided in include/zephyr/net/net_pkt_filter.h to statically define condition instances for various conditions, and `NPF_RULE()` and `NPF_PRIORITY()` to create a rule instance with an immediate outcome or a priority change.

See also net-pkt-filter sample for an example of how to create and manage packet filters. The network shell has a `net filter` command that can be used to see the installed rules at runtime.

Examples Here's an example usage:

```
static NPF_SIZE_MAX(maxsize_200, 200);
static NPF_ETH_TYPE_MATCH(ip_packet, NET_ETH_PTYPE_IP);

static NPF_RULE(small_ip_pkt, NET_OK, ip_packet, maxsize_200);

void install_my_filter(void)
{
    npf_insert_recv_rule(&npf_default_drop);
    npf_insert_recv_rule(&small_ip_pkt);
}
```

The above would accept IP packets that are 200 bytes or smaller, and drop all other packets.

Another (less efficient) way to achieve the same result could be:

```
static NPF_SIZE_MIN(minsize_201, 201);
static NPF_ETH_TYPE_UNMATCH(not_ip_packet, NET_ETH_PTYPE_IP);

static NPF_RULE(reject_big_pkts, NET_DROP, minsize_201);
static NPF_RULE(reject_non_ip, NET_DROP, not_ip_packet);

void install_my_filter(void) {
    npf_append_recv_rule(&reject_big_pkts);
    npf_append_recv_rule(&reject_non_ip);
    npf_append_recv_rule(&npf_default_ok);
}
```

This example assigns priorities to different network traffic. It gives network control priority (`NET_PRIORITY_NC`) to the `ptp` packets, critical applications priority (`NET_PRIORITY_CA`) to the in-

ternet traffic of version 6, excellent effort (NET_PRIORITY_EE) for internet protocol version 4 traffic, and the lowest background priority (NET_PRIORITY_BK) to lldp and arp.

Priority rules are only really useful if multiple traffic class queues are enabled in the project configuration CONFIG_NET_TC_RX_COUNT. The mapping from the priority of the packet to the traffic class queue is in accordance with the standard 802.1Q and depends on the CONFIG_NET_TC_RX_COUNT.

```
static NPF_ETH_TYPE_MATCH(is_arp, NET_ETH_PTYPE_ARP);
static NPF_ETH_TYPE_MATCH(is_lldp, NET_ETH_PTYPE_LLDP);
static NPF_ETH_TYPE_MATCH(is_ptp, NET_ETH_PTYPE_PTP);
static NPF_ETH_TYPE_MATCH(is_ipv4, NET_ETH_PTYPE_IP);
static NPF_ETH_TYPE_MATCH(is_ipv6, NET_ETH_PTYPE_IPV6);

static NPF_PRIORITY(priority_bk, NET_PRIORITY_BK, is_arp, is_lldp);
static NPF_PRIORITY(priority_ee, NET_PRIORITY_EE, is_ipv4);
static NPF_PRIORITY(priority_ca, NET_PRIORITY_CA, is_ipv6);
static NPF_PRIORITY(priority_nc, NET_PRIORITY_NC, is_ptp);

void install_my_filter(void) {
    npf_append_recv_rule(&priority_bk);
    npf_append_recv_rule(&priority_ee);
    npf_append_recv_rule(&priority_ca);
    npf_append_recv_rule(&priority_nc);
    npf_append_recv_rule(&npf_default_ok);
}
```

API Reference

Network Packet Filter API

Related code samples

- Network packet filterInstall network packet filter hooks.

Basic Filter Conditions

Ethernet Filter Conditions

Network Shell Network shell, and its companion shells provide helpers for figuring out network status, enabling/disabling features, and issuing commands like ping or DNS resolving. Note that net-shell should probably not be used in production code as it will require extra memory. See also *generic shell* for detailed shell information.

Note that by default both the enabled or disabled net-shell commands are available to the user. This helps user to discover what commands are available and how to enable them. This extra help can be turned off by disabling CONFIG_NET_SHELL_SHOW_DISABLED_COMMANDS option.

The following net-shell commands are implemented:

Table 10: net-shell commands

Command	Description
net allocs	Print network memory allocations. Only available if CONFIG_NET_DEBUG_NET_PKT_ALLOC is set.
net arp	Print information about IPv4 ARP cache. Only available if CONFIG_NET_ARP is set in IPv4 enabled networks.

continues on next page

Table 10 – continued from previous page

Command	Description
net bridge	Print information and manipulate Ethernet bridges. Only available if CONFIG_NET_ETHERNET_BRIDGE_SHELL is set.
net capture	Monitor network traffic See *Monitor Network Traffic* for details.
net cm	Connection manager shell. Only available if CONFIG_NET_CONNECTION_MANAGER is set.
net conn	Print information about network connections.
net dhcpv4	Enable / disable DHCPv4 client or server support. Only available if CONFIG_NET_DHCPV4_SERVER or CONFIG_NET_DHCPV4 is set.
net dhcpv6	Enable / disable DHCPv6 client support. Only available if CONFIG_NET_DHCPV6 is set.
net dns	Show how DNS is configured. The command can also be used to resolve a DNS name. Only available if CONFIG_DNS_RESOLVER is set.
net events	Enable network event monitoring. Only available if CONFIG_NET_MGMT_EVENT_MONITOR is set.
net filter	View network packet filter rules. Only available if CONFIG_NET_PKT_FILTER is set.
net ftp	Connect to FTP server to transfer files. Only available if CONFIG_FTP_CLIENT is set.
net gptp	Print information about gPTP support. Only available if CONFIG_NET_GPTP is set.
net http	Show HTTP server information, or send HTTP GET/POST/PUT/DELETE requests. Only available if CONFIG_HTTP_SERVER or CONFIG_HTTP_CLIENT is set.
net iface	Print information about network interfaces.
net ipv4	Print IPv4 specific information and configuration. Only available if CONFIG_NET_IPV4 is set.
net ipv6	Print IPv6 specific information and configuration. Only available if CONFIG_NET_IPV6 is set.
net mem	Print information about network memory usage. The command will print more information if CONFIG_NET_BUF_POOL_USAGE is set.
net nbr	Print neighbor information. Only available if CONFIG_NET_IPV6 is set.
net ping	Ping a network host.
net pkt	Print low level network packet information for debugging purposes.
net pmtu	Print MTU path discovery information. Only available if CONFIG_NET_IPV6_PMTU or CONFIG_NET_IPV4_PMTU is set.
net ppp	Print Point-to-Point protocol information. Only available if CONFIG_NET_L2_PPP and CONFIG_NET_PPP are set.
net ptp	Print information about PTP support. Use net ptp <port> for detailed per-port view. Only available if CONFIG_PTP is set.
net qbv	Show and configure IEEE 802.1Qbv Time-Aware Shaper (TAS) information. Only available if CONFIG_NET_QBV is set.
net quic	Show and configure QUIC transport. Only available if CONFIG_QUIC is set.
net resume	Resume a network interface if network power management is enabled.
net route	Show IPv6 or IPv4 network routes. Only available if CONFIG_NET_IPV6_ROUTING or CONFIG_NET_IPV4_ROUTING is set.
net sockets	Show network socket information and statistics. Only available if CONFIG_NET_SOCKETS_OBJ_CORE and CONFIG_OBJ_CORE are set.
net ssh	SSH client support. Only available if CONFIG_SSH_CLIENT is set.
net sshd	SSH server support. Only available if CONFIG_SSH_SERVER is set.
net ssh_key	SSH key generation/removing/saving/loading support. Only available if CONFIG_SSH_CLIENT or CONFIG_SSH_SERVER is set.
net stats	Show network statistics.
net suspend	Suspend a network interface if network power management is enabled.
net tcp	Connect/send data/close TCP connection. Only available if CONFIG_NET_TCP is set.
net udp	Send UDP data directly from shell. Only available if CONFIG_NET_UDP is set.

continues on next page

Table 10 – continued from previous page

Command	Description
net virtual	Show and manipulate network virtual interfaces. Only available if `CONFIG_NET_L2_VIRTUAL` is set.
net vlan	Show Ethernet virtual LAN information. Only available if `CONFIG_NET_VLAN` is set.
net websocket	Print websocket information. Only available if `CONFIG_WEBSOCKET_CLIENT` is set.
net wg	Show WireGuard VPN information and setup VPNs. Only available if `CONFIG_WIREGUARD` is set.

The Wi-Fi shell provides commands to scan, connect, disconnect and configure Wi-Fi networks. Following Wi-Fi shell commands are implemented:

Table 11: wifi-shell commands

Command	Description
wifi <cmd>	Multiple commands for Wi-Fi network connection, disconnection, scanning and configuration. Only available if `CONFIG_NET_L2_WIFI_SHELL` is set.
wifi cred	Show/add/delete Wi-Fi network credentials. Only available if `CONFIG_WIFI_CREDENTIALS_SHELL` is set. See *Wi-Fi credentials Library* for details.

The TLS credentials shell provides commands to list, add, delete and retrieve TLS credential information either from a volatile or protected backend storage. Following TLS credentials shell commands are implemented. These commands are only available if `CONFIG_TLS_CREDENTIALS_SHELL` is set. See *TLS Credentials Shell* for details.

Table 12: tls-credentials-shell commands

Command	Description
cred buf	Buffer in credential data so it can be added.
cred add	Add a TLS credential.
cred del	Delete a TLS credential.
cred get	Retrieve the contents of a TLS credential.
cred list	List stored TLS credentials.

TLS Credentials Shell The TLS Credentials shell provides a command-line interface for managing installed TLS credentials.

Commands

Buffer Credential (`buf`) Buffer data incrementally into the credential buffer so that it can be added using the *Add Credential (add)* command.

Alternatively:

- Clear the credential buffer.
- Load credential directly to the credential buffer, ending with `Ctrl + c`.

Usage To append `<DATA>` to the credential buffer, use:

```
cred buf <DATA>
```

Use this as many times as needed to load the full credential into the credential buffer, then use the *Add Credential (add)* command to store it.

To load <DATA> directly to the credential buffer, use:

```
cred buf load
<DATA>
Ctrl + c
```

To clear the credential buffer, use:

```
cred buf clear
```

Arguments

Argument	Description
<DATA>	Text data to be appended to credential buffer. It can be either text, or base64-encoded binary. See *Add Credential (add)* and *Storage/Retrieval Formats* for details.

Add Credential (add) Add a TLS credential to the TLS Credential store.

Credential contents can be provided in-line with the call to `cred add`, or will otherwise be sourced from the credential buffer.

Usage To add a TLS credential using the data from the credential buffer, use:

```
cred add <SECTAG> <TYPE> <BACKEND> <FORMAT>
```

To add a TLS credential using data provided with the same command, use:

```
cred add <SECTAG> <TYPE> <BACKEND> <FORMAT> <DATA>
```

Arguments

Argument	Description
<SECTAG>	The sectag to use for the new credential. Can be any non-negative integer.
<TYPE>	The type of credential to add. See *Credential Types* for valid values.
<BACKEND>	Reserved. Must always be `DEFAULT` (case-insensitive).
<FORMAT>	Specifies the storage format of the provided credential. See *Storage/Retrieval Formats* for valid values.
<DATA>	If provided, this argument will be used as the credential data, instead of any data in the credential buffer. Can be either text, or base64-encoded binary.

Delete Credential (del) Delete a specified credential from the credential store.

Usage To delete a credential matching a specified sectag and credential type (if it exists), use:

```
cred del <SECTAG> <TYPE>
```

Arguments

Argument	Description
<SECTAG>	The sectag of the credential to delete. Can be any non-negative integer.
<TYPE>	The type of credential to delete. See *Credential Types* for valid values.

Get Credential Contents (get) Retrieve and print the contents of a specified credential.

Usage To retrieve and print a credential matching a specified sectag and credential type (if it exists), use:

```
cred get <SECTAG> <TYPE> <FORMAT>
```

Arguments

Argument	Description
<SECTAG>	The sectag of the credential to get. Can be any non-negative integer.
<TYPE>	The type of credential to get. See *Credential Types* for valid values.
<FORMAT>	Specifies the retrieval format for the provided credential. See *Storage/Retrieval Formats* for valid values.

List Credentials (list) List TLS credentials in the credential store.

Usage To list all available credentials, use:

```
cred list
```

To list all credentials with a specified sectag, use:

```
cred list <SECTAG>
```

To list all credentials with a specified credential type, use:

```
cred list any <TYPE>
```

To list all credentials with a specified credential type and sectag, use:

```
cred list <SECTAG> <TYPE>
```

Arguments

Argument	Description
<SECTAG>	Optional. If provided, only list credentials with this sectag. Pass any or omit to allow any sectag. Otherwise, can be any non-negative integer.
<TYPE>	Optional. If provided, only list credentials with this credential type. Pass any or omit to allow any credential type. Otherwise, see *Credential Types* for valid values.

Output The command outputs all matching credentials in the following (CSV-compliant) format:

```
<SECTAG>,<TYPE>,<DIGEST>,<STATUS>
```

Where:

Symbol	Value
<SECTAG>	The sectag of the listed credential. A non-negative integer.
<TYPE>	Credential type short-code (see *Credential Types* for details) of the listed credential.
<DIGEST>	A string digest representing the credential contents. The exact nature of this digest may vary depending on credentials storage backend, but currently for all backends this is a base64 encoded SHA256 hash of the raw credential contents (so different storage formats for essentially identical credentials will have different digests).
<STATUS>	Status code indicating success or failure with generating a digest of the listed credential. 0 if successful, negative error code specific to the storage backend otherwise. Lines for which status is not zero will be printed with error formatting.

After the list is printed, a final summary of the found credentials will be printed in the form:

```
<N> credentials found.
```

Where <N> is the number of credentials found, and is zero if none are found.

Credential Types The following keywords (case-insensitive) may be used to specify a credential type:

Keyword(s)	Meaning
CA_CERT, CA	A trusted CA certificate.
SERVER_CERT, SELF_CERT, CLIENT_CERT, CLIENT, SELF, SERV	Self or server certificate.
PRIVATE_KEY, PK	A private key.
PRE_SHARED_KEY, PSK	A pre-shared key.
PRE_SHARED_KEY_ID, PSK_ID	ID for pre-shared key.

Storage/Retrieval Formats The *tls_credentials* module treats stored credentials as arbitrary binary buffers.

For convenience, the TLS credentials shell offers four formats for providing and later retrieving these buffers using the shell.

These formats and their (case-insensitive) keywords are as follows:

Kw	Meaning	Behavior during storage (`cred add`)	Behavior during retrieval (`cred get`)
B	Credential is handled by shell as base64 and stored without NULL termination.	Data entered into shell will be decoded from base64 into raw binary before storage. No terminator will be appended.	Stored data will be encoded into base64 before being printed.
B	Credential is handled by shell as base64 and stored with NULL termination.	Data entered into shell will be decoded from base64 into raw binary and a NULL terminator will be appended before storage.	NULL terminator will be truncated from stored data before said data is encoded into base64 and then printed.
S	Credential is handled by shell as literal string and stored without NULL termination.	Text data entered into shell will be passed into storage as-written, without a NULL terminator.	Stored data will be printed as text. Non-printable characters will be printed as ?
S	Credential is handled by shell as literal string and stored with NULL-termination.	Text data entered into shell will be passed into storage as-written, with a NULL terminator.	NULL terminator will be truncated from stored data before said data is printed as text. Non-printable characters will be printed as ?

The `BIN` format can be used to install credentials of any type, since base64 can be used to encode any concievable binary buffer. The remaining three formats are provided for convenience in special use-cases.

For example:

- To install printable pre-shared-keys, use `STR` to enter the PSK without first encoding it. This ensures it is stored without a NULL terminator.
- To install DER-formatted X.509 certificates (or other raw-binary credentials, such as non-printable PSKs) base64-encode the binary and use the `BIN` format.
- To install PEM-formatted X.509 certificates or certificate chains, base64 encode the full PEM string (including new-lines and `----BEGIN X ----` / `----END X----` markers), and then use the `BINT` format to make sure the stored string is NULL-terminated. This is required because Zephyr does not support multi-line strings in the shell. Otherwise, the `STRT` format could be used for this purpose without base64 encoding. It is possible to use `BIN` instead if you manually encode a NULL terminator into the base64.

Time Sensitive Networking

generic Precision Time Protocol (gPTP)

- *Overview*
- *Supported features*
- *Supported hardware*
- *Enabling the stack*
- *Static timeReceiver operation*
- *Application interfaces*
- *Testing*

- *API Reference*

Overview This gPTP stack supports the protocol and procedures as defined in the IEEE 802.1AS-2011 standard (Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks).

Supported features The stack handles communications and state machines defined in the IEEE 802.1AS-2011 standard. Mandatory requirements for a full-duplex point-to-point link end-point, as defined in Annex A of the standard, are supported.

The stack is in principle capable of handling communications on multiple network interfaces (also defined as "ports" in the standard) and thus act as a 802.1AS bridge. However, this mode of operation has not been validated on the Zephyr OS.

The stack can also operate as a statically configured time receiver on networks that follow the IEEE 802.1AS automotive profile, where no Announce messages are exchanged. See *Static timeReceiver operation* below.

Supported hardware Although the stack itself is hardware independent, Ethernet frame timestamping support must be enabled in ethernet drivers.

Boards supported:

- frdm_k64f
- nucleo_h743zi
- nucleo_h745zi_q
- nucleo_f767zi
- sam_e70_xplained
- native_sim (only usable for simple testing, limited capabilities due to lack of hardware clock)
- qemu_x86 (emulated, limited capabilities due to lack of hardware clock)

Enabling the stack The following configuration option must me enabled in `prj.conf` file.

- `CONFIG_NET_GPTP`

Static timeReceiver operation Networks built after the IEEE 802.1AS automotive profile (AVnu "Automotive Ethernet AVB Functional and Interoperability Specification") use static port roles instead of the Best Master Clock Algorithm. A bridge on such a network transmits Sync and Follow_Up messages but no Announce messages, and is not required to answer Pdelay requests on its timeTransmitter ports. The default stack cannot synchronize to such a bridge: without a received Announce a port never reaches the time receiver ("slave") role.

Enabling `CONFIG_NET_GPTP_STATIC_TIME_RECEIVER` configures the node as a statically configured time receiver: BMCA and all Announce handling are bypassed, every port is pinned to the time receiver role, asCapable is forced so synchronization does not depend on the Pdelay measurement, and the local clock is disciplined from the received Sync and Follow_Up messages alone. The node never becomes grandmaster and never transmits Sync or Announce messages, even if `CONFIG_NET_GPTP_GM_CAPABLE` is set. This mirrors the linuxptp ptp4l automotive time receiver configuration (BMCA "noop", clientOnly, inhibit_announce, asCapable "true", ignore_source_id). The standards analog of disabling the BMCA this way is the external port configuration of IEEE 802.1AS-2020 (clause 10.3.14, derived from IEEE 1588-2019 clause 17.6.2).

Application interfaces The following Application Interfaces as defined in section 9 of the standard are available:

- `ClockSourceTime` interface (`gptp_clk_src_time_invoke()`)
- `ClockTargetPhaseDiscontinuity` interface (`gptp_register_phase_dis_cb()`)
- `ClockTargetEventCapture` interface (`gptp_event_capture()`)

Testing The stack has been informally tested using the OpenAVnu gPTP and Linux ptp4l daemons. The gPTP sample application from the Zephyr source distribution can be used for testing.

API Reference
gPTP support

Related code samples

- gPTPEnable gPTP support and monitor functionality using net-shell.

Network time representation in the network stack

API Reference
Network time representation.

Precision Time Protocol (PTP) time format

- *Overview*
- *API Reference*

Overview The PTP time struct can store time information in high precision format (nanoseconds). The extended timestamp format can store the time in fractional nanoseconds accuracy. The PTP time format is used in *generic Precision Time Protocol (gPTP)* implementation.

API Reference
PTP time

Related code samples

- gPTPEnable gPTP support and monitor functionality using net-shell.

Precision Time Protocol (PTP)

- *Overview*
- *Supported features*
- *Network transmission modes*
- *Supported Management messages*
- *Timestamping notes*

- *Supported hardware*
- *Enabling the stack*
- *Testing*
- *API Reference*

Overview PTP is a network protocol implemented in the application layer, used to synchronize clocks in a computer network. It's accurate up to less than a microsecond. The stack supports the protocol and procedures as defined in the IEEE 1588-2019 standard (IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems). It has multiple profiles, and can be implemented on top of L2 (Ethernet) or L3 (UDP/IPv4 or UDP/IPv6). Its accuracy is achieved by using hardware timestamping of the protocol packets.

Zephyr's implementation of PTP stack consist following items:

- PTP stack thread that handles incoming messages and events
- Integration with ptp_clock driver
- PTP stack initialization executed during system init

The implementation automatically creates PTP Ports (each PTP Port corresponds to unique interface).

Supported features Implementation of the stack doesn't support all features specified in the standard. In the table below all supported features are listed.

Table 13: Supported features

Feature	Supported
Ordinary Clock	yes
Boundary Clock	yes
Transparent Clock	
Management Node	
End to end delay mechanism	yes
Peer to peer delay mechanism	yes (two-step)
Multicast operation mode	yes
Hybrid operation mode	yes
Unicast operation mode	
Non-volatile storage	
UDP IPv4 transport protocol	yes
UDP IPv6 transport protocol	yes
IEEE 802.3 (Ethernet) transport protocol	yes
Hardware timestamping	yes
Software timestamping	
TIME_RECEIVER_ONLY PTP Instance	yes
TIME_TRANSMITTER_ONLY PTP Instance	

Network transmission modes The network transmission mode is selected with the `PTP_NETWORK_MODE` Kconfig choice:

- Multicast mode (`CONFIG_PTP_NETWORK_MODE_MULTICAST`, the default) is the standard PTP mode whereby all PTP messages are sent to the default multicast addresses. The implication is that every node receives the `Delay_Req` and `Delay_Resp` message pairs of all other nodes, thereby increasing the amount of traffic on the network. This is usually not a problem on smaller networks employing only a few timeReceivers.

- Hybrid mode (`CONFIG_PTP_NETWORK_MODE_HYBRID`) still sends `Announce`, `Sync` and `Follow_Up` messages to the default multicast addresses, but a timeReceiver sends its `Delay_Req` messages as unicast directly to the protocol address (IP address, or MAC address for the IEEE 802.3 transport) of the current timeTransmitter, which in turn responds with a unicast `Delay_Resp` to the requesting timeReceiver. This reduces the level of PTP traffic on the network, which can be a factor when scaling to larger networks employing many timeReceivers. Hybrid mode only requires the network to support multicast transmission from the timeTransmitter to the timeReceivers. The mode is compatible with the `hybrid_e2e` option of linuxptp and the `hybrid` network mode of sfptpd; unicast negotiation is not supported.

  If the timeTransmitter fails to answer `CONFIG_PTP_HYBRID_FALLBACK_ATTEMPTS` consecutive unicast `Delay_Req` messages, the PTP Port logs an error, reverts to multicast delay measurement and stays in multicast until a new timeTransmitter is selected. The fallback can be disabled with `CONFIG_PTP_NETWORK_MODE_HYBRID_NO_FALLBACK`, in which case the port always keeps sending unicast `Delay_Req` messages.

  Hybrid mode requires the End-to-End delay mechanism (`CONFIG_PTP_DELAY_MECHANISM_E2E`) and is supported on all transport protocols (UDP IPv4, UDP IPv6 and IEEE 802.3).

Supported Management messages Based on Table 59 from section 15.5.2.3 of the IEEE 1588-2019 following management TLVs are supported:

Table 14: Supported management message's IDs

Management_ID	Management_ID name	Allowed actions
0x0000	NULL_PTP_MANAGEMENT	GET SET COMMAND
0x0001	CLOCK_DESCRIPTION	GET
0x0002	USER_DESCRIPTION	GET
0x0003	SAVE_IN_NON_VOLATILE_STORAGE	•
0x0004	RESET_NON_VOLATILE_STORAGE	•
0x0005	INITIALIZE	•
0x0006	FAULT_LOG	•
0x0007	FAULT_LOG_RESET	•
0x2000	DEFAULT_DATA_SET	GET
0x2001	CURRENT_DATA_SET	GET
0x2002	PARENT_DATA_SET	GET
0x2003	TIME_PROPERTIES_DATA_SET	GET
0x2004	PORT_DATA_SET	GET
0x2005	PRIORITY1	GET SET
0x2006	PRIORITY2	GET SET
0x2007	DOMAIN	GET SET
0x2008	TIME_RECEIVER_ONLY	GET SET
0x2009	LOG_ANNOUNCE_INTERVAL	GET SET
0x200A	ANNOUNCE_RECEIPT_TIMEOUT	GET SET
0x200B	LOG_SYNC_INTERVAL	GET SET
0x200C	VERSION_NUMBER	GET SET
0x200D	ENABLE_PORT	COMMAND
0x200E	DISABLE_PORT	COMMAND
0x200F	TIME	GET SET
0x2010	CLOCK_ACCURACY	GET SET
0x2011	UTC_PROPERTIES	GET SET
0x2012	TRACEBILITY_PROPERTIES	GET SET
0x2013	TIMESCALE_PROPERTIES	GET SET
0x2014	UNICAST_NEGOTIATION_ENABLE	•
0x2015	PATH_TRACE_LIST	•
0x2016	PATH_TRACE_ENABLE	•

continues on next page

Table 14 – continued from previous page

Management_ID	Management_ID name	Allowed actions
0x2017	GRANDMASTER_CLUSTER_TABLE	•
0x2018	UNICAST_TIME_TRANSMITTER_TABLE	•
0x2019	UNICAST_TIME_TRANSMITTER_MAX_TABLE_SIZE	•
0x201A	ACCEPTABLE_TIME_TRANSMITTER_TABLE	•
0x201B	ACCEPTABLE_TIME_TRANSMITTER_TABLE_ENAB	•
0x201C	ACCEPTABLE_TIME_TRANSMITTER_MAX_TABLE	•
0x201D	ALTERNATE_TIME_TRANSMITTER	•
0x201E	ALTERNATE_TIME_OFFSET_ENABLE	•
0x201F	ALTERNATE_TIME_OFFSET_NAME	•
0x2020	ALTERNATE_TIME_OFFSET_MAX_KEY	•
0x2021	ALTERNATE_TIME_OFFSET_PROPERTIES	•
0x3000	EXTERNAL_PORT_CONFIGURATION_ENABLED	
0x3001	TIME_TRANSMITTER_ONLY	•
0x3002	HOLDOVER_UPGRADE_ENABLE	•
0x3003	EXT_PORT_CONFIG_PORT_DATA_SET	•
0x4000	TRANSPARENT_CLOCK_DEFAULT_DATA_SET	•
0x4001	TRANSPARENT_CLOCK_PORT_DATA_SET	•
0x4002	PRIMARY_DOMAIN	•
0x6000	DELAY_MECHANISM	GET
0x6001	LOG_MIN_PDELAY_REQ_INTERVAL	GET SET

Timestamping notes When RX hardware timestamps are unavailable or invalid, synchronization falls back to reading PHC time during receive processing. This can introduce additional jitter from software handling latency (packet path and scheduling) between frame arrival and PHC read.

This behavior is expected for L2 AF_PACKET paths without true driver-provided RX hardware timestamps.

For IEEE 802.3 transport, Sync is sent in two-step mode and Follow_Up is generated from TX timestamp callbacks. If a TX timestamp is missing or late, the stack logs a warning and skips Follow_Up for that Sync sequence, then continues normal Sync transmission on subsequent intervals (best-effort behavior).

Peer-to-peer delay measurement can be selected with `CONFIG_PTP_DELAY_MECHANISM_P2P`. The first supported P2P mode is two-step `Pdelay_Req` / `Pdelay_Resp` / `Pdelay_Resp_Follow_Up`. One-step `Pdelay_Resp` samples are rejected and logged for a later implementation.

Supported hardware Although the stack itself is hardware independent, Ethernet frame timestamping support must be enabled in ethernet drivers.

Boards supported:

- nucleo_h563zi
- nucleo_h743zi
- nucleo_h745zi_q
- nucleo_f767zi
- frdm_mcxn947
- native_sim (only usable for simple testing, limited capabilities due to lack of hardware clock)

Enabling the stack The following configuration option must me enabled in `prj.conf` file.

- `CONFIG_PTP`

Testing The stack has been informally tested using the Linux ptp4l daemons. It has also been tested with the nucleo_h563zi and frdm_mcxn947 board against a GPS clock, both with a direct Ethernet connection and with a PTP-capable switch in between. All tests were performed using the PTP sample application, with UDP IPv4, UDP IPv6, and IEEE 802.3 transport.

The following table summarizes the informal test matrix:

	IEEE 802.3	UDP IPv4	UDP IPv6
ptp4l daemons	yes	yes	yes
GPS Clock (direct link)	yes	yes	yes
PTP-capable switch	yes	yes	yes
GPS Clock + PTP-capable switch	yes	yes	yes

Peer-to-peer delay measurement is implemented and validated for ordinary-clock use. Boundary-clock operation is expected to share the same port-level Pdelay machinery but has not yet been validated on multi-port hardware.

The PTP sample application from the Zephyr source distribution can be used for testing.

API Reference
PTP support

Related code samples

- PTPEnable PTP support and monitor messages and events via logging.

zperf: Network Traffic Generator

- *Overview*
- *Sample Usage*
- *iperf3*
- *Session Management*
- *Custom Data Upload*
- *Raw TX Mode*

Overview zperf is a shell utility which allows to generate network traffic in Zephyr. The tool may be used to evaluate network bandwidth.

zperf is compatible with iPerf 2.0.10 and newer. For compatibility with older versions, enable `CONFIG_NET_ZPERF_LEGACY_HEADER_COMPAT`. Alternatively, zperf can speak iperf3, see *iperf3*. A build speaks one of the two.

zperf can be enabled in any application, a dedicated sample is also present in Zephyr. See zperf sample application for details.

Sample Usage If Zephyr acts as a client, iPerf must be executed in server mode. For example, the following command line must be used for UDP testing:

```
$ iperf -s -l 1K -u -V -B 2001:db8::2
```

For TCP testing, the command line would look like this:

```
$ iperf -s -l 1K -V -B 2001:db8::2
```

In the Zephyr console, zperf can be executed as follows:

```
zperf udp upload 2001:db8::2 5001 10 1K 1M
```

A UDP baud rate of `0` sends as fast as possible, the same as `iperf -b 0`.

For TCP the zperf command would look like this:

```
zperf tcp upload 2001:db8::2 5001 10 1K 1M
```

If the IP addresses of Zephyr and the host machine are specified in the config file, zperf can be started as follows:

```
zperf udp upload2 v6 10 1K 1M
```

or like this if you want to test TCP:

```
zperf tcp upload2 v6 10 1K 1M
```

If Zephyr is acting as a server, set the download mode as follows for UDP:

```
zperf udp download 5001
```

or like this for TCP:

```
zperf tcp download 5001
```

and in the host side, iPerf must be executed with the following command line if you are testing UDP:

```
$ iperf -l 1K -u -V -c 2001:db8::1 -p 5001
```

and this if you are testing TCP:

```
$ iperf -l 1K -V -c 2001:db8::1 -p 5001
```

iPerf output can be limited by using the -b option if Zephyr is not able to receive all the packets in orderly manner.

iperf3 With `CONFIG_NET_ZPERF_IPERF3`, zperf interoperates with iperf3 instead of iPerf 2. The support is experimental. The zperf API and shell commands stay the same, and the default port becomes 5201, the iperf3 default.

When Zephyr acts as a server, enable the protocols to accept:

```
zperf tcp download
zperf udp download
```

An iperf3 client chooses TCP or UDP for each test and uses one port for both, with a TCP control connection either way. The two commands therefore share the listening port: each enables its own protocol, and a test asking for one that is not enabled is refused. On the host:

```
$ iperf3 -c 2001:db8::1
$ iperf3 -c 2001:db8::1 -u -b 10M
```

When Zephyr acts as a client, run `iperf3 -s` on the host, and in the Zephyr console:

```
zperf tcp upload 2001:db8::2 5201 10 1K
zperf udp upload 2001:db8::2 5201 10 1K 10M
```

A test runs over a single stream. Reverse (`-R`), bidirectional (`--bidir`) and parallel (`-P`) tests, and omitting the start of a test (`-O`), are refused with the message iperf3 prints for an option the server does not implement. One test runs at a time, and a second client is told that the server is busy. Multicast and `CONFIG_ZPERF_SESSION_PER_THREAD` are iPerf 2 features.

Points to keep in mind when comparing with iPerf 2:

- An iperf3 server counts TCP data until the client ends the test on the control connection, which can overtake the last of the data, so it may report slightly less than the client sent. An iPerf 2 server counts until the connection closes.
- For UDP, datagrams lost after the last one the server received are counted as lost in the zperf upload report, as with iPerf 2. iperf3 itself leaves them out.
- iperf3 does not report reordering, so an upload reports no out-of-order packets.
- The server reads only the header of each UDP datagram, relying on `ZSOCK_MSG_TRUNC` to learn its full length. A socket offload driver that ignores the flag makes it count too few bytes.
- A test takes a control connection and a data stream on each end, and closed connections linger in TIME_WAIT for a while after each test. Allow for them in `CONFIG_NET_MAX_CONTEXTS` and `CONFIG_NET_MAX_CONN`, as the sample's `overlay-iperf3.conf` does.

Session Management If `CONFIG_ZPERF_SESSION_PER_THREAD` option is set, then multiple upload sessions can be done at the same time if user supplies `-a` option when starting the upload. Each session will have their own work queue to run the test. The session test results can be viewed also after the tests have finished. The sessions can be started with `-w` option which then lets the worker threads to wait a start signal so that all the threads can be started at the same time. This will prevent the case where the zperf shell cannot run because it is running in lower priority than the already started session thread. If you have only one upload session, then the `-w` is not really needed.

Following zperf shell commands are available for session management:

zperf shell command	Description
`jobs`	Show currently active or finished sessions
`jobs all`	Show statistics of finished sessions
`jobs clear`	Clear finished session statistics
`jobs start`	Start all the waiting sessions

Example:

```
uart:~$ zperf udp upload -a -t 5 192.0.2.2 5001 10 1K 1M
Remote port is 5001
Connecting to 192.0.2.2
Duration:       10.00 s
Packet size:    1000 bytes
Rate:           1000 kbps
Starting...
```

(continues on next page)

(continued from previous page)

```
Rate:           1.00 Mbps
Packet duration 7 ms

uart:~$ zperf jobs all
No sessions sessions found
uart:~$ zperf jobs
           Thread    Remaining
Id  Proto  Priority  time (sec)
[1] UDP    5            4

Active sessions have not yet finished
-
Upload completed!
Statistics:             server  (client)
Duration:               30.01 s (30.01 s)
Num packets:            3799    (3799)
Num packets out order:  0
Num packets lost:       0
Jitter:                 63 us
Rate:                   1.01 Mbps       (1.01 Mbps)
Thread priority:        5
Protocol:               UDP
Session id:             1

uart:~$ zperf jobs all
-
Upload completed!
Statistics:             server  (client)
Duration:               30.01 s (30.01 s)
Num packets:            3799    (3799)
Num packets out order:  0
Num packets lost:       0
Jitter:                 63 us
Rate:                   1.01 Mbps       (1.01 Mbps)
Thread priority:        5
Protocol:               UDP
Session id:             1
Total 1 sessions done

uart:~$ zperf jobs clear
Cleared data from 1 sessions

uart:~$ zperf jobs
No active upload sessions
No finished sessions found
```

The `-w` option can be used like this to delay the startup of the jobs.

```
uart:~$ zperf tcp upload -a -t 6 -w 192.0.2.2 5001 10 1K
Remote port is 5001
Connecting to 192.0.2.2
Duration:       10.00 s
Packet size:    1000 bytes
Rate:           10 kbps
Waiting "zperf jobs start" command.
[01:06:51.392,288] <inf> net_zperf: [0] TCP waiting for start
```

(continues on next page)

(continued from previous page)

```
uart:~$ zperf udp upload -a -t 6 -w 192.0.2.2 5001 10 1K 10M
Remote port is 5001
Connecting to 192.0.2.2
Duration:       10.00 s
Packet size:    1000 bytes
Rate:           10000 kbps
Waiting "zperf jobs start" command.
Rate:           10.00 Mbps
Packet duration 781 us
[01:06:58.064,552] <inf> net_zperf: [0] UDP waiting for start

uart:~$ zperf jobs start
-
Upload completed!
-
Upload completed!

# Note that the output may be garbled as two threads printed
# output at the same time. Just print out the fresh listing
# like this.

uart:~$ zperf jobs all
-
Upload completed!
Statistics:             server  (client)
Duration:               9.99 s  (10.00 s)
Num packets:            11429   (11429)
Num packets out order:  0
Num packets lost:       0
Jitter:                 164 us
Rate:                   9.14 Mbps       (9.14 Mbps)
Thread priority:        6
Protocol:               UDP
Session id:             0
-
Upload completed!
Duration:               10.00 s
Num packets:            15487
Num errors:             0 (retry or fail)
Rate:                   12.38 Mbps
Thread priority:        6
Protocol:               TCP
Session id:             0
Total 2 sessions done
```

Custom Data Upload zperf supports more advanced data upload profiling by setting a custom data source through `zperf_upload_params.data_loader`. This enables the generation of custom packet contents instead of sending a constant packet consisting solely of the z character. An example use case would be determining the maximum throughput of uploading data from an external flash memory chip.

Raw TX Mode zperf supports raw packet transmission mode for testing custom L2 frames or vendor-specific protocols. This mode bypasses UDP/TCP and sends raw packets directly using packet sockets.

To enable raw TX mode, set the following Kconfig options:

```
CONFIG_NET_SOCKETS_PACKET=y
CONFIG_NET_ZPERF_RAW_TX=y
```

The maximum header size can be configured with `CONFIG_NET_ZPERF_RAW_TX_MAX_HDR_SIZE` (default: 64 bytes).

Before using raw TX mode, TX injection mode must be enabled on the network interface:

```
uart:~$ net iface txinjection 1 on
```

The raw TX upload command syntax is:

```
zperf raw upload [-a] <if_index> <header_hex> [<duration_sec>] [<packet_size>]␣
↪[<rate_kbps>]
```

Where:

- `-a`: Optional async mode flag
- `if_index`: Network interface index (e.g., 1)
- `header_hex`: User-provided header as hex string (vendor metadata + frame header)
- `duration_sec`: Test duration in seconds (default: 1)
- `packet_size`: Total packet size including header (default: 256)
- `rate_kbps`: Target rate in Kbps, supports K/M suffixes (default: 10). `0` sends as fast as possible.

Example for sending raw 802.11 frames with vendor metadata:

```
uart:~$ zperf raw upload 1␣
↪12345678000400030000000088000000ffffffffffa06960e35215a06960e3521500000000aaaa030000000800␣
↪10 1024 50M
```

The header hex string contains:

- Vendor metadata (first 12 bytes in this example)
- 802.11 QoS Data header with LLC/SNAP

The payload (filled with z characters) is automatically appended to reach the specified packet size.

Connection Manager

Overview Connection Manager is a collection of optional Zephyr features that aim to allow applications to monitor and control connectivity (access to IP-capable networks) with minimal concern for the specifics of underlying network technologies.

Using Connection Manager, applications can use a single abstract API to control network association and monitor Internet access, and avoid excessive use of technology-specific boilerplate.

This allows an application to potentially support several very different connectivity technologies (for example, Wi-Fi and LTE) with a single codebase.

Applications can also use Connection Manager to generically manage and use multiple connectivity technologies simultaneously.

Structure Connection Manager is split into the following two subsystems:

- *Connectivity monitoring* (header file `include/zephyr/net/conn_mgr_monitoring.h`) monitors all available *Zephyr network interfaces (ifaces)* and triggers *network management* events indicating when IP connectivity is gained or lost.
- *Connectivity control* (header file `include/zephyr/net/conn_mgr_connectivity.h`) provides an abstract API for controlling iface network association.

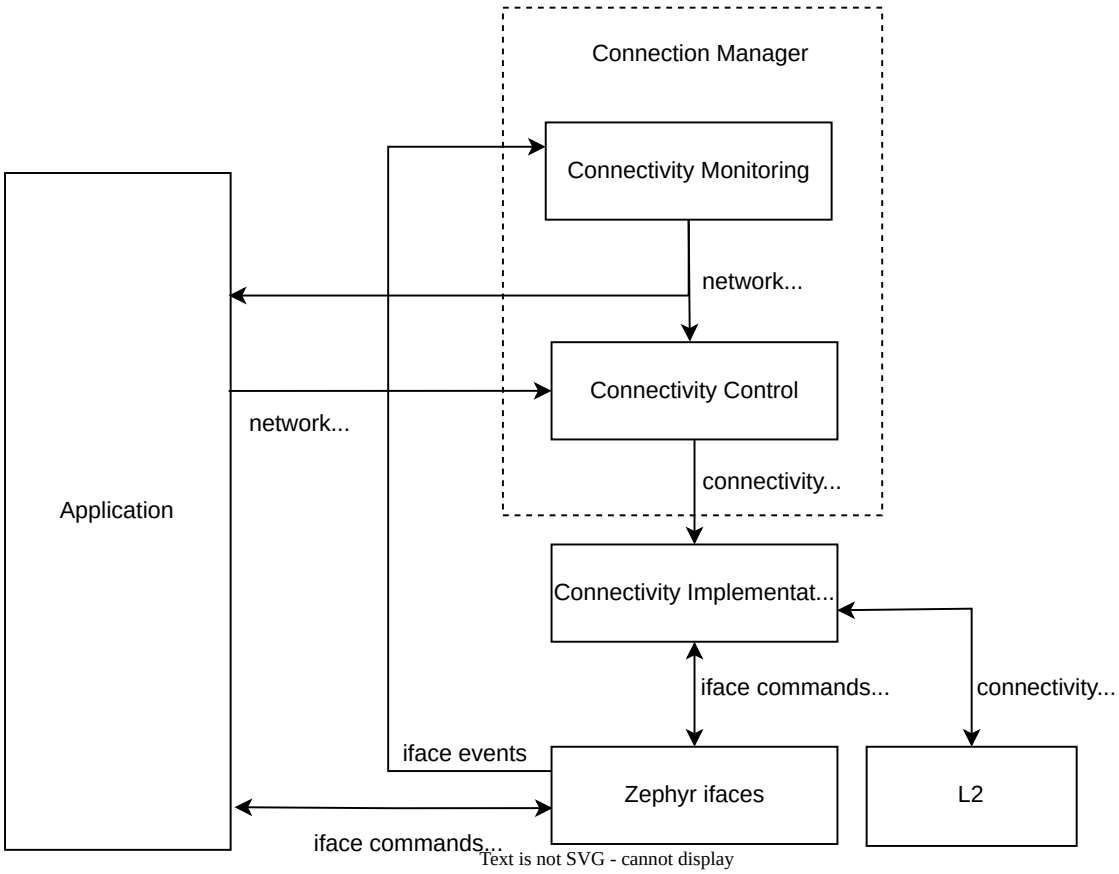


Fig. 10: A simplified view of how Connection Manager integrates with Zephyr and the application.

See *here* for a more detailed version.

Connectivity monitoring Connectivity monitoring tracks all available ifaces (whether or not they support *Connectivity control*) as they transition through various *operational states* and acquire or lose assigned IP addresses.

Each available iface is considered ready if it meets the following criteria:

- The iface is admin-up
  - This means the iface has been instructed to become operational-up (ready for use). This is done by a call to `net_if_up()`.
- The iface is oper-up
  - This means the interface is completely ready for use; It is online, and if applicable, has associated with a network.
  - See *Network interface state management* for details.
- The iface has at least one assigned IP address
  - Both IPv4 and IPv6 addresses are acceptable. This condition is met as soon as one or both of these is assigned.

  - See *Network Interface* for details on iface IP assignment.
- The iface has not been ignored
  - Ignored ifaces are always treated as unready.
  - See *Ignoring ifaces* for more details.

> **Note**
>
> Typically, iface state and IP assignment are updated either by the iface's *L2 implementation* or bound *connectivity implementation*.
>
> See *Implement iface state reporting* for details.

A ready iface ceases to be ready the moment any of the above conditions is lost.

When at least one iface is ready, the `NET_EVENT_L4_CONNECTED` *network management* event is triggered, and IP connectivity is said to be ready.

Afterwards, ifaces can become ready or unready without firing additional events, so long as there always remains at least one ready iface.

When there are no longer any ready ifaces left, the `NET_EVENT_L4_DISCONNECTED` *network management* event is triggered, and IP connectivity is said to be unready.

> **Note**
>
> Connection Manager also fires the following more specific `CONNECTED` / `DISCONNECTED` events:
>
> - `NET_EVENT_L4_IPV4_CONNECTED`
> - `NET_EVENT_L4_IPV4_DISCONNECTED`
> - `NET_EVENT_L4_IPV6_CONNECTED`
> - `NET_EVENT_L4_IPV6_DISCONNECTED`
>
> These are similar to `NET_EVENT_L4_CONNECTED` and `NET_EVENT_L4_DISCONNECTED`, but specifically track whether IPv4- and IPv6-capable ifaces are ready.

Usage Connectivity monitoring is enabled if the `CONFIG_NET_CONNECTION_MANAGER` Kconfig option is enabled.

To receive connectivity updates, create and register a listener for the `NET_EVENT_L4_CONNECTED` and `NET_EVENT_L4_DISCONNECTED` *network management* events:

```
/* Callback struct where the callback will be stored */
struct net_mgmt_event_callback l4_callback;

/* Callback handler */
static void l4_event_handler(struct net_mgmt_event_callback *cb,
                             uint32_t event, struct net_if *iface)
{
        if (event == NET_EVENT_L4_CONNECTED) {
                LOG_INF("Network connectivity gained!");
        } else if (event == NET_EVENT_L4_DISCONNECTED) {
                LOG_INF("Network connectivity lost!");
        }

        /* Otherwise, it's some other event type we didn't register for. */
```

(continues on next page)

(continued from previous page)

```
}

/* Call this before Connection Manager monitoring initializes */
static void my_application_setup(void)
{
        /* Configure the callback struct to respond to (at least) the L4_CONNECTED
         * and L4_DISCONNECTED events.
         *
         *
         * Note that the callback may also be triggered for events other than␣
↪those specified here!
         * (See the net_mgmt documentation)
         */
        net_mgmt_init_event_callback(
                &l4_callback, l4_event_handler,
                NET_EVENT_L4_CONNECTED | NET_EVENT_L4_DISCONNECTED
        );

        /* Register the callback */
        net_mgmt_add_event_callback(&l4_callback);
}
```

You can also use `NET_MGMT_REGISTER_EVENT_HANDLER` to register your callback handler at compile time, instead of at runtime. That way, you can ensure your callback is registered before Connection Manager monitoring initializes.

```
static void l4_event_handler(uint64_t event, struct net_if *iface, void *info,
                             size_t info_length, void *user_data)
{
        if (event == NET_EVENT_L4_CONNECTED) {
                LOG_INF("Network connectivity gained!");
        } else if (event == NET_EVENT_L4_DISCONNECTED) {
                LOG_INF("Network connectivity lost!");
        }

        /* Otherwise, it's some other event type we didn't register for. */
}

NET_MGMT_REGISTER_EVENT_HANDLER(l4_callback, l4_event_handler,
                                NET_EVENT_L4_CONNECTED |␣
↪NET_EVENT_L4_DISCONNECTED, NULL);
```

See *Listening to network events* for more details on listening for net_mgmt events.

> **Note**
>
> To avoid missing initial connectivity events, you should register your listener(s) before Connection Manager monitoring initializes. See *Avoiding missed notifications* for strategies to ensure this.

Avoiding missed notifications Connectivity monitoring may trigger events immediately upon initialization.

If your application registers its event listeners after connectivity monitoring initializes, it is possible to miss this first wave of events, and not be informed the first time network connectivity is gained.

If this is a concern, your application should *register its event listeners* before connectivity monitoring initializes.

Connectivity monitoring initializes using the `SYS_INIT` `APPLICATION` initialization priority specified by the `CONFIG_NET_CONNECTION_MANAGER_MONITOR_PRIORITY` Kconfig option.

You can register your callbacks before this initialization by using `SYS_INIT` with an earlier initialization priority than this value, for instance priority 0:

```
static int my_application_setup(void)
{
        /* Register callbacks here */
        return 0;
}

SYS_INIT(my_application_setup, APPLICATION, 0);
```

If this is not feasible, you can instead request that connectivity monitoring resend the latest connectivity events at any time by calling `conn_mgr_mon_resend_status()`:

```
static void my_late_application_setup(void)
{
  /* Register callbacks here */

  /* Once done, request that events be re-triggered */
  conn_mgr_mon_resend_status();
}
```

Ignoring ifaces Applications can request that ifaces be ignored by Connection Manager by calling `conn_mgr_ignore_iface()` with the iface to be ignored.

Alternatively, an entire *L2 implementation* can be ignored by calling `conn_mgr_ignore_l2()`.

This has the effect of individually ignoring all the ifaces using that *L2 implementation*.

While ignored, the iface is treated by Connection Manager as though it were unready for network traffic, no matter its actual state.

This may be useful, for instance, if your application has configured one or more ifaces that cannot (or for whatever reason should not) be used to contact the wider Internet.

Bulk convenience functions optionally skip ignored ifaces.

See `conn_mgr_ignore_iface()` and `conn_mgr_watch_iface()` for more details.

Connectivity monitoring API Include header file `include/zephyr/net/conn_mgr_monitoring.h` to access these.

Connection Manager API

Connectivity control Many network interfaces require a network association procedure to be completed before being usable.

For such ifaces, connectivity control can provide a generic API to request network association (`conn_mgr_if_connect()`) and disassociation (`conn_mgr_if_disconnect()`). Network interfaces implement support for this API by *binding themselves to a connectivity implementation*.

Using this API, applications can associate with networks with minimal technology-specific boilerplate.

Connectivity control also provides the following additional features:

- Standardized *persistence and timeout* behaviors during association.

- *Bulk functions* for controlling the admin state and network association of all available ifaces simultaneously.
- Optional *convenience automations* for common connectivity actions.

Basic operation The following sections outline the basic operation of Connection Manager's connectivity control.

Binding Before an iface can be commanded to associate or disassociate using Connection Manager, it must first be bound to a *connectivity implementation*. Binding is performed by the provider of the iface, not by the application (see *Binding an iface to an implementation*), and can be thought of as an extension of the iface declaration.

Once an iface is bound, all connectivity commands passed to it (such as `conn_mgr_if_connect()` or `conn_mgr_if_disconnect()`) will be routed to the corresponding implementation function in the connectivity implementation.

> **Note**
>
> To avoid inconsistent behavior, all connectivity implementations must adhere to the *implementation guidelines*.

Connecting Once a bound iface is admin-up (see *Network interface state management*), `conn_mgr_if_connect()` can be called to cause it to associate with a network.

If association succeeds, the connectivity implementation will mark the iface as operational-up (see *Network interface state management*).

If association fails unrecoverably, the *fatal error event* will be triggered.

You can configure an optional *timeout* for this process.

> **Note**
>
> The `conn_mgr_if_connect()` function is intentionally minimalistic, and does not take any kind of configuration. Each connectivity implementation should provide a way to pre-configure or automatically configure any required association settings or credentials. See *Allow connectivity pre-configuration* for details.

Connection loss If connectivity is lost due to external factors, the connectivity implementation will mark the iface as operational-down.

Depending on whether *persistence* is set, the iface may then attempt to reconnect.

Manual disconnection The application can also request that connectivity be intentionally abandoned by calling `conn_mgr_if_disconnect()`.

In this case, the connectivity implementation will disassociate the iface from its network and mark the iface as operational-down (see *Network interface state management*). A new connection attempt will not be initiated, regardless of whether persistence is enabled.

Timeouts and Persistence Connection Manager requires that all connectivity implementations support the following standard key features:

- *Connection timeouts*
- *Connection persistence*

These features describe how ifaces should behave during connect and disconnect events. You can individually set them for each iface.

> **Note**
>
> It is left to connectivity implementations to successfully and accurately implement these two features as described below. See *Implementing timeouts and persistence* for more details from the connectivity implementation perspective.

The Connection Manager also implements the following optional feature:

- *Interface idle timeouts*

> **Note**
>
> The only requirement on the connectivity implementation to implement idle timeouts is to call `conn_mgr_if_used()` each time the interface is used.

Connection Timeouts When `conn_mgr_if_connect()` is called on an iface, a connection attempt begins.

The connection attempt continues indefinitely until it succeeds, unless a timeout has been specified for the iface (using `conn_mgr_if_set_timeout()`).

In that case, the connection attempt will be abandoned if the timeout elapses before it succeeds. If this happens, the *timeout event* is raised.

Interface Idle Timeout The connection manager enables users to apply an inactivity timeout on an interface (`conn_mgr_if_set_idle_timeout()`). Once connected, if the interface goes for the configured number of seconds without any activity, the interface is automatically disconnected. If this happens, the *idle timeout event* is raised. An idle timeout is considered an unintentional connection loss for the purposes of *Connection persistence*.

Connection Persistence Each iface also has a connection persistence setting that you can enable or disable by setting the `CONN_MGR_IF_PERSISTENT` flag with `conn_mgr_binding_set_flag()`.

This setting specifies how the iface should handle unintentional connection loss.

If persistence is enabled, any unintentional connection loss will initiate a new connection attempt, with a new timeout if applicable.

Otherwise, the iface will not attempt to reconnect.

> **Note**
>
> Persistence not does affect connection attempt behavior. Only the timeout setting affects this.
>
> For instance, if a connection attempt on an iface times out, the iface will not attempt to reconnect, even if it is persistent.
>
> Conversely, if there is not a specified timeout, the iface will try to connect forever until it succeeds, even if it is not persistent.
>
> See *Persistence during connection attempts* for the equivalent implementation guideline.

Control events Connectivity control triggers *network management* events to inform the application of important state changes.

See *Trigger connectivity control events* for the corresponding connectivity implementation guideline.

Fatal Error The `NET_EVENT_CONN_IF_FATAL_ERROR` event is raised when an iface encounters an error from which it cannot recover (meaning any subsequent attempts to associate are guaranteed to fail, and all such attempts should be abandoned).

Handlers of this event will be passed a pointer to the iface for which the fatal error occurred. Individual connectivity implementations may also pass an application-specific data pointer.

Timeout The `NET_EVENT_CONN_IF_TIMEOUT` event is raised when an *iface association* attempt *times out*.

Handlers of this event will be passed a pointer to the iface that timed out attempting to associate.

Idle Timeout The `NET_EVENT_CONN_IF_IDLE_TIMEOUT` event is raised when an interface is considered *inactive*.

Handlers of this event will be passed a pointer to the iface that timed out attempting to associate.

Listening for control events You can listen for control events as follows:

```
/* Declare a net_mgmt callback struct to store the callback */
struct net_mgmt_event_callback my_conn_evt_callback;

/* Declare a handler to receive control events */
static void my_conn_evt_handler(struct net_mgmt_event_callback *cb,
                                uint32_t event, struct net_if *iface)
{
        if (event == NET_EVENT_CONN_IF_TIMEOUT) {
                /* Timeout occurred, handle it */
        } else if (event == NET_EVENT_CONN_IF_FATAL_ERROR) {
                /* Fatal error occurred, handle it */
        }

        /* Otherwise, it's some other event type we didn't register for. */
}

int main()
{
        /* Configure the callback struct to respond to (at least) the␣
↪CONN_IF_TIMEOUT
         * and CONN_IF_FATAL_ERROR events.
         *
         * Note that the callback may also be triggered for events other than␣
↪those specified here!
         * (See the net_mgmt documentation)
         */

        net_mgmt_init_event_callback(
                &conn_mgr_conn_callback, conn_mgr_conn_handler,
                    NET_EVENT_CONN_IF_TIMEOUT | NET_EVENT_CONN_IF_FATAL_ERROR
        );

        /* Register the callback */
```

(continues on next page)

(continued from previous page)

```
        net_mgmt_add_event_callback(&conn_mgr_conn_callback);
        return 0;
}
```

See *Listening to network events* for more details on listening for net_mgmt events.

Automated behaviors There are a few actions related to connectivity that are (by default at least) performed automatically for the user.

Automatic admin-up

In Zephyr, ifaces are automatically taken admin-up (see *Network interface state management* for details on iface states) during initialization.

Applications can disable this behavior by setting the `NET_IF_NO_AUTO_START` interface flag with `net_if_flag_set()`.

Automatic connect

By default, Connection Manager will automatically connect any *bound* iface that becomes admin-up.

Applications can disable this by setting the `CONN_MGR_IF_NO_AUTO_CONNECT` connectivity flag with `conn_mgr_if_set_flag()`.

Automatic admin-down

By default, Connection Manager will automatically take any bound iface admin-down if it has given up on associating.

Applications can disable this for all ifaces by disabling the `CONFIG_NET_CONNECTION_MANAGER_AUTO_IF_DOWN` Kconfig option, or for individual ifaces by setting the `CONN_MGR_IF_NO_AUTO_DOWN` connectivity flag with `conn_mgr_if_set_flag()`.

Connectivity control API Include header file `include/zephyr/net/conn_mgr_connectivity.h` to access these.

Connection Manager Connectivity API

Bulk API Connectivity control provides several bulk functions allowing all ifaces to be controlled at once.

You can restrict these functions to operate only on non-*ignored* ifaces if desired.

Include header file `include/zephyr/net/conn_mgr_connectivity.h` to access these.

Connection Manager Connectivity Bulk API

Connectivity Implementations

Overview Connectivity implementations are technology-specific modules that allow specific Zephyr ifaces to support *Connectivity Control*. They are responsible for translating generic *connectivity control API* calls into hardware-specific operations. They are also responsible for implementing standardized *persistence and timeout* behaviors.

See the *implementation guidelines* for details on writing conformant connectivity implementations.

Architecture The *implementation API* allows connectivity implementations to be *defined* at build time using `CONN_MGR_CONN_DEFINE`.

This creates a static instance of the `conn_mgr_conn_impl` struct, which then stores a reference to the passed in `conn_mgr_conn_api` struct (which should be populated with implementation callbacks).

Once defined, you can reference implementations by name and bind them to any unbound iface using `CONN_MGR_BIND_CONN`. Make sure not to accidentally bind two connectivity implementations to a single iface.

Once the iface is bound, *connectivity control API* functions can be called on the iface, and they will be translated to the corresponding implementation functions in `conn_mgr_conn_api`.

Binding an iface does not directly modify its `iface struct`.

Instead, an instance of `conn_mgr_conn_binding` is created and appended an internal *iterable section*.

This binding structure will contain a reference to the bound iface, the connectivity implementation it is bound to, as well as a pointer to a per-iface *context pointer*.

This iterable section can then be iterated over to find out what (if any) connectivity implementation has been bound to a given iface. This search process is used by most of the functions in the *connectivity control API*. As such, these functions should be called sparingly, due to their relatively high search cost.

A single connectivity implementation may be bound to multiple ifaces. See *Do not instance implementations* for more details.

Context Pointer Since a single connectivity implementation may be shared by several Zephyr ifaces, each binding instantiates a context container (of *configurable type*) unique to that binding. Each binding is then instantiated with a reference to that container, which implementations can then use to access per-iface state information.

See also *Do not access bindings without locking them* and *Do not instance implementations*.

Defining an implementation A connectivity implementation may be defined as follows:

```
/* Create the API implementation functions */
int my_connect_impl(struct conn_mgr_conn_binding *const binding) {
        /* Cause your underlying technology to associate */
}
int my_disconnect_impl(struct conn_mgr_conn_binding *const binding) {
        /* Cause your underlying technology to disassociate */
}
void my_init_impl(struct conn_mgr_conn_binding *const binding) {
        /* Perform any required initialization for your underlying technology */
}

/* Declare the API struct */
static struct conn_mgr_conn_api my_impl_api = {
        .connect = my_connect_impl,
        .disconnect = my_disconnect_impl,
        .init = my_init_impl,
        /* ... so on */
};
```

(continues on next page)

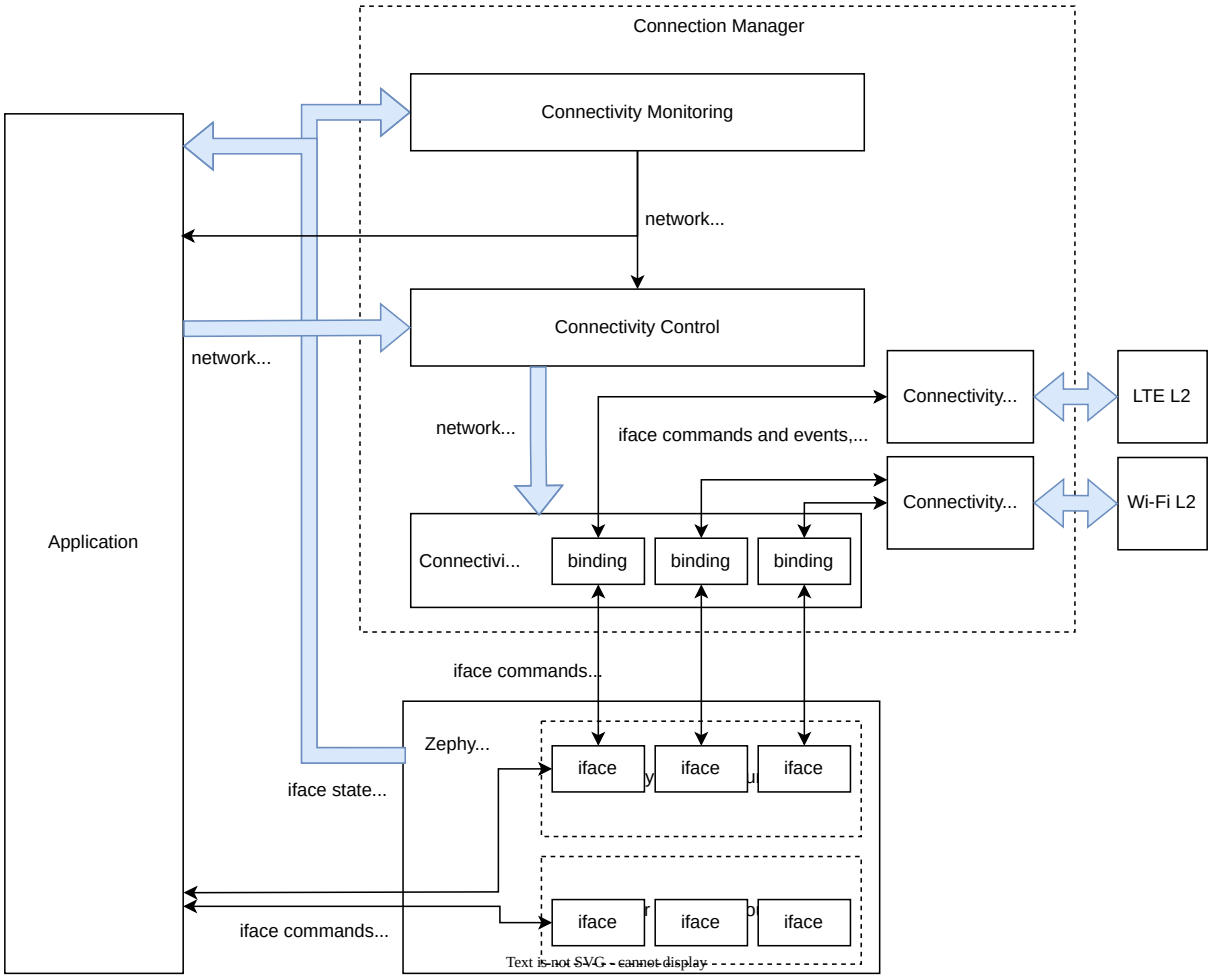


Fig. 11: A detailed view of how Connection Manager integrates with Zephyr and the application. See *here* for a simplified version.

(continued from previous page)

```
/* Define the implementation (named MY_CONNECTIVITY_IMPL) */
CONN_MGR_CONN_DEFINE(MY_CONNECTIVITY_IMPL, &my_impl_api);
```

> **Note**
>
> This does not work unless you also *declare the context pointer type*.

Declaring an implementation publicly Once defined, you can make a connectivity implementation available to other compilation units by declaring it (in a header file) as follows:

Listing 8: `my_connectivity_header.h`

```
CONN_MGR_CONN_DECLARE_PUBLIC(MY_CONNECTIVITY_IMPL);
```

The header file that contains this declaration must be included in any compilation units that need to reference the implementation.

Declaring a context type For `CONN_MGR_CONN_DEFINE` to work, you must declare a corresponding context pointer type. This is because all connectivity bindings contain a *Context Pointer* of their associated context pointer type.

If you are using `CONN_MGR_CONN_DECLARE_PUBLIC`, declare this type alongside the declaration:

Listing 9: `my_connectivity_impl.h`

```
#define MY_CONNECTIVITY_IMPL_CTX_TYPE struct my_context_type *
CONN_MGR_CONN_DECLARE_PUBLIC(MY_CONNECTIVITY_IMPL);
```

Then, make sure to include the header file before calling `CONN_MGR_CONN_DEFINE`:

Listing 10: `my_connectivity_impl.c`

```
#include "my_connectivity_impl.h"

CONN_MGR_CONN_DEFINE(MY_CONNECTIVITY_IMPL, &my_impl_api);
```

Otherwise, it is sufficient to simply declare the context pointer type immediately before the call to `CONN_MGR_CONN_DEFINE`:

```
#define MY_CONNECTIVITY_IMPL_CTX_TYPE struct my_context_type *
CONN_MGR_CONN_DEFINE(MY_CONNECTIVITY_IMPL, &my_impl_api);
```

> **Note**
>
> Naming is important. Your context pointer type declaration must use the same name as your implementation declaration, but with `_CTX_TYPE` appended.
>
> In the previous example, the context type is named `MY_CONNECTIVITY_IMPL_CTX_TYPE`, because `MY_CONNECTIVITY_IMPL` was used as the connectivity implementation name.

If your connectivity implementation does not need a context pointer, simply declare the type as void:

```
#define MY_CONNECTIVITY_IMPL_CTX_TYPE void *
```

Binding an iface to an implementation A defined connectivity implementation may be bound to an iface by calling `CONN_MGR_BIND_CONN` anywhere after the iface's device definition:

```
/* Define an iface */
NET_DEVICE_INIT(my_iface,
        /* ... the specifics here don't matter ... */
);

/* Now bind MY_CONNECTIVITY_IMPL to that iface --
 * the name used should match with the above
 */
CONN_MGR_BIND_CONN(my_iface, MY_CONNECTIVITY_IMPL);
```

Connectivity implementation guidelines Rather than implement all features centrally, Connection Manager relies on each connectivity implementation to implement many behaviors and features individually.

This approach allows Connection Manager to remain lean, and allows each connectivity implementation to choose the most appropriate approach to these behaviors for itself. However, it relies on trust that all connectivity implementations will faithfully implement the features that have been delegated to them.

To maintain consistency between all connectivity implementations, observe the following guidelines when writing your own implementation:

Completely implement timeout and persistence All connectivity implementations must offer complete support for *timeout and persistence*, such that a user can disable or enable these features, regardless of the inherent behavior of the underlying technology. In other words, no matter how the underlying technology behaves, your implementation must make it appear to the end user to behave exactly as specified in the *Timeouts and Persistence* section.

See *Implementing timeouts and persistence* for a detailed technical discussion on implementing timeouts and persistence.

Conform to API specifications Each `implementation API function` you implement should behave as-described in the corresponding connectivity control API function.

For example, your implementation of `conn_mgr_conn_api.connect` should conform to the behavior described for `conn_mgr_if_connect()`.

Allow connectivity pre-configuration Connectivity implementations should provide means for applications to pre-configure all necessary connection parameters (for example, network SSID, or PSK, if applicable), before the call to `conn_mgr_if_connect()`. It should not be necessary to provide this information as part of, or following the `conn_mgr_if_connect()` call, although implementations *should await this information if it is not provided*.

Await valid connectivity configuration If network association fails because the application pre-configured invalid connection parameters, or did not configure connection parameters at all, this should be treated as a network failure.

In other words, the connectivity implementation should not give up on the connection attempt, even if valid connection parameters have not been configured.

Instead, the connectivity implementation should asynchronously wait for valid connection parameters to be configured, either indefinitely, or until the configured *connectivity timeout* elapses.

An exception to this is if the network interface has been configured as non-persistent and the connectivity implementation defines an implementation for `conn_mgr_conn_api.has_connection_config`.

In this case, to reduce power consumption and prevent unnecessary state transitions, `conn_mgr_if_connect()` will exit early without bringing the interface up if `conn_mgr_conn_api.has_connection_config` returns `false`. The `NET_EVENT_CONN_IF_NO_CONFIGURATION` event will be emitted to notify subscribers of the misconfiguration.

Implement iface state reporting All connectivity implementations must keep bound iface state up to date.

To be specific:

- Set the iface to dormant, carrier-down, or both during `binding init`.
  - See *Network interface state management* for details regarding iface carrier and dormant states.
- Update dormancy and carrier state so that the iface is non-dormant and carrier-up whenever (and only when) association is complete and connectivity is ready.
- Set the iface either to dormant or to carrier-down as soon as interruption of service is detected.
  - It is acceptable to gate this behind a small timeout (separate from the connection timeout) for network technologies where service is commonly intermittent.
- If the technology also handles IP assignment, ensure those IP addresses are *assigned to the iface*.

Note

iface state updates do not necessarily need to be performed directly by connectivity implementations.

For instance:

- IP assignment is not necessary if *DHCP* is used for the iface.
- The connectivity implementation does not need to update iface dormancy if the underlying *L2 implementation* already does so.

Do not use iface state as implementation state Zephyr ifaces may be accessed from other threads without respecting the binding mutex. As such, Zephyr iface state may change unpredictably during connectivity implementation callbacks.

Therefore, do not base implementation behaviors on iface state.

Keep iface state updated to reflect network availability, but do not read iface state for any purpose.

If you need to keep track of dormancy or IP assignment, use a separate state variable stored in the *context pointer*.

Remain non-interferent Connectivity implementations should not prevent applications from interacting directly with associated technology-specific APIs.

In other words, it should be possible for an application to directly use your underlying technology without breaking your connectivity implementation.

If exceptions to this are absolutely necessary, they should be constrained to specific API calls and should be documented.

> **Note**
>
> While connectivity implementations must not break, it is acceptable for implementations to have potentially unexpected behavior if applications attempt to directly control the association state.
>
> For instance, if an application directly instructs an underlying technology to disassociate, it would be acceptable for the connectivity implementation to interpret this as an unexpected connection loss and immediately attempt to re-associate.

Remain non-blocking All connectivity implementation callbacks should be non-blocking.

For instance, calls to `conn_mgr_conn_api.connect` should initiate a connection process and return immediately.

One exception is `conn_mgr_conn_api.init`, whose implementations are permitted to block.

However, bear in mind that blocking during this callback will delay system init, so still consider offloading time-consuming tasks to a background thread.

Make API immediately ready Connectivity implementations must be ready to receive API calls immediately after `conn_mgr_conn_api.init`.

For instance, a call to `conn_mgr_conn_api.connect` must eventually lead to an association attempt, even if called immediately after `conn_mgr_conn_api.init`.

If the underlying technology cannot be made ready for connect commands immediately when `conn_mgr_conn_api.init` is called, calls to `conn_mgr_conn_api.connect` must be queued in a non-blocking fashion, and then executed later when ready.

Do not store state information outside the context pointer Connection Manager provides a context pointer to each binding.

Connectivity implementations should store all state information in this context pointer.

The only exception is connectivity implementations that are meant to be bound to only a single iface. Such implementations may use statically declared state instead.

See also *Do not instance implementations*.

Access ifaces only through binding structs Do not use statically declared ifaces or externally acquire references to ifaces.

For example, do not use `net_if_get_default()` under the assumption that the bound iface will be the default iface.

Instead, always use the `iface pointer` provided by the relevant `binding struct`. See also *Do not access bindings without locking them*.

Make implementations optional at compile-time Connectivity implementations should provide a Kconfig option to enable or disable the implementation without affecting bound iface availability.

In other words, it should be possible to configure builds that include Connectivity Manager, as well as the iface that would have been bound to the implementation, but not the implementation itself, nor its binding.

Do not instance implementations Do not declare a separate connectivity implementation for every iface you are going to bind to.

Instead, bind one global connectivity implementation to all of your ifaces, and use the context pointer to store state relevant to individual ifaces.

See also *Do not access bindings without locking them* and *Access ifaces only through binding structs*.

Do not access bindings without locking them Bindings may be accessed and modified at random by multiple threads, so modifying or reading from a binding without first `locking it` may lead to unpredictable behavior.

This applies to all descendents of the binding, including anything in the *context container*.

Make sure to `unlock` the binding when you are done accessing it.

> **Note**
>
> A possible exception to this rule is if the resource in question is inherently thread-safe.
>
> However, be careful taking advantage of this exception. It may still be possible to create a race condition, for instance when accessing multiple thread-safe resources simultaneously.
>
> Therefore, it is recommended to simply always lock the binding, whether or not the resource being accessed is inherently thread-safe.

Do not disable built-in features Do not attempt to prevent the use of built-in features (such as *Timeouts and Persistence* or *Automated behaviors*).

All connectivity implementations must fully support these features. Implementations must not attempt to force certain features to be always enabled or always disabled.

Trigger connectivity control events Connectivity control *network management* events are not triggered automatically by Connection Manager.

Connectivity implementations must trigger these events themselves.

Trigger `NET_EVENT_CONN_CMD_IF_TIMEOUT` when a connection *timeout* occurs. See *Timeout* for details.

Trigger `NET_EVENT_CONN_IF_FATAL_ERROR` when a fatal (non-recoverable) connection error occurs. See *Fatal Error* for details.

See *Network Management* for details on firing network management events.

Implementing timeouts and persistence First, see *Timeouts and Persistence* for a high-level description of the expected behavior of timeouts and persistence.

Connectivity implementations must fully conform to that description, regardless of the behavior of the underlying connectivity technology.

Sometimes this means writing extra logic in the connectivity implementation to fake certain behaviors. The following sections discuss various common edge-cases and nuances and how to handle them.

Inherently persistent technologies If the underlying technology automatically attempts to reconnect or retry connection after connection loss or failure, the connectivity implementation must manually cancel such attempts when they are in conflict with timeout or persistence settings.

For example:

- If the underlying technology automatically attempts to reconnect after losing connection, and persistence is disabled for the iface, the connectivity implementation should immediately cancel this reconnection attempt.
- If a connection attempt times out on an iface whose underlying technology does not have a built-in timeout, the connectivity implementation must simulate a timeout by cancelling the connection attempt manually.

Technologiess that give up on connection attempts If the underlying technology has no mechanism to retry connection attempts, or would give up on them before the user-configured timeout, or would not reconnect after connection loss, the connectivity implementation must manually re-request connection to counteract these deviances.

- If your underlying technology is not persistent, you must manually trigger reconnect attempts when persistence is enabled.
- If your underlying technology does not support a timeout, you must manually cancel connection attempts if the timeout is enabled.
- If your underlying technology forces a timeout, you must manually trigger a new connection attempts if that timeout is shorter than the Connection Manager timeout.

Technologies with association retry Many underlying technologies do not usually associate in a single attempt.

Instead, these underlying technologies may need to make multiple back-to-back association attempts in a row, usually with a small delay.

In these situations, the connectivity implementation should treat this series of back-to-back association sub-attempts as a single unified connection attempt.

For instance, after a sub-attempt failure, persistence being disabled should not prevent further sub-attempts, since they all count as one single overall connection attempt. See also *Persistence during connection attempts*.

At which point a series of failed sub-attempts should be considered a failure of the connection attempt as a whole is up to each implementation to decide.

If the connection attempt crosses this threshold, but the configured timeout has not yet elapsed, or there is no timeout, sub-attempts should continue.

Persistence during connection attempts Persistence should not affect any aspect of implementation behavior during a connection attempt. Persistence should only affect whether or not connection attempts are automatically triggered after a connection loss.

The configured timeout should fully determine whether connection retry should be performed.

Implementation API Include header file `include/zephyr/net/conn_mgr_connectivity_impl.h` to access these.

Only for use by connectivity implementations.

Connection Manager Connectivity Implementation API

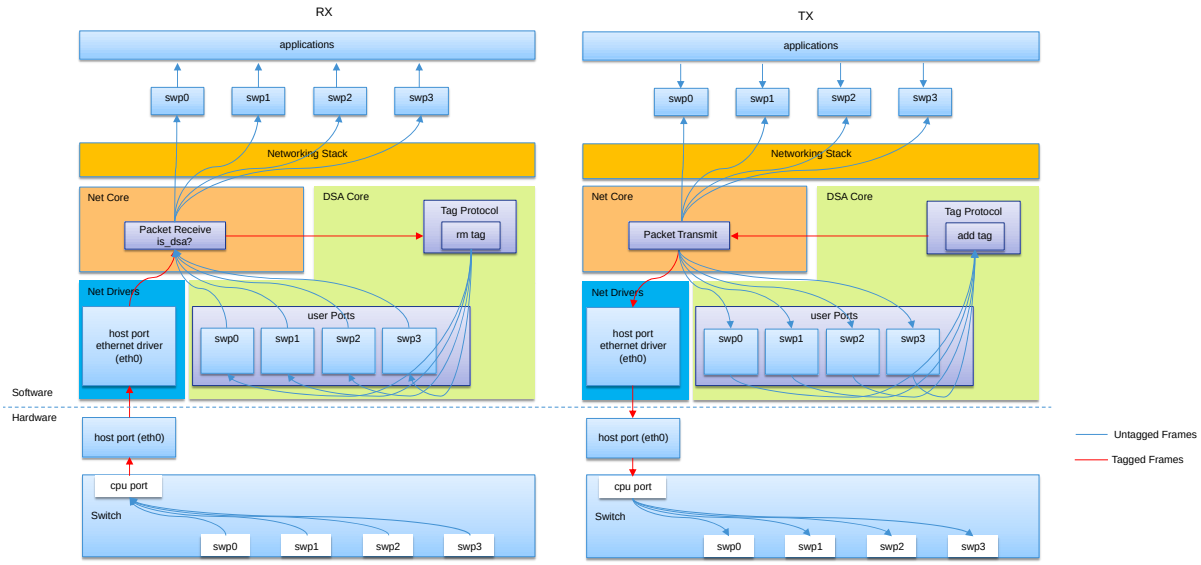
Distributed Switch Architecture (DSA)

- *DSA switch TX/RX process*
- *Host interface*

- *Switch interface*
- *Switch tagging protocols*
- *Networking stack process*
- *DSA device driver support*
- *Common pitfalls using DSA setups*
- *TODO work*

Distributed Switch Architecture (DSA) is not something new. It has been a mature subsystem in Linux for many years. This document just skips the background, terms and any knowledge related description, as user may find all of these in Linux DSA documentation.

DSA switch TX/RX process The DSA switch TX/RX process is as below.



Host interface Host interface network devices use regular and unmodified ethernet driver working as DSA conduit port, which manages the switch through processor.

Switch interface The switch interfaces are also exposed as standard ethernet interfaces in zephyr. The one connected to conduit port work as CPU port, and the others for user purpose work as user ports.

Switch tagging protocols Generally, switch tagging protocols are vendor specific. They all contain something which:

- identifies which port the Ethernet frame came from/should be sent to
- provides a reason why this frame was forwarded to the management interface

And tag on the packets to give them a switch frame header. But there are also tag-less case. It depends on the vendor.

Networking stack process In order to have the DSA subsystem process the Ethernet switch specific tagging protocol via conduit port.

For RX path, put `dsa_recv()` at beginning of `net_recv_data()` in `subsys/net/ip/net_core.c` to handle first for untagging and re-directing interface.

For TX path, the switch interfaces register as standard ethernet devices with `dsa_xmit()` as `ethernet_api->send`. The `dsa_xmit()` processes the tagging and re-directing to conduit port work.

DSA device driver support As the DSA core driver interacts with subsystems/drivers including MDIO, PHY and device tree to to support the common DSA setup and working process. The device driver support is much easy.

For device tree, the switch description should be following the `dts/bindings/dsa/dsa.yaml`.

For device driver, all needed to prepare are `dsa_api`, private data if has, and `dsa_port_config`. The macro functions could be utilized. And below is an example of i.MX NETC.

- `DSA_SWITCH_INST_INIT`
- `DSA_PORT_INST_INIT`

```
#define DSA_NETC_PORT_INST_INIT(port, n)
↪         \
        COND_CODE_1(DT_NUM_PINCTRL_STATES(port),
↪         \
                        (PINCTRL_DT_DEFINE(port);), (EMPTY))
↪         \
        struct dsa_netc_port_config dsa_netc_##n##_##port##_config = {
↪         \
                .pincfg = COND_CODE_1(DT_NUM_PINCTRL_STATES(port),
↪         \
                                (PINCTRL_DT_DEV_CONFIG_GET(port)), NULL),
↪         \
                .phy_mode = NETC_PHY_MODE(port),
↪         \
        };
↪         \
        struct dsa_port_config dsa_##n##_##port##_config = {
↪         \
                .mcfg = NET_ETH_MAC_DT_CONFIG_INIT(port),
↪         \
                .port_idx = DT_REG_ADDR(port),
↪         \
                .phy_dev = DEVICE_DT_GET_OR_NULL(DT_PHANDLE(port, phy_handle)),
↪         \
                .phy_mode = DT_PROP_OR(port, phy_connection_type, ""),
↪         \
                .ethernet_connection = DEVICE_DT_GET_OR_NULL(DT_PHANDLE(port,
↪ethernet)),    \
                .prv_config = &dsa_netc_##n##_##port##_config,
↪         \
        };
↪         \
        DSA_PORT_INST_INIT(port, n, &dsa_##n##_##port##_config)

#define DSA_NETC_DEVICE(n)
↪         \
        AT_NONCACHEABLE_SECTION_ALIGN(static netc_cmd_bd_t
```

(continues on next page)

(continued from previous page)

```
↪dsa_netc_##n##_cmd_bd[8],        \
                                     NETC_BD_ALIGN);
↪         \
        static struct dsa_netc_data dsa_netc_data_##n = {
↪         \
                .cmd_bd = dsa_netc_##n##_cmd_bd,
↪         \
        };
↪         \
        DSA_SWITCH_INST_INIT(n, &dsa_netc_api, &dsa_netc_data_##n,
↪DSA_NETC_PORT_INST_INIT);
```

Common pitfalls using DSA setups This is copied from Linux DSA documentation. It applies to zephyr too. Although conduit port and cpu port exposed as ethernet device in zephyr, they are not able to be used.

> **Note**
>
> Once a conduit network device is configured to use DSA (dev->dsa_ptr becomes non-NULL), and the switch behind it expects a tagging protocol, this network interface can only exclusively be used as a conduit interface. Sending packets directly through this interface (e.g.: opening a socket using this interface) will not make us go through the switch tagging protocol transmit function, so the Ethernet switch on the other end, expecting a tag will typically drop this frame.

TODO work Comparing to Linux, there are too much features to support in/based on zephyr DSA. But basically bridge layer should be supported. Then DSA could provide two options for users to use switch ports.

- Standalone mode: all user ports work as regular ethernet devices. No switching.
- Bridge mode: switch mode enabled with adding user ports into virtual bridge device. IP address could be assigned to the bridge.

4.1.2 Bluetooth

This section contains information regarding the Bluetooth stack of the Zephyr OS. You can use this information to understand the principles behind the operation of the layers and how they were implemented.

Zephyr includes a complete Bluetooth Low Energy stack from application to radio hardware, as well as portions of a Classical Bluetooth (BR/EDR) Host layer.

Supported features

Since its inception, Zephyr has had a strong focus on Bluetooth and, in particular, on Bluetooth Low Energy (LE). Through the contributions of several companies and individuals involved in existing open source implementations of the Bluetooth specification (Linux's BlueZ) as well as the design and development of Bluetooth LE radio hardware, the protocol stack in Zephyr has grown to be mature and feature-rich, as can be seen in the section below.

- Bluetooth v5.3 compliant
  - Highly configurable
    - Features, buffer sizes/counts, stack sizes, etc.

  - Portable to all architectures supported by Zephyr (including big and little endian, alignment flavors and more)
  - Support for *all combinations* of Host and Controller builds:
    * Controller-only (HCI) over UART, SPI, USB and IPC physical transports
    * Host-only over UART, SPI, and IPC (shared memory)
    * Combined (Host + Controller)
- *Bluetooth-SIG qualifiable*
  - Conformance tests run regularly on all layers (Controller and Host, except BT Classic) on Nordic Semiconductor hardware.
- *Bluetooth Low Energy Controller* (LE Link Layer)
  - Unlimited role and connection count, all roles supported
  - All v5.3 specification features supported (except a few minor items)
  - Concurrent multi-protocol support ready
  - Intelligent scheduling of roles to minimize overlap
  - Portable design to any open Bluetooth LE radio, currently supports Nordic Semiconductor nRF52x and nRF53x SoC Series, as well as proprietary radios
  - Supports little and big endian architectures, and abstracts the hard real-time specifics so that they can be encapsulated in a hardware-specific module
  - Support for Controller (HCI) builds over different physical transports
  - Isochronous channels
- *Bluetooth Host*
  - Generic Access Profile (GAP) with all possible LE roles
    * Peripheral & Central
    * Observer & Broadcaster
    * Multiple PHY support (2Mbit/s, Coded)
    * Extended Advertising
    * Periodic Advertising (including Sync Transfer)
  - GATT (Generic Attribute Profile)
    * Server (to be a sensor)
    * Client (to connect to sensors)
    * Enhanced ATT (EATT)
    * GATT Database Hash
    * GATT Multiple Notifications
  - Pairing support, including the Secure Connections feature from Bluetooth 4.2
  - Non-volatile storage support for permanent storage of Bluetooth-specific settings and data
  - Bluetooth Mesh support
    * Relay, Friend Node, Low-Power Node (LPN) and GATT Proxy features
    * Both Provisioning roles and bearers supported (PB-ADV & PB-GATT)
    * Foundation Models included
    * Highly configurable, fits as small as 16k RAM devices

- Basic Bluetooth BR/EDR (Classic) support
  - Generic Access Profile (GAP)
  - Logical Link Control and Adaptation Protocol (L2CAP)
  - Serial Port emulation (RFCOMM protocol)
  - Service Discovery Protocol (SDP)
- Clean HCI driver abstraction
  - 3-Wire (H:5) & 5-Wire (H:4) UART
  - SPI
  - Local controller support as a virtual HCI driver
- Verified with multiple popular controllers
- Isochronous channels
- *LE Audio*

Qualification

Details on the Bluetooth SIG qualification process can be found at: https://www.bluetooth.com/develop-with-bluetooth/qualify

Qualification setup The Zephyr Bluetooth host can be qualified using Bluetooth's PTS (Profile Tuning Suite) software. It is originally a manual process, but is automated by using the AutoPTS automation software.

The setup is described in more details in the pages linked below.

AutoPTS on Windows 10 with nRF52 board This tutorial shows how to setup AutoPTS client and server to run both on Windows 10. We use WSL1 with Ubuntu only to build a Zephyr project to an elf file, because Zephyr SDK is not available on Windows yet. Tutorial covers only nrf52840dk.

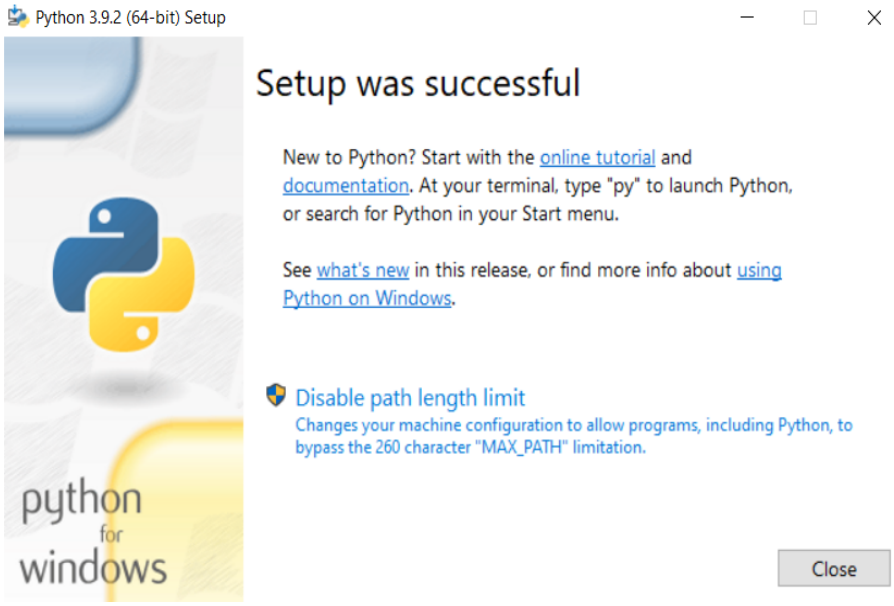
- *Update Windows and drivers*
- *Install Python 3*
- *Install Git*
- *Install PTS 8*
- *Setup Zephyr project for Windows*
- *Install nrftools*
- *Connect devices*
- *Flash board*
- *Setup auto-pts project*
- *Install socat.exe*
- *Running AutoPTS*
- *Troubleshooting*

Update Windows and drivers Update Windows in:

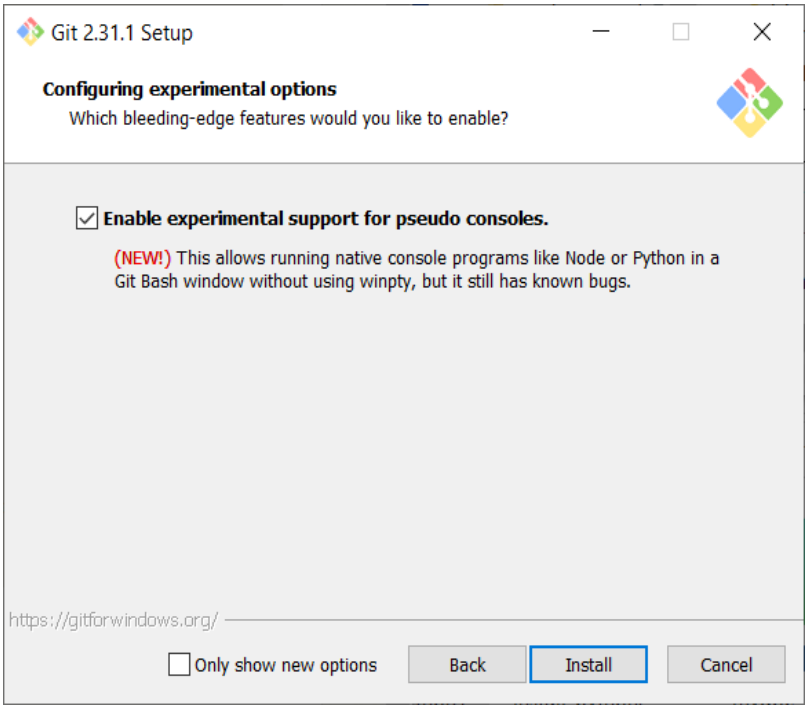
Start -> Settings -> Update & Security -> Windows Update

Update drivers, following the instructions from your hardware vendor.

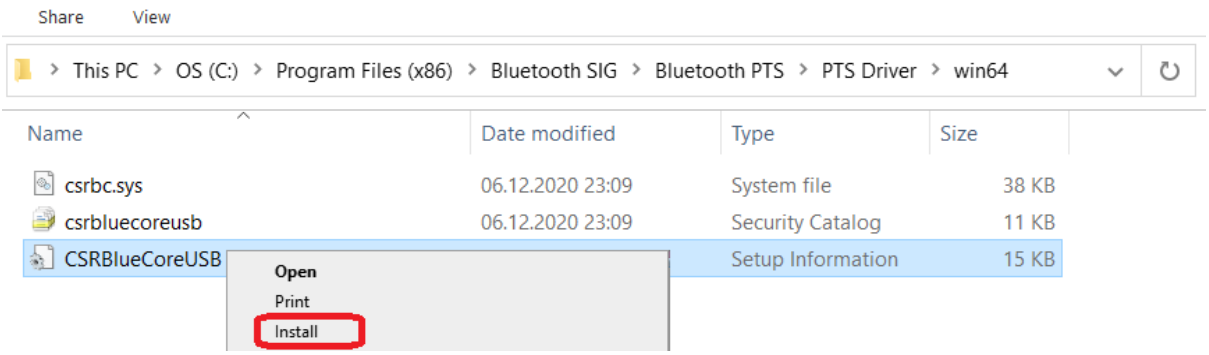
Install Python 3 Download and install Python 3. Setup was tested with versions >=3.8. Let the installer add the Python installation directory to the PATH and disable the path length limitation.



Install Git Download and install Git. During installation enable option: Enable experimental support for pseudo consoles. We will use Git Bash as Windows terminal.



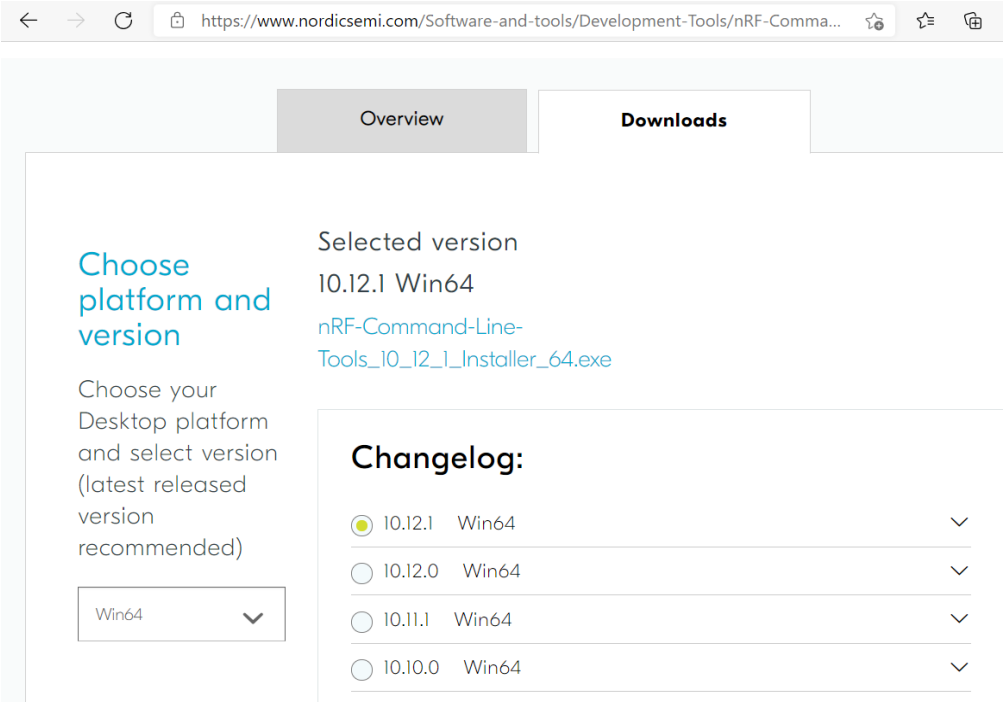
Install PTS 8 Install latest PTS from https://www.bluetooth.org. Remember to install drivers from installation directory "C:/Program Files (x86)/Bluetooth SIG/Bluetooth PTS/PTS Driver/win64/CSRBlueCoreUSB.inf"

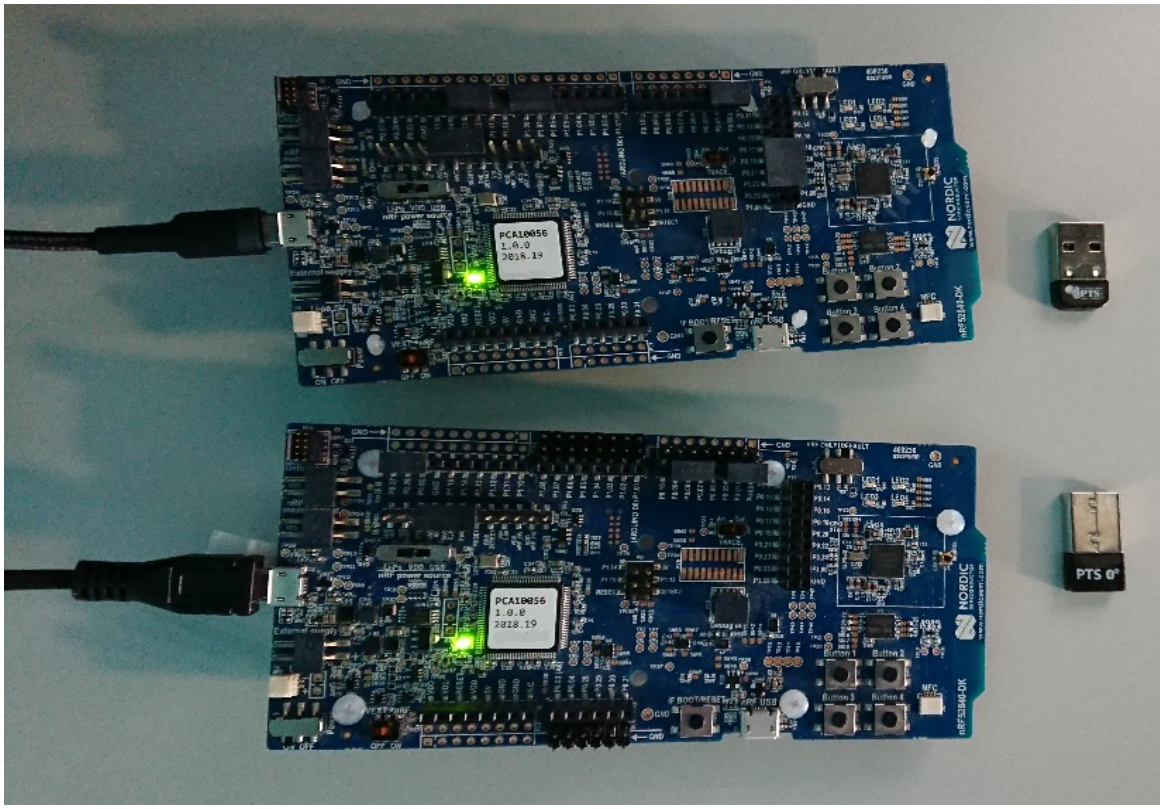


> **Note**
>
> Starting with PTS 8.0.1 the Bluetooth Protocol Viewer is no longer included. So to capture Bluetooth events, you have to download it separately.

Setup Zephyr project for Windows Perform Windows setup from *Getting Started Guide*.

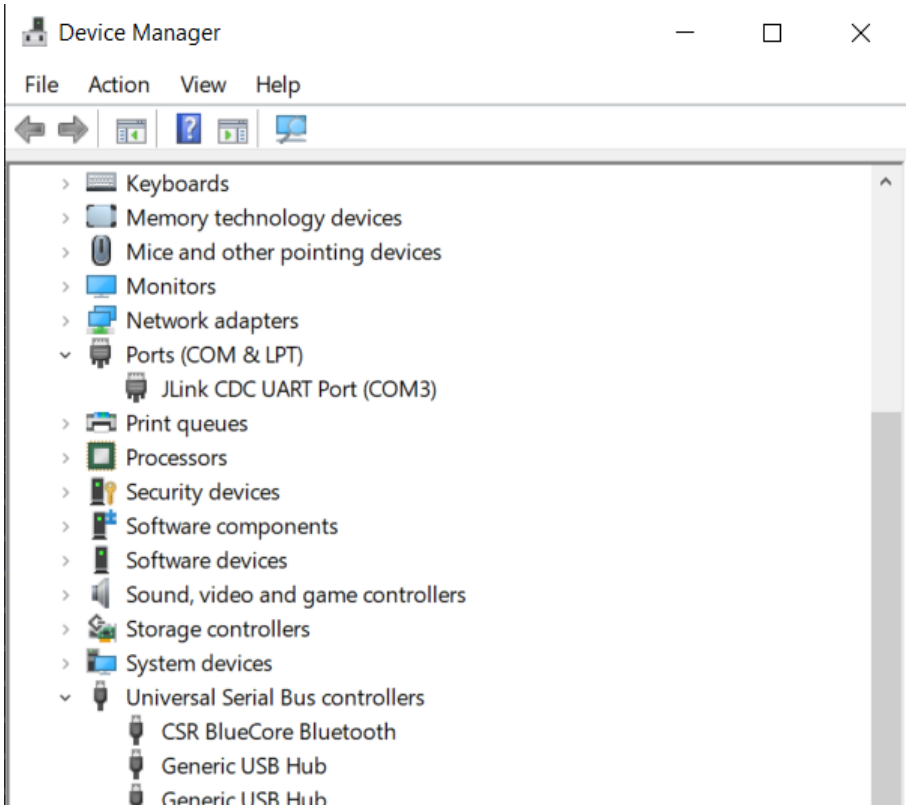
Install nrftools On Windows download latest nrftools (version >= 10.12.1) from site https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Command-Line-Tools/Download and run default install.





Connect devices

Flash board In Device Manager find COM port of your nrf board. In my case it is COM3.



In Git Bash, go to zephyrproject

```
cd ~/zephyrproject
```

Build the auto-pts tester app

```
west build -p auto -b nrf52840dk/nrf52840 zephyr/tests/bluetooth/tester/
```

You can display flashing options with:

```
west flash --help
```

and flash board with built earlier elf file:

```
west flash --no-rebuild --board-dir /dev/ttyS2 --elf-file ~/
↪zephyrproject/build/zephyr/zephyr.elf
```

Note that west does not accept COMs, so use /dev/ttyS2 as the COM3 equivalent, /dev/ttyS2 as the COM3 equivalent, etc.(/dev/ttyS + decremented COM number).

Setup auto-pts project In Git Bash, clone project repo:

```
git clone https://github.com/auto-pts/auto-pts.git
```

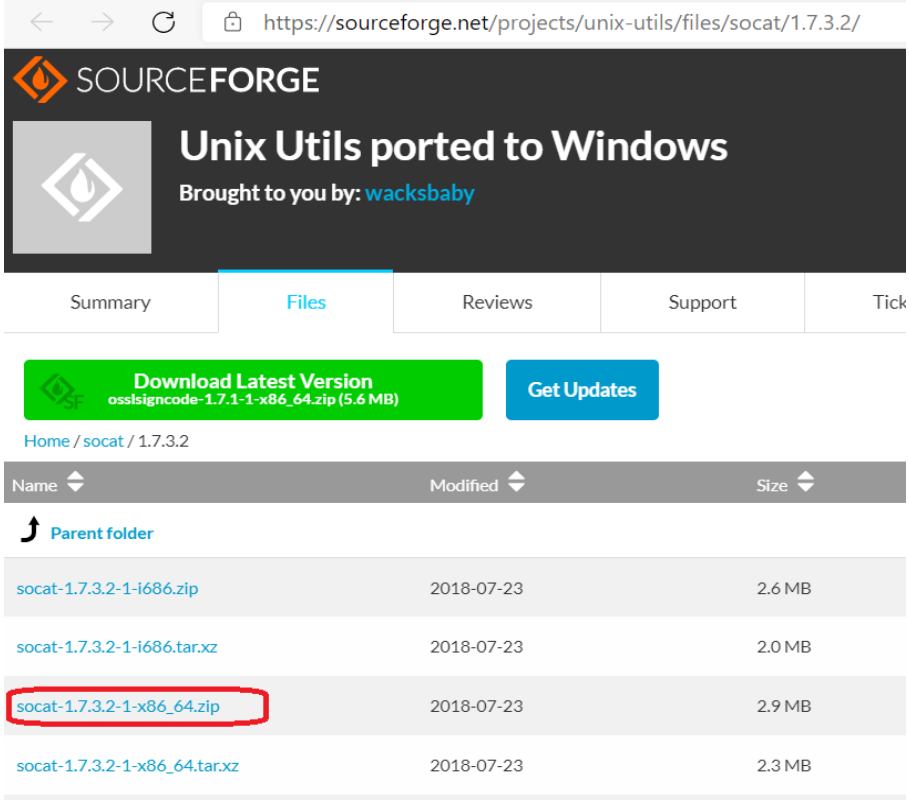
Go into the project folder:

```
cd auto-pts
```

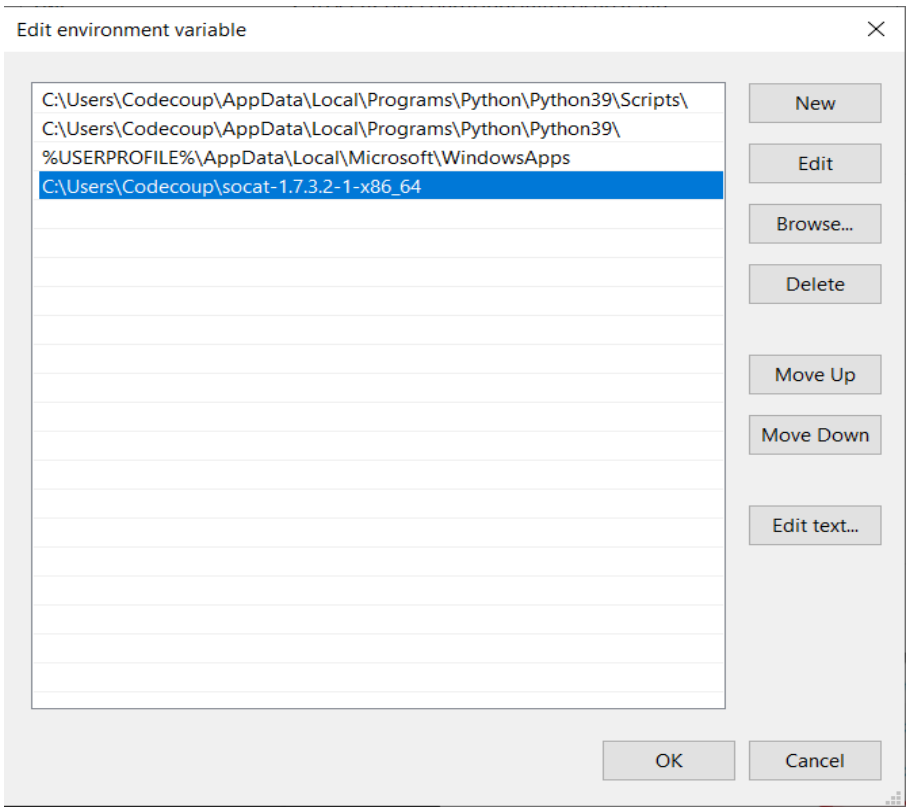
Install required python modules:

```
pip3 install --user wheel
pip3 install --user -r autoptsserver_requirements.txt
pip3 install --user -r autoptsclient_requirements.txt
```

Install socat.exe Download and extract socat.exe from https://sourceforge.net/projects/unix-utils/files/socat/1.7.3.2/ into folder ~/socat-1.7.3.2-1-x86_64/.

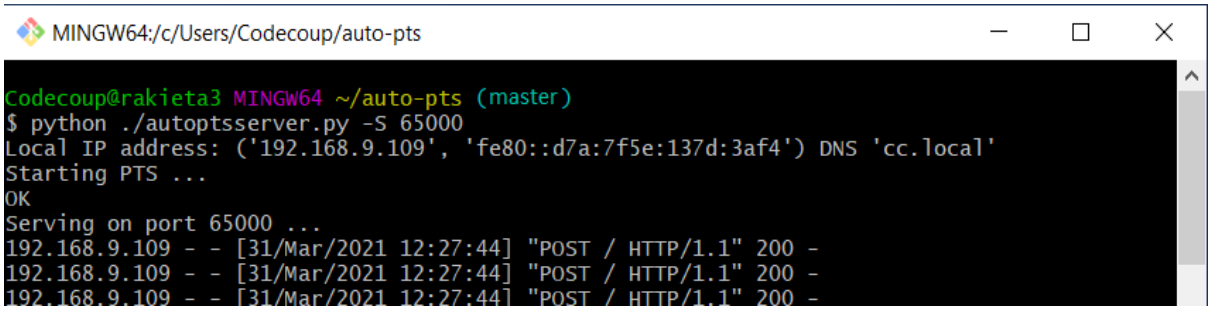


Add path to directory of socat.exe to PATH:



Running AutoPTS Server and client by default will run on localhost address. Run server:

```
python ./autoptsserver.py -S 65000
```



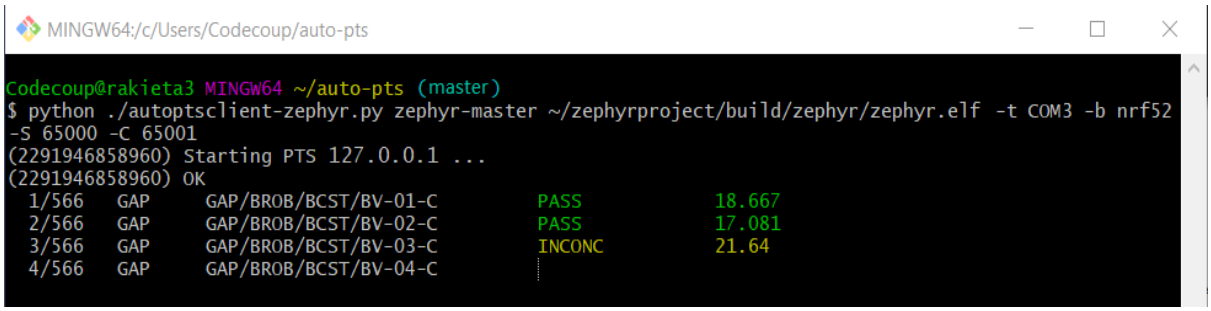
Note

If the error "ImportError: No module named pywintypes" appeared after the fresh setup, uninstall and install the pywin32 module:

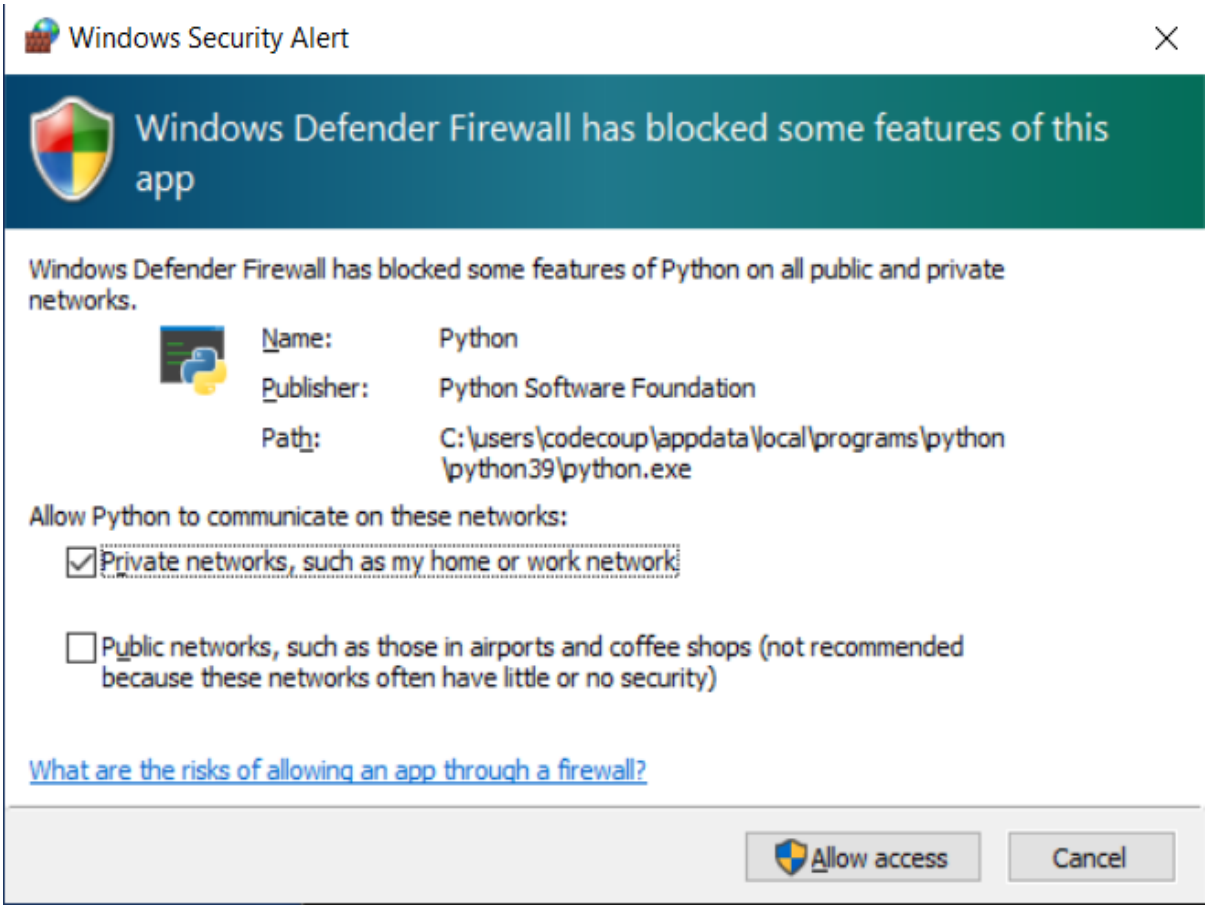
```
pip install --upgrade --force-reinstall pywin32
```

Run client:

```
python ./autoptsclient-zephyr.py zephyr-master ~/
↪zephyrproject/build/zephyr/zephyr.elf -t COM3 -b nrf52 -S 65000 -C 65001
```



At the first run, when Windows asks, enable connection through firewall:



Troubleshooting

- "When running actual hardware test mode, I have only BTP TIMEOUTs."

This is a problem with connection between auto-pts client and board. There are many possible causes. Try:

- Clean your auto-pts and zephyr repos with

> **Warning**
>
> This command will force the irreversible removal of all uncommitted files in the repo.

```
git clean -fdx
```

then build and flash tester elf again.

- If you have set up Windows on virtual machine, check if guest extensions are installed properly or change USB compatibility mode in VM settings to USB 2.0.
- Check, if firewall in not blocking python.exe or socat.exe.
- Check if board sends ready event after restart (hex 00 00 80 ff 00 00). Open serial connection to board with e.g. PuTTy with proper COM and baud rate. After board reset you should see some strings in console.
- Check if socat.exe creates tunnel to board. Run in console

```
socat.exe -x -v tcp-listen:65123 /dev/ttyS2,raw,b115200
```

where /dev/ttyS2 is the COM3 equivalent. Open PuTTY, set connection type to Raw, IP to 127.0.0.1, port to 65123. After board reset you should see some strings in console.

AutoPTS on Linux This tutorial shows how to setup AutoPTS client on Linux with AutoPTS server running on Windows 10 virtual machine. Tested with Ubuntu 20.4 and Linux Mint 20.4.

You must have a Zephyr development environment set up. See *Getting Started Guide* for details.

Supported methods to test zephyr bluetooth host:

- Testing Zephyr Host Stack on QEMU
- Testing Zephyr Host Stack on native_sim
- Testing Zephyr combined (controller + host) build on Real hardware (such as nRF52)

For running with QEMU or native_sim, see *Running on QEMU or native_sim*.

- *Setup Linux*
- *Setup Windows 10/11 virtual machine*
  - *Update Windows*
  - *Setup NAT*
  - *Setup static IP*
  - *Install Python 3*
  - *Install Git*
  - *Install PTS*
  - *Connect PTS dongle*
- *Connect devices (only required in the actual hardware test mode)*
- *Setup auto-pts project*
  - *AutoPTS client on Linux*
  - *Autopts server on Windows virtual machine*
- *Running AutoPTS*
  - *Testing Zephyr Host Stack on hardware*
  - *Testing Zephyr Host Stack on QEMU*
  - *Testing Zephyr Host Stack on native_sim*
- *Troubleshooting*
  - *After running one test, I need to restart my Windows virtual machine to run another, because of fail verdict from APICOM in PTS logs*
  - *I cannot start autoptsserver-zephyr.py. I always get a Python error*
  - *The PTS automation window keeps opening and closing*

Setup Linux Please follow *Getting Started Guide* on how to setup Linux for building and flashing applications.

Setup Windows 10/11 virtual machine Choose and install your hypervisor like VMWare Workstation(preferred) or VirtualBox. On VirtualBox could be some issues, if your host has fewer than 6 CPU.

Create Windows virtual machine instance. Make sure it has at least 2 cores and installed guest extensions.

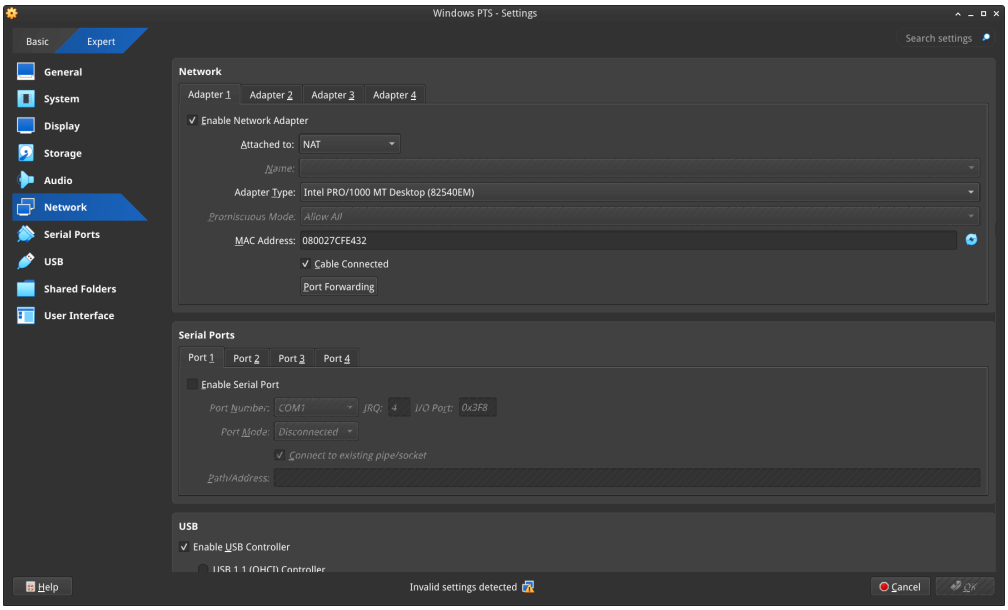
Setup tested with VirtualBox 7.2.4 and VMWare Workstation 16.1.1 Pro.

Update Windows Update Windows in:

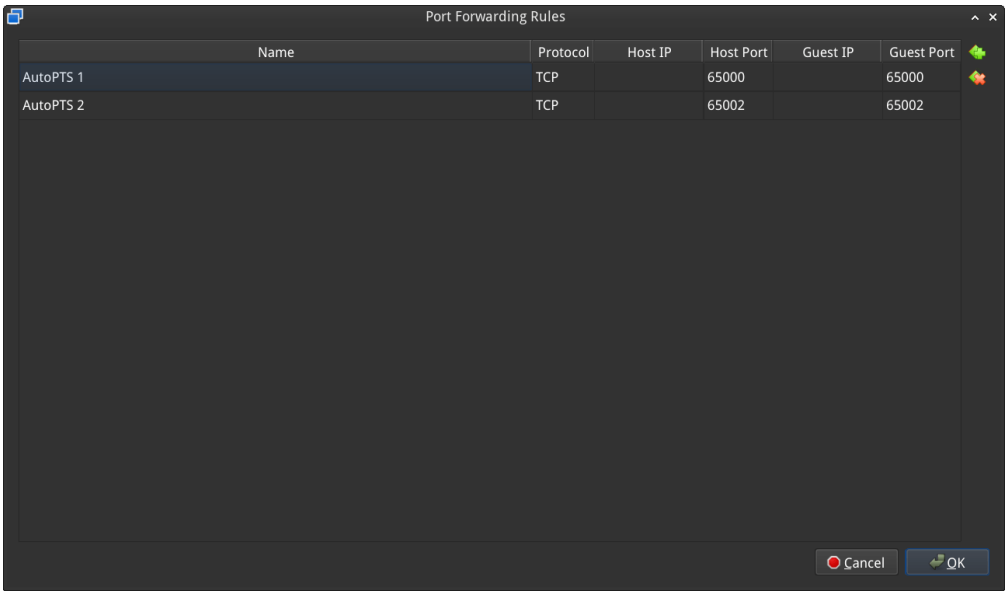
Start -> Settings -> Update & Security -> Windows Update

Setup NAT It is possible to use NAT and portforwarding to setup communication between a Linux host and a Windows guest. This is easiest setup for VirtualBox, and does not require any static IPs to be configured, and will not get blocked by the Windows Firewall.

VirtualBox Open virtual machine network settings. On adapter 1 you will have created by default NAT. Open the Port Forwarding menu an add the ports you want.

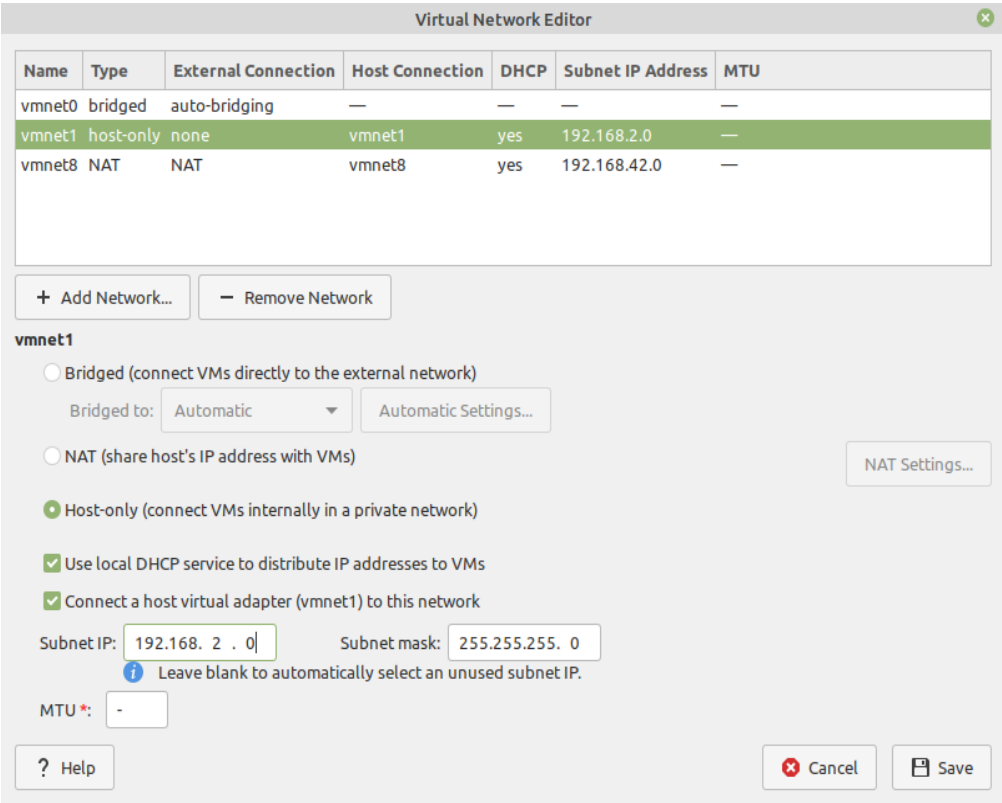


For example setting up the following will allow you to use `localhost:65000` and `localhost:65002` (or `127.0.0.0:65000` and `127.0.0.0:65002`) to connect to an AutoPTS Server in Windows running on ports 65000 and 65002.

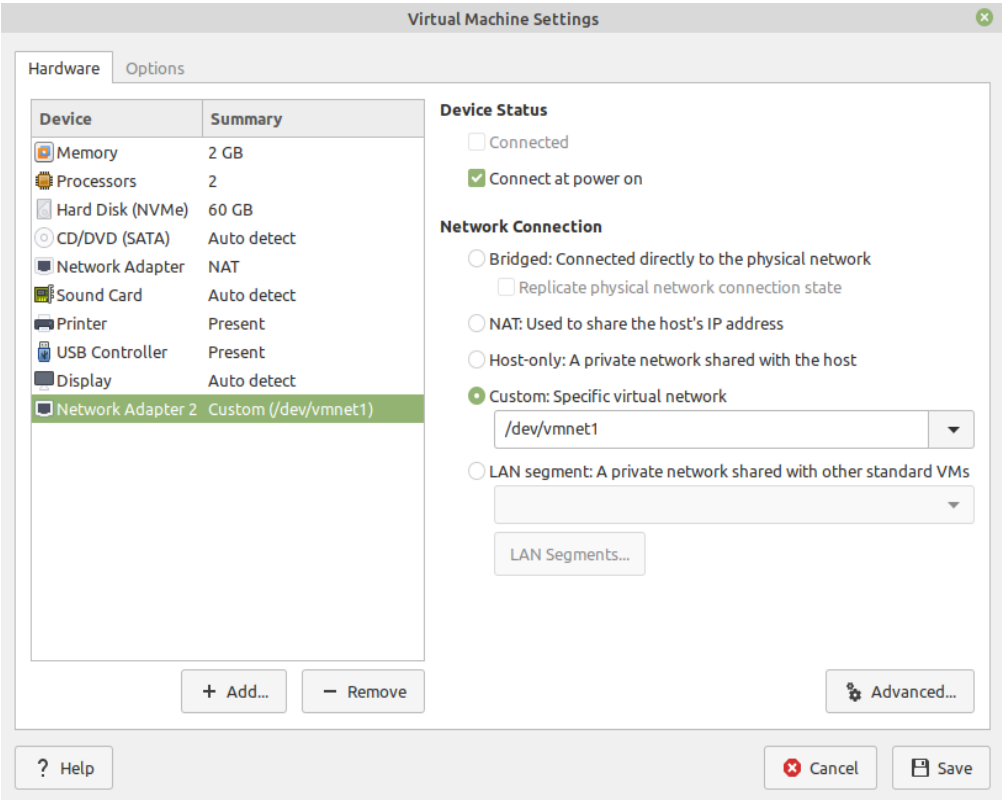


Setup static IP If you cannot or do not want to use NAT it is possible to configure a static IP.

VMWare Works On Linux, open Virtual Network Editor app and create network:



Open virtual machine network settings. Add custom adapter:



If you type 'ifconfig' in terminal, you should be able to find your host IP:

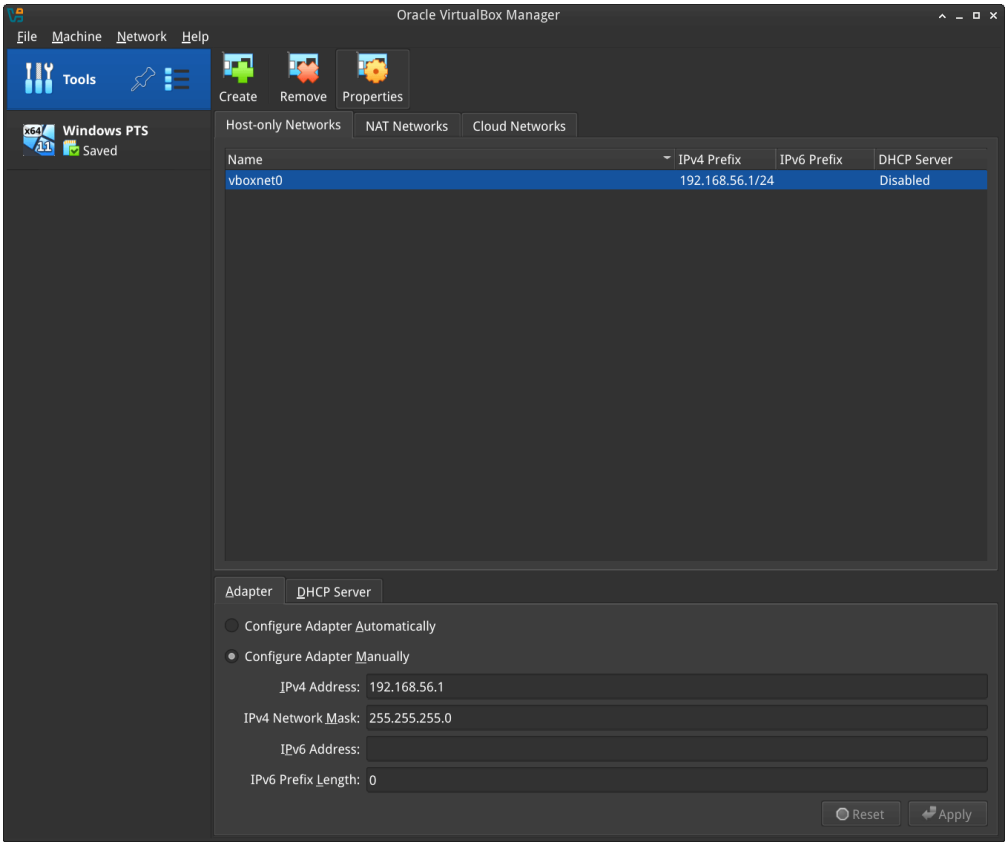
```
vmnet1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.1  netmask 255.255.255.0  broadcast 192.168.2.255
        inet6 fe80::250:56ff:fec0:1  prefixlen 64  scopeid 0x20<link>
        ether 00:50:56:c0:00:01  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 42  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

VirtualBox VirtualBox on Linux, macOS and Solaris Oracle VM VirtualBox will only allow IP addresses in `192.168.56.0/21` range to be assigned to host-only adapters, so if using a static address with VirtualBox this is the only address range you can use.

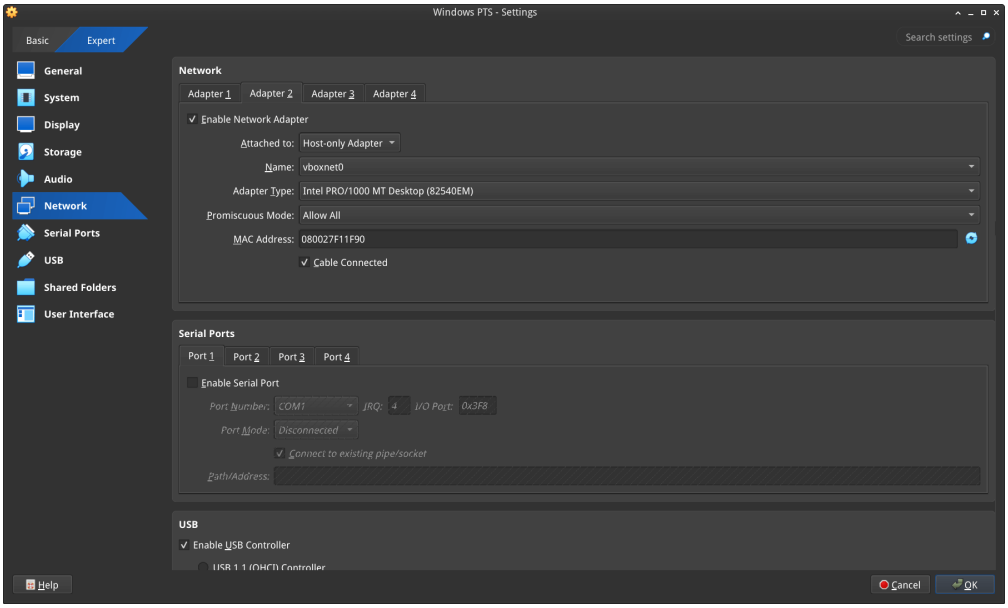
Go to:

File -> Tools -> Network Manager

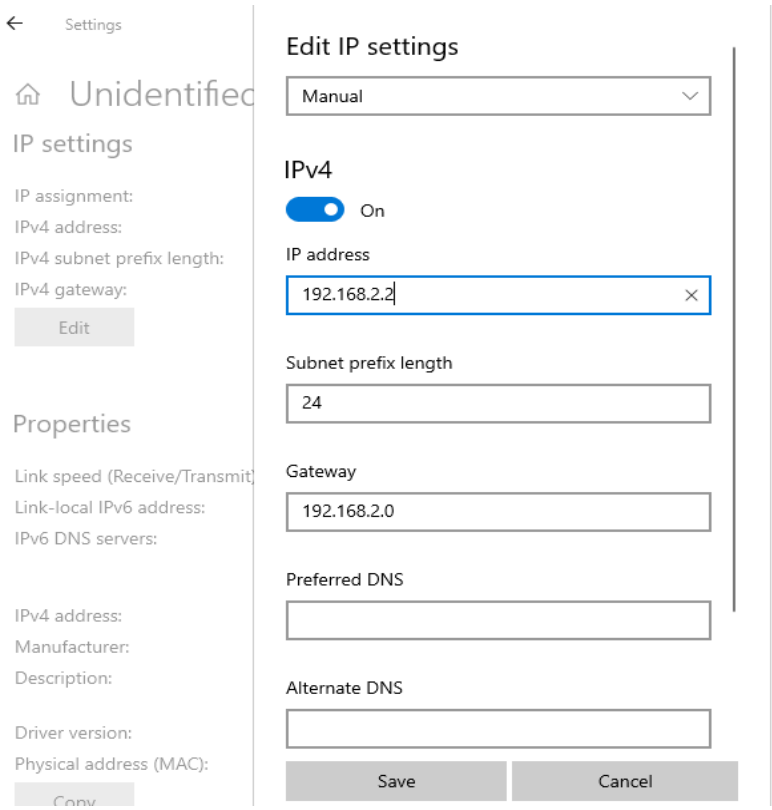
and create network:



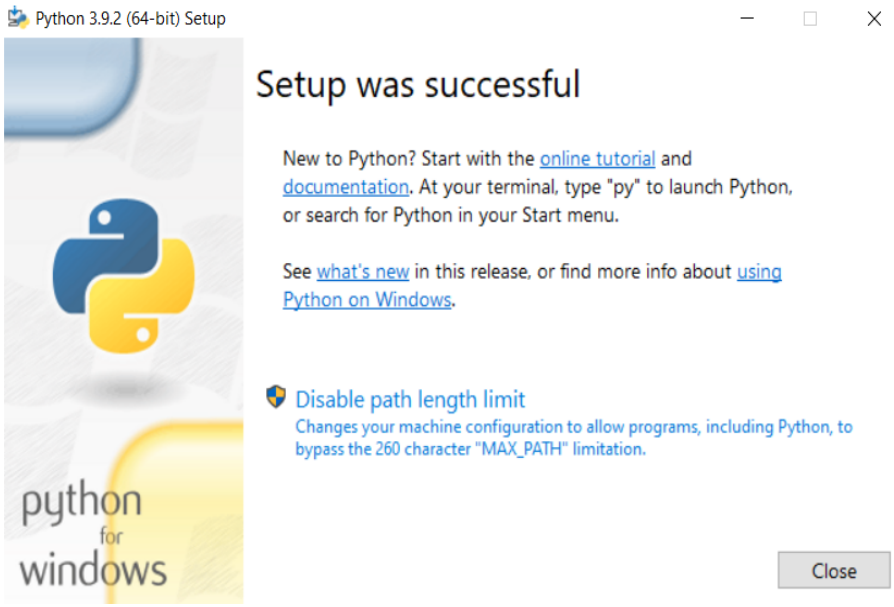
Open virtual machine network settings. On adapter 1 you will have created by default NAT. Add adapter 2:



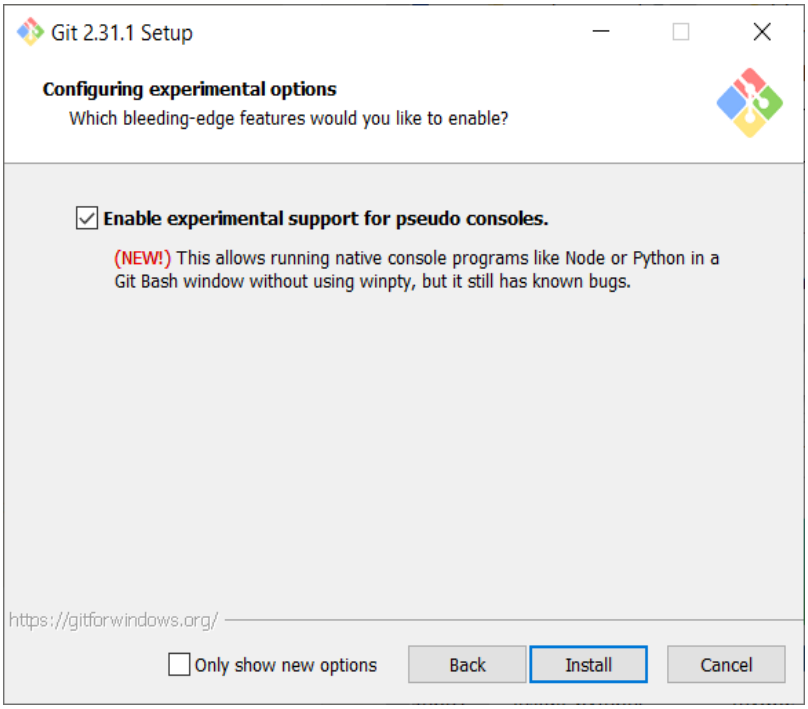
Windows Setup static IP on Windows virtual machine. Go to Settings -> Network & Internet -> Ethernet -> Unidentified network -> Edit and set:



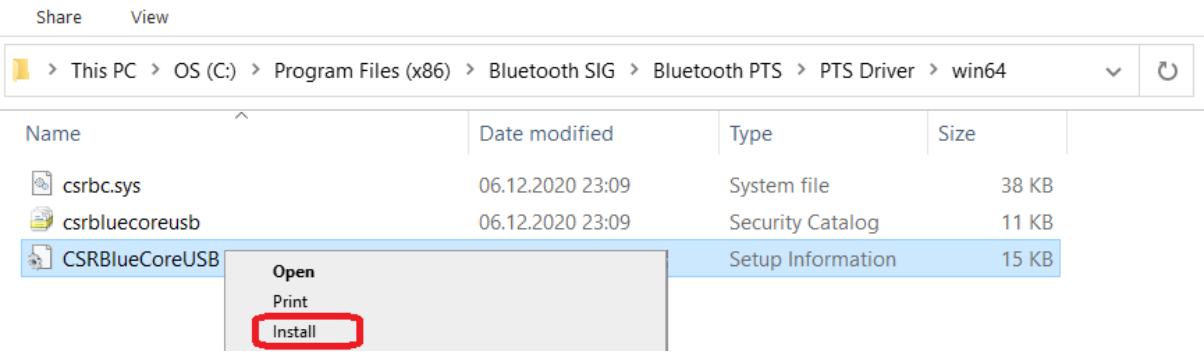
Install Python 3 Download and install latest Python 3 on Windows. Let the installer add the Python installation directory to the PATH and disable the path length limitation.



Install Git Download and install Git. During installation enable option: Enable experimental support for pseudo consoles. We will use Git Bash as Windows terminal.



Install PTS On Windows virtual machine, install the latest PTS from https://pts.bluetooth.com/download. Remember to install drivers from installation directory "C:/Program Files (x86)/Bluetooth SIG/Bluetooth PTS/PTS Driver/win64/CSRBlueCoreUSB.inf"



> **Note**
>
> Starting with PTS 8.0.1 the Bluetooth Protocol Viewer is no longer included. So to capture Bluetooth events, you have to download it separately.

Connect PTS dongle With VirtualBox there should be no problem. Just find dongle in Devices -> USB and connect.

With VMWare you might need to use some trick, if you cannot find dongle in VM -> Removable Devices. Type in Linux terminal:

```
usb-devices
```

and find in output your PTS Bluetooth USB dongle

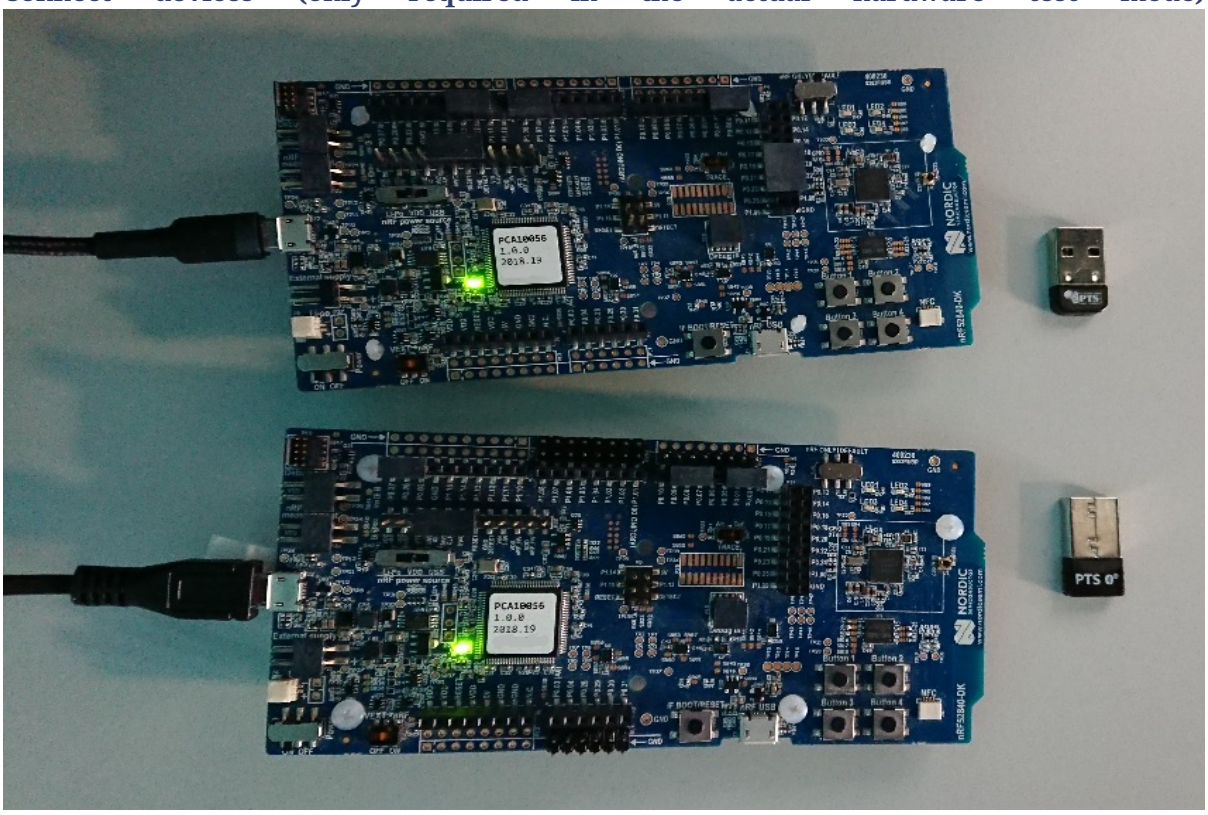
```
T:  Bus=01 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  6 Spd=12  MxCh= 0
D:  Ver= 2.00 Cls=e0(wlcon) Sub=01 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=0a12 ProdID=0001 Rev=25.20
S:  Product=CSR rck PTS Dongle
C:  #Ifs= 3 Cfg#= 1 Atr=c0 MxPwr=0mA
```

Note Vendor and ProdID number. Close VMWare Workstation and open .vmx of your virtual machine (path similar to /home/codecoup/vmware/Windows 10/Windows 10.vmx) in text editor. Write anywhere in the file following line:

```
usb.autoConnect.device0 = "0x0a12:0x0001"
```

just replace 0x0a12 with Vendor number and 0x0001 with ProdID number you found earlier.

Connect devices (only required in the actual hardware test mode)



Setup auto-pts project

AutoPTS client on Linux Clone auto-pts project:

```
git clone https://github.com/auto-pts/auto-pts.git
```

Install socat, that is used to transfer BTP data stream from UART's tty file:

```
sudo apt-get install python-setuptools socat
```

Install required python modules:

```
cd auto-pts
pip3 install --user -r autoptsclient_requirements.txt
```

Autopts server on Windows virtual machine In Git Bash, clone auto-pts project repo:

```
git clone https://github.com/auto-pts/auto-pts.git
```

Install required python modules:

```
cd auto-pts
pip3 install --user wheel
pip3 install --user -r autoptsserver_requirements.txt
```

Restart virtual machine.

Running AutoPTS Please follow the information from https://github.com/zephyrproject-rtos/zephyr/tree/main/tests/bluetooth/tester on how to build, flash and run the Bluetooth Tester application.

Server and client by default will run on localhost address. Run the server in the Windows virtual machine:

```
python ./autoptsserver.py
```

```
codecoup@DESKTOP-5SFTBGB MINGW64 ~/auto-pts (master)
$ python ./autoptsserver.py
Local IP address: ('10.0.2.15', 'fe80::ccb2:6ca1:6368:342a') DNS 'home'
Local IP address: ('192.168.2.2', 'fe80::fd12:15d5:2ddb:9147') DNS None
Starting PTS ...
OK
Serving on port 65000 ...
```

See also https://github.com/auto-pts/auto-pts for additional information on how to run auto-pts.

Testing Zephyr Host Stack on hardware

```
python ./autoptsclient-zephyr.py zephyr-master -t /dev/ttyACM0 -b BOARD -i␣
↪SERVER_IP -l LOCAL_IP
```

Where `/dev/ttyACM0` is the tty for the board, `BOARD` is the board to use (e.g. `nrf53_audio`), `SERVER_IP` is the IP of the AutoPTS server, `LOCAL_IP` is the local IP of the Linux machine.

Testing Zephyr Host Stack on QEMU A Bluetooth controller needs to be mounted. For running with HCI UART, please visit bluetooth_hci_uart.

```
python ./autoptsclient-zephyr.py zephyr-master BUILD_DIR/zephyr/zephyr.elf -i␣
↪SERVER_IP -l LOCAL_IP
```

Where `BUILD_DIR` is the build directory, `SERVER_IP` is the IP of the AutoPTS server, `LOCAL_IP` is the local IP of the Linux machine.

Testing Zephyr Host Stack on native_sim When tester application has been built for native_sim it produces a `zephyr.exe` file, that can be run as a native Linux application. Depending on your system, you may need to perform the following steps to successfully run `zephyr.exe`.

Setting capabilities Since the application will need access to connect to a socket for HCI, you may need to perform the following

```
setcap cap_net_raw,cap_net_admin,cap_sys_admin+ep zephyr.exe
```

This is not required if you run `zephyr.exe` or `./autoptsclient-zephyr.py` with e.g. `sudo`.

Downing the HCI controller You may also need to "down" or "power off" the HCI controller before running `zephyr.exe`. This can be done either with `hciconfig` as

```
hciconfig hciX down
```

Where `hciX` is a value like `hci0`. You may run `hciconfig` to get a list of your HCI devices.

Since `hciconfig` is deprecated on some systems, you may need to use

```
btmgmt -i hciX power off
```

Similar to `hciconfig`, `btmgmt info` may be used to list current controllers and their states.

Both `hciconfig` and `btmgmt` may require `sudo` when powering down a controller.

Running the client The application can be run as

```
python ./autoptsclient-zephyr.py zephyr-master --hci HCI␣
↪BUILD_DIR/zephyr/zephyr.exe -i SERVER_IP -l LOCAL_IP
```

Where `HCI` is the HCI index, e.g. `0` or `1`, `BUILD_DIR` is the build directory, `SERVER_IP` is the IP of the AutoPTS server, `LOCAL_IP` is the local IP of the Linux machine.

Troubleshooting

After running one test, I need to restart my Windows virtual machine to run another, because of fail verdict from APICOM in PTS logs It means your virtual machine has not enough processor cores or memory. Try to add more in settings. Note that a host with 4 CPUs could be not enough with VirtualBox as hypervisor. In this case, choose rather VMWare Workstation.

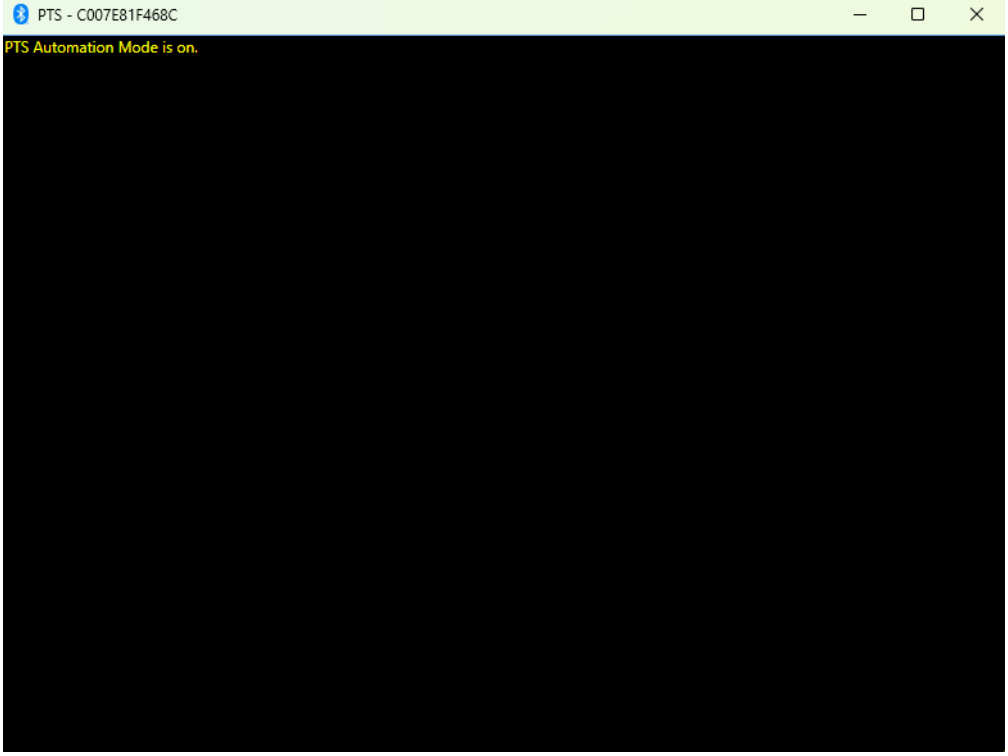
I cannot start autoptsserver-zephyr.py. I always get a Python error

```
codecoup@DESKTOP-5SFTBGB MINGW64 ~/auto-pts (master)
$ python ./autoptsserver.py
Local IP address: ('10.0.2.15', 'fe80::ccb2:6ca1:6368:342a') DNS 'home'
Local IP address: ('192.168.2.2', 'fe80::fd12:15d5:2ddb:9147') DNS None
Starting PTS ...
Traceback (most recent call last):
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 127, in <module>
    main()
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 115, in main
    pts = PyPTSWithXmlRpcCallback()
  File "C:\Users\codecoup\auto-pts\autoptsserver.py", line 52, in __init__
    ptscontrol.PyPTS.__init__(self)
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 267, in __init__
    self.restart_pts()
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 391, in restart_pts
    self.start_pts()
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 440, in start_pts
    log("PTS Bluetooth Address: %s", self.get_bluetooth_address())
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 762, in get_bluetooth_address
    raise e
  File "C:\Users\codecoup\auto-pts\ptscontrol.py", line 759, in get_bluetooth_address
    address = self._pts.GetPTSBluetoothAddress()
  File "<COMObject ProfileTuningSuite_6.PTSControlServer>", line 3, in GetPTSBluetoothAddress
pywintypes.com_error: (-2147352567, 'Exception occurred.', (0, 'PTS', '(HRESULT:0x849C0043) Error HResult!', None
, 0, -2070151101), None)
```

One or more of the following steps should help:

- Close all PTS Windows.
- Replug PTS bluetooth dongle.
- Delete temporary workspace. You will find it in auto-pts-code/workspaces/zephyr/zephyr-master/ as temp_zephyr-master. Be careful, do not remove the original one zephyr-master.pqw6.
- Restart Windows virtual machine.

The PTS automation window keeps opening and closing This indicates that it fails to capture a PTS dongle. If the AutoPTS server is able to find and use a PTS dongle, then the title of the window will show the Bluetooth address of the dongle. If this does not happen then ensure that the dongle is plugged in, updated and recognized by PTS.



If it still fails to run tests after this, please ensure that the Bluetooth Protocol Viewer is installed.

ICS Features The Zephyr ICS file for the Host features can be downloaded here: `ICS_Zephyr_Bluetooth_Host.pts`.

Use the Bluetooth Qualification website to view and edit the ICS.

Qualified releases The Zephyr Project provides a pre-qualified Bluetooth host stack to make it easy for users to build qualified Bluetooth products. It can be included in a product qualification design, providing feature coverage and reducing qualification effort. The scope of qualification may vary by release, and the details can be checked below.

Release	Design Number	Details
v4.4	Q385945	Bluetooth qualification listing 332380

Stack Architecture

Overview This page describes the software architecture of Zephyr's Bluetooth protocol stack.

> **Note**
>
> Zephyr supports mainly Bluetooth Low Energy (LE), the low-power version of the Bluetooth specification. Zephyr also has limited support for portions of the BR/EDR Host.

Bluetooth LE Layers There are 3 main layers that together constitute a full Bluetooth Low Energy protocol stack:

- **Host**: This layer sits right below the application, and is comprised of multiple (non real-time) network and transport protocols enabling applications to communicate with peer devices in a standard and interoperable way.
- **Controller**: The Controller implements the Link Layer (LE LL), the low-level, real-time protocol which provides, in conjunction with the Radio Hardware, standard-interoperable over-the-air communication. The LL schedules packet reception and transmission, guarantees the delivery of data, and handles all the LL control procedures.
- **Radio Hardware**: Hardware implements the required analog and digital baseband functional blocks that permit the Link Layer firmware to send and receive in the 2.4GHz band of the spectrum.

Host Controller Interface The Bluetooth Specification describes the format in which a Host must communicate with a Controller. This is called the Host Controller Interface (HCI) protocol. HCI can be implemented over a range of different physical transports like UART, SPI, or USB. This protocol defines the commands that a Host can send to a Controller and the events that it can expect in return, and also the format for user and protocol data that needs to go over the air. The HCI ensures that different Host and Controller implementations can communicate in a standard way making it possible to combine Hosts and Controllers from different vendors.

Configurations The three separate layers of the protocol and the standardized interface make it possible to implement the Host and Controller on different platforms. The two following configurations are commonly used:

- **Single-chip configuration**: In this configuration, a single microcontroller implements all three layers and the application itself. This can also be called a system-on-chip (SoC) implementation. In this case the Bluetooth Host and the Bluetooth Controller communicate directly through function calls and queues in RAM. The Bluetooth specification does not specify how HCI is implemented in this single-chip configuration and so how HCI commands, events, and data flows between the two can be implementation-specific. This configuration is well suited for those applications and designs that require a small footprint and the lowest possible power consumption, since everything runs on a single IC.
- **Dual-chip configuration**: This configuration uses two separate ICs, one running the Application and the Host, and a second one with the Controller and the Radio Hardware. This is sometimes also called a connectivity-chip configuration. This configuration allows for a wider variety of combinations of Hosts when using the Zephyr OS as a Controller. Since HCI ensures interoperability among Host and Controller implementations, including of course Zephyr's very own Bluetooth Host and Controller, users of the Zephyr Controller can choose to use whatever Host running on any platform they prefer. For example, the host can be the Linux Bluetooth Host stack (BlueZ) running on any processor capable of supporting Linux. The Host processor may of course also run Zephyr and the Zephyr OS Bluetooth Host. Conversely, combining an IC running the Zephyr Host with an external Controller that does not run Zephyr is also supported.

Build Types The Zephyr software stack as an RTOS is highly configurable, and in particular, the Bluetooth subsystem can be configured in multiple ways during the build process to include only the features and layers that are required to reduce RAM and ROM footprint as well as power

consumption. Here's a short list of the different Bluetooth-enabled builds that can be produced from the Zephyr project codebase:

- **Controller-only build**: When built as a Bluetooth Controller, Zephyr includes the Link Layer and a special application. This application is different depending on the physical transport chosen for HCI:
  - bluetooth_hci_uart
  - bluetooth_hci_usb
  - bluetooth_hci_spi

  This application acts as a bridge between the UART, SPI or USB peripherals and the Controller subsystem, listening for HCI commands, sending application data and responding with events and received data. A build of this type sets the following Kconfig option values:
  - `CONFIG_BT=y`
  - `CONFIG_BT_HCI=y`
  - `CONFIG_BT_HCI_RAW=y`

  The controller itself needs to be enabled as well, typically by making sure the corresponding device tree node is enabled.
- **Host-only build**: A Zephyr OS Host build will contain the Application and the Bluetooth Host, along with an HCI driver (UART or SPI) to interface with an external Controller chip. A build of this type sets the following Kconfig option values:
  - `CONFIG_BT=y`
  - `CONFIG_BT_HCI=y`

  Additionally, if the platform supports also a local controller, it needs to be disabled, typically by disabling the corresponding device tree node. This is done together with enabling the device tree node for some other HCI driver and making sure that the `zephyr,bt-hci` device tree chosen property points at it.

  All of the samples located in `samples/bluetooth` except for the ones used for Controller-only builds can be built as Host-only
- **Combined build**: This includes the Application, the Host and the Controller, and it is used exclusively for single-chip (SoC) configurations. A build of this type sets the following Kconfig option values:
  - `CONFIG_BT=y`
  - `CONFIG_BT_HCI=y`

  The controller itself needs to be enabled as well, typically by making sure the corresponding device tree node is enabled.

  All of the samples located in `samples/bluetooth` except for the ones used for Controller-only builds can be built as Combined

The picture below shows the SoC or single-chip configuration when using a Zephyr combined build (a build that includes both a Bluetooth Host and a Controller in the same firmware image that is programmed onto the chip):

When using connectivity or dual-chip configurations, several Host and Controller combinations are possible, some of which are depicted below:

When using a Zephyr Host (left side of image), two instances of Zephyr OS must be built with different configurations, yielding two separate images that must be programmed into each of the chips respectively. The Host build image contains the application, the Bluetooth Host and the selected HCI driver (UART or SPI), while the Controller build runs either the bluetooth_hci_uart, or the bluetooth_hci_spi app to provide an interface to the Bluetooth Controller.

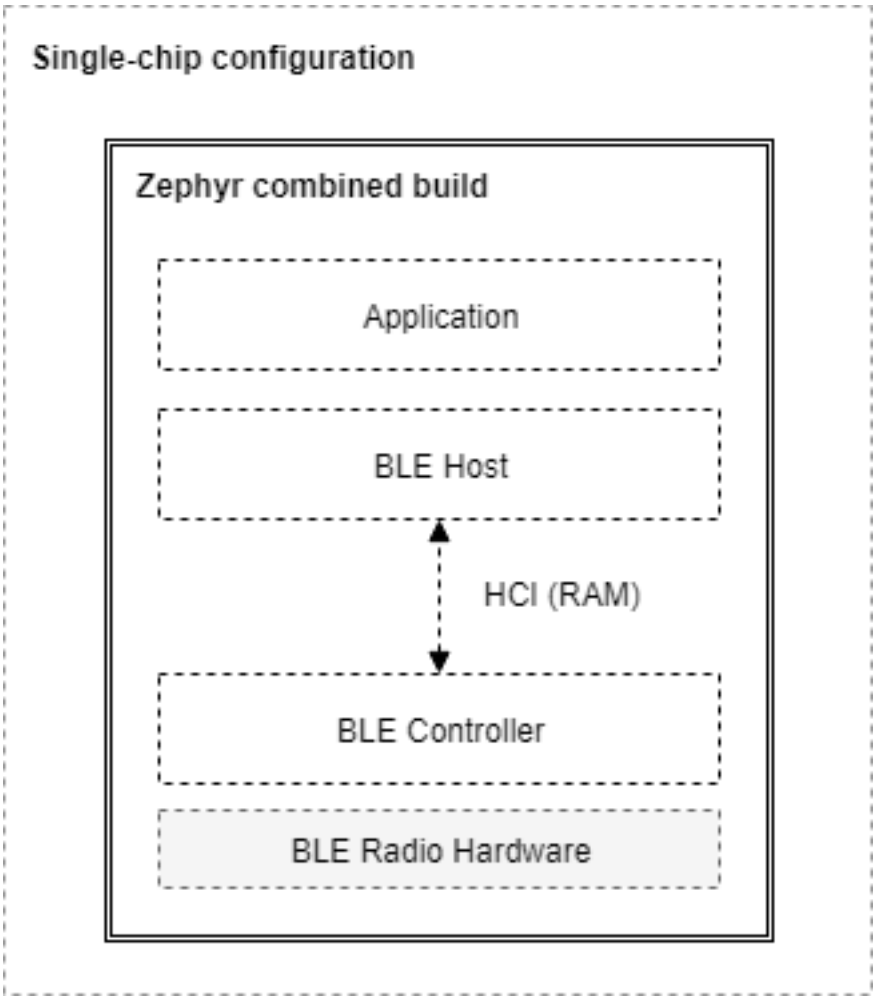


Fig. 12: A Combined build on a Single-Chip configuration

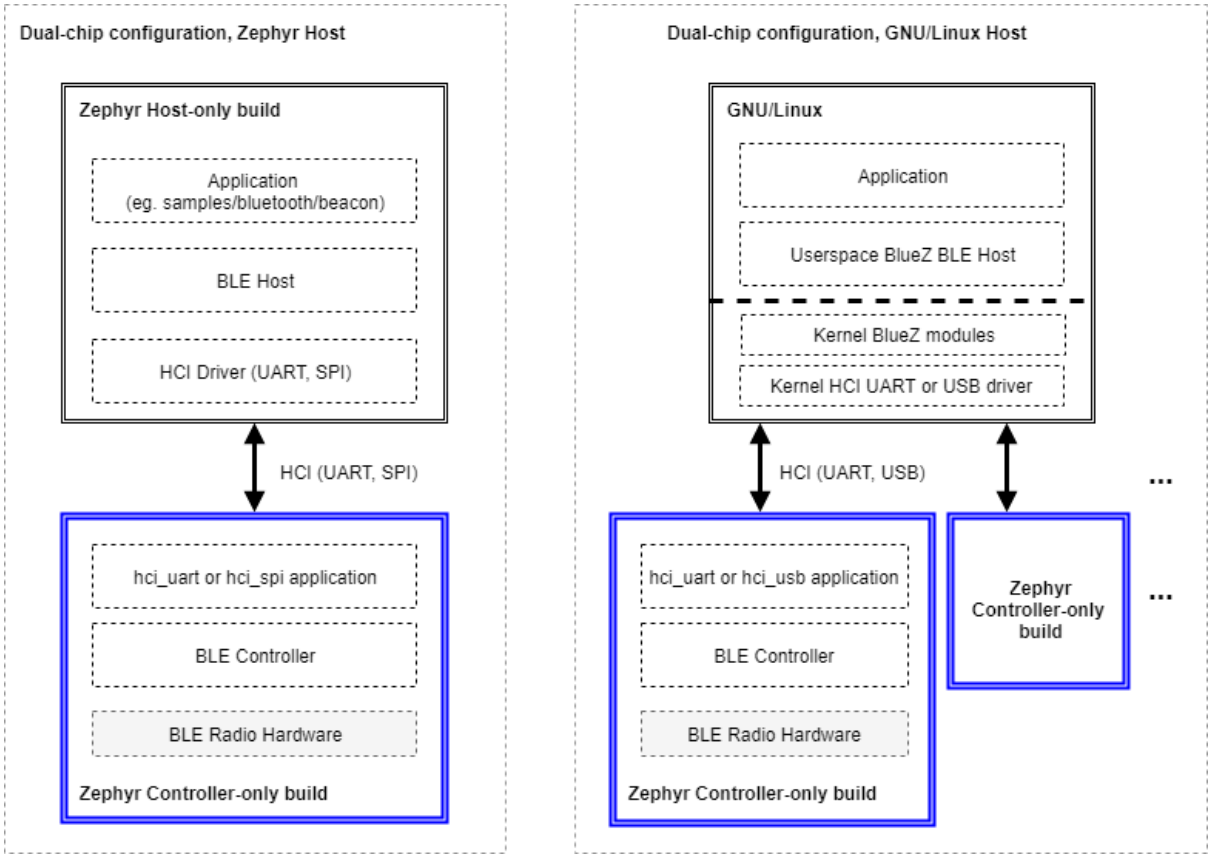


Fig. 13: Host-only and Controller-only builds on dual-chip configurations

This configuration is not limited to using a Zephyr OS Host, as the right side of the image shows. One can indeed take one of the many existing GNU/Linux distributions, most of which include Linux's own Bluetooth Host (BlueZ), to connect it via UART or USB to one or more instances of the Zephyr OS Controller build. BlueZ as a Host supports multiple Controllers simultaneously for applications that require more than one Bluetooth radio operating at the same time but sharing the same Host stack.

Source tree layout The stack is split up as follows in the source tree:

subsys/bluetooth/host
The host stack. This is where the HCI command and event handling as well as connection tracking happens. The implementation of the core protocols such as L2CAP, ATT, and SMP is also here.

subsys/bluetooth/controller
Bluetooth LE Controller implementation. Implements the controller-side of HCI, the Link Layer as well as access to the radio transceiver.

include/zephyr/bluetooth/
Public API header files. These are the header files applications need to include in order to use Bluetooth functionality.

drivers/bluetooth/
HCI transport drivers. Every HCI transport needs its own driver. For example, the two common types of UART transport protocols (3-Wire and 5-Wire) have their own drivers.

samples/bluetooth/
Sample Bluetooth code. This is a good reference to get started with Bluetooth application development.

tests/bluetooth/
Test applications. These applications are used to verify the functionality of the Bluetooth stack, but are not necessary the best source for sample code (see samples/bluetooth instead).

doc/services/connectivity/bluetooth/
Extra documentation, such as PICS documents.

LE Host

The Bluetooth Host implements all the higher-level protocols and profiles, and most importantly, provides a high-level API for applications. The following diagram depicts the main protocol & profile layers of the host.

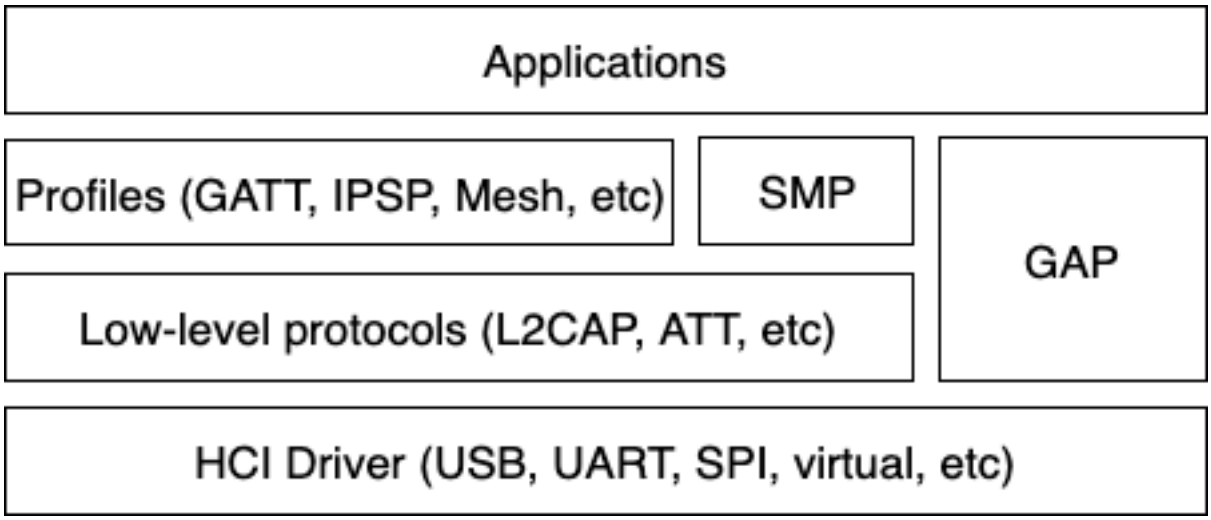


Fig. 14: Bluetooth Host protocol & profile layers.

Lowest down in the host stack sits a so-called HCI driver, which is responsible for abstracting away the details of the HCI transport. It provides a basic API for delivering data from the controller to the host, and vice-versa.

Perhaps the most important block above the HCI handling is the Generic Access Profile (GAP). GAP simplifies Bluetooth LE access by defining four distinct roles of Bluetooth usage:

- Connection-oriented roles
  - Peripheral (e.g. a smart sensor, often with a limited user interface)
  - Central (typically a mobile phone or a PC)
- Connection-less roles
  - Broadcaster (sending out Bluetooth LE advertisements, e.g. a smart beacon)
  - Observer (scanning for Bluetooth LE advertisements)

Each role comes with its own build-time configuration option: `CONFIG_BT_PERIPHERAL`, `CONFIG_BT_CENTRAL`, `CONFIG_BT_BROADCASTER` & `CONFIG_BT_OBSERVER`. Of the connection-oriented roles central implicitly enables observer role, and peripheral implicitly enables broadcaster role. Usually the first step when creating an application is to decide which roles are needed and go from there. Bluetooth Mesh is a slightly special case, requiring at least the observer and broadcaster roles, and possibly also the Peripheral role. This will be described in more detail in a later section.

Peripheral role Most Zephyr-based Bluetooth LE devices will most likely be peripheral-role devices. This means that they perform connectable advertising and expose one or more GATT

services. After registering services using the `bt_gatt_service_register()` API the application will typically start connectable advertising using the `bt_le_adv_start()` API.

There are several peripheral sample applications available in the tree, such as samples/bluetooth/peripheral_hr.

Central role Central role may not be as common for Zephyr-based devices as peripheral role, but it is still a plausible one and equally well supported in Zephyr. Rather than accepting connections from other devices a central role device will scan for available peripheral device and choose one to connect to. Once connected, a central will typically act as a GATT client, first performing discovery of available services and then accessing one or more supported services.

To initially discover a device to connect to the application will likely use the `bt_le_scan_start()` API, wait for an appropriate device to be found (using the scan callback), stop scanning using `bt_le_scan_stop()` and then connect to the device using `bt_conn_le_create()`.

There are some sample applications for the central role available in the tree, such as samples/bluetooth/central_hr.

Observer role An observer role device will use the `bt_le_scan_start()` API to scan for device, but it will not connect to any of them. Instead it will simply utilize the advertising data of found devices, combining it optionally with the received signal strength (RSSI).

Broadcaster role A broadcaster role device will use the `bt_le_adv_start()` API to advertise specific advertising data, but the type of advertising will be non-connectable, i.e. other device will not be able to connect to it.

Connections Connection handling and the related APIs can be found in the *Connection Management* section.

Callback execution contexts The Host delivers events to the application through registered callbacks, for example those in `bt_conn_cb`, `bt_le_scan_cb` or `bt_l2cap_chan_ops`. Unless documented otherwise, these callbacks are invoked from a thread context, never from an ISR. Most run in a context internal to the stack, but some are today invoked synchronously from within the API call that triggers them, and so run in the calling thread: for example `bt_unpair()` invokes `bond_deleted`, and `bt_gatt_unsubscribe()` can invoke `notify` with `NULL` data, before returning. Neither behavior is part of the API: a callback delivered synchronously today may be deferred to a stack-internal context in a future release, or vice versa, and the specific thread has changed between releases before and may change again. Applications should rely only on the guarantees above, not on being called from, or before the return of, anything in particular.

When a callback runs in a context internal to the stack, that context is shared with the stack's own processing: time spent in the callback delays other Bluetooth activity, and blocking carries an additional risk, since a wait that only Bluetooth processing itself can satisfy becomes a deadlock, because that processing cannot proceed until the callback returns. The classic example is allocating a buffer with `K_FOREVER` from a pool that is replenished by the same context that runs the callback. As applications cannot rely on which context a given callback uses, the practices below apply to every callback.

Calling Bluetooth APIs from callbacks is common and supported, even though most of them may block; the blocking risk is then managed rather than avoided:

- Keep callbacks short, and defer work that is long-running or blocks indefinitely to an application-owned thread or work queue.
- Prefer allocations with `K_NO_WAIT` or a bounded timeout over `K_FOREVER`, and handle the failure.
- Size buffer pools (for example `CONFIG_BT_L2CAP_TX_BUF_COUNT` or `CONFIG_BT_ATT_TX_COUNT`) so that allocations made from callbacks do not have to wait.

Security To achieve a secure relationship between two Bluetooth devices a process called pairing is used. This process can either be triggered implicitly through the security properties of GATT services, or explicitly using the `bt_conn_set_security()` API on a connection object.

To achieve a higher security level, and protect against Man-In-The-Middle (MITM) attacks, it is recommended to use some out-of-band channel during the pairing. If the devices have a sufficient user interface this "channel" is the user itself. The capabilities of the device are registered using the `bt_conn_auth_cb_register()` API. The `bt_conn_auth_cb` struct that's passed to this API has a set of optional callbacks that can be used during the pairing - if the device lacks some feature the corresponding callback may be set to NULL. For example, if the device does not have an input method but does have a display, the `passkey_entry` and `passkey_confirm` callbacks would be set to NULL, but the `passkey_display` would be set to a callback capable of displaying a passkey to the user.

Depending on the local and remote security requirements & capabilities, there are four possible security levels that can be reached:

`BT_SECURITY_L1`
: No encryption and no authentication.

`BT_SECURITY_L2`
: Encryption but no authentication (no MITM protection).

`BT_SECURITY_L3`
: Encryption and authentication using the legacy pairing method from Bluetooth 4.0 and 4.1.

`BT_SECURITY_L4`
: Encryption and authentication using the LE Secure Connections feature available since Bluetooth 4.2.

> **Note**
>
> Mesh has its own security solution through a process called provisioning. It follows a similar procedure as pairing, but is done using separate mesh-specific APIs.

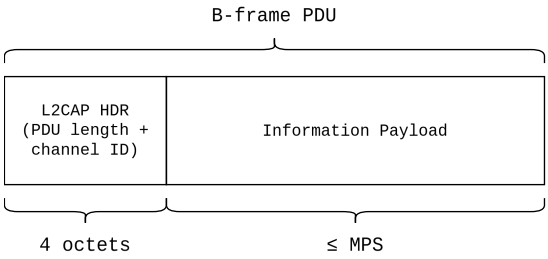
L2CAP L2CAP stands for the Logical Link Control and Adaptation Protocol. It is a common layer for all communication over Bluetooth connections, however an application comes in direct contact with it only when using it in the so-called Connection-oriented Channels (CoC) mode. More information on this can be found in the *L2CAP API section*.

Terminology The definitions are from the Core Specification version 5.4, volume 3, part A 1.4.

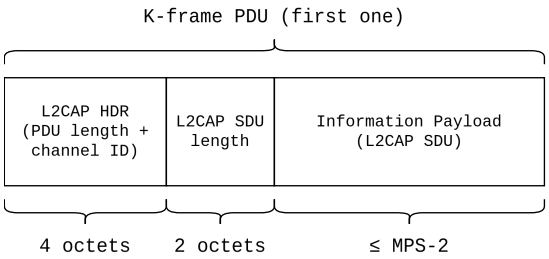
Term	Description
Upper layer	Layer above L2CAP, it exchanges data in form of SDUs. It may be an application or a higher level protocol.
Lower layer	Layer below L2CAP, it exchanges data in form of PDUs (or fragments). It is usually the HCI.
Service Data Unit (SDU)	Packet of data that L2CAP exchanges with the upper layer. This term is relevant only in Enhanced Retransmission mode, Streaming mode, Retransmission mode and Flow Control Mode, not in Basic L2CAP mode.
Protocol Data Unit (PDU)	Packet of data containing L2CAP data. PDUs always start with Basic L2CAP header. Types of PDUs for LE: *B-frames* and *K-frames*. Types of PDUs for BR/EDR: I-frames, S-frames, C-frames and G-frames.
Maximum Transmission Unit (MTU)	Maximum size of an SDU that the upper layer is capable of accepting.
Maximum Payload Size (MPS)	Maximum payload size that the L2CAP layer is capable of accepting. In Basic L2CAP mode, the MTU size is equal to MPS. In credit-based channels without segmentation, the MTU is MPS minus 2.
Basic L2CAP header	Present at the beginning of each PDU. It contains two fields, the PDU length and the Channel Identifier (CID).

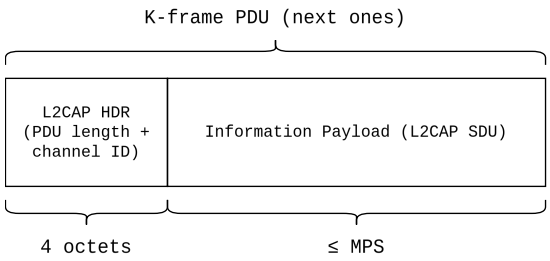
PDU Types

B-frame: Basic information frame PDU used in Basic L2CAP mode. It contains the payload received from the upper layer or delivered to the upper layer as its payload.



K-frame: Credit-based frame PDU used in LE Credit Based Flow Control mode and Enhanced Credit Based Flow Control mode. It contains a SDU segment and additional protocol information.



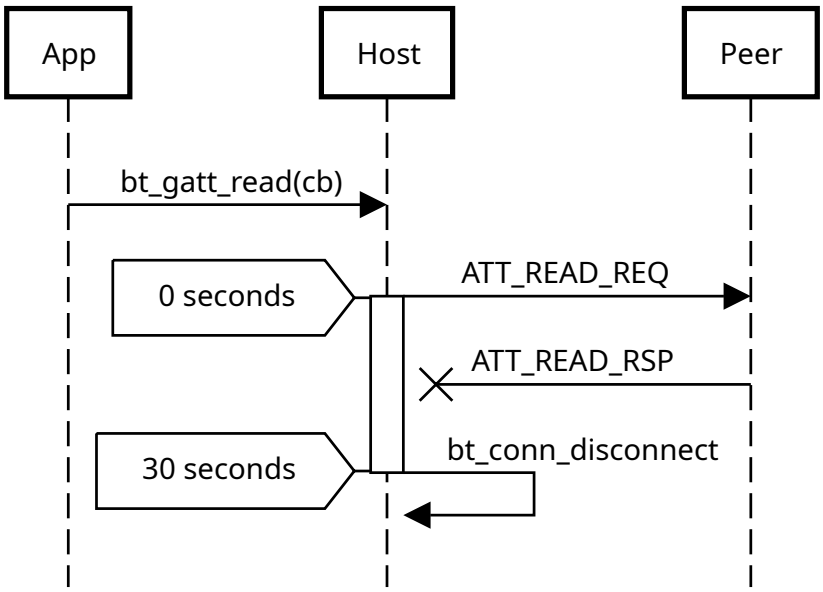


Relevant Kconfig

Kconfig symbol	Description
CONFIG_BT_BUF_ACL_RX_SIZE	Represents the MPS
CONFIG_BT_L2CAP_TX_MTU	Represents the L2CAP MTU
CONFIG_BT_L2CAP_DYNAMIC_CHANN	Enables LE Credit Based Flow Control and thus the stack may use *K-frame* PDUs

GATT The Generic Attribute Profile is the most common means of communication over LE connections. A more detailed description of this layer and the API reference can be found in the *GATT API reference section*.

ATT timeout If the peer device does not respond to an ATT request (such as read or write) within the ATT timeout, the host will automatically initiate a disconnect. This simplifies error handling by reducing rare failure conditions to a common disconnection, allowing developers to manage unexpected disconnects without special cases for ATT timeouts.



Mesh Mesh is a little bit special when it comes to the needed GAP roles. By default, mesh requires both observer and broadcaster role to be enabled. If the optional GATT Proxy feature is desired, then peripheral role should also be enabled.

The API reference for mesh can be found in the *Mesh API reference section*.

LE Audio The LE audio is a set of profiles and services that utilizes GATT and Isochronous Channel to provide audio over Bluetooth Low Energy. The architecture and API references can be found in *Bluetooth Audio Architecture*.

Persistent storage The Bluetooth host stack uses the settings subsystem to implement persistent storage to flash. This requires the presence of a flash driver and a designated "storage" partition on flash. A typical set of configuration options needed will look something like the following:

```
CONFIG_BT_SETTINGS=y
CONFIG_FLASH=y
CONFIG_FLASH_PAGE_LAYOUT=y
CONFIG_FLASH_MAP=y
CONFIG_NVS=y
CONFIG_SETTINGS=y
```

Once enabled, it is the responsibility of the application to call settings_load() after having initialized Bluetooth (using the `bt_enable()` API).

LE Audio Stack

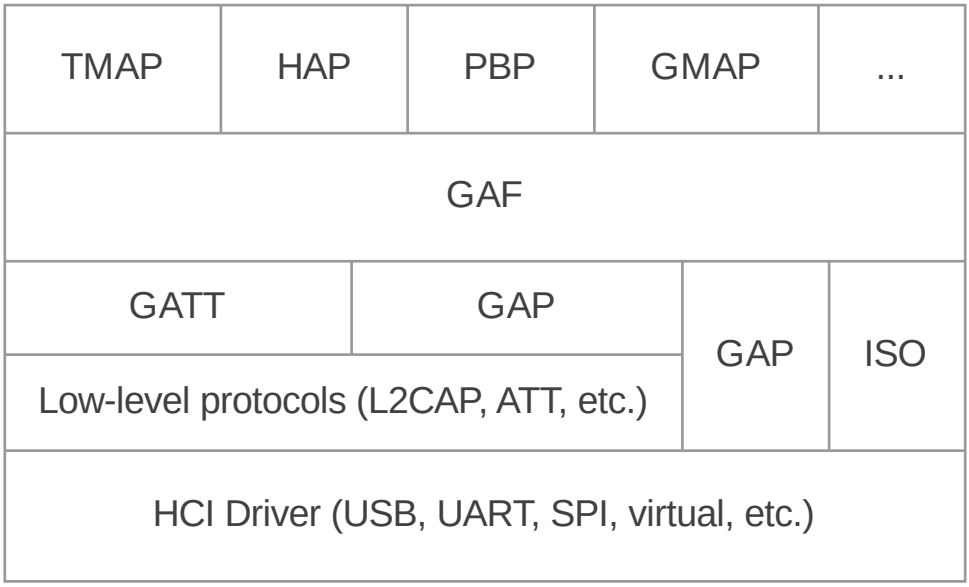


Fig. 15: Bluetooth Audio Architecture

Overall design The overall design of the LE Audio stack is that the implementation follows the specifications as closely as possible, both in terms of structure but also naming. Most API functions are prefixed by the specification acronym (e.g. `bt_bap` for the Basic Audio Profile (BAP) and `bt_vcp` for the Volume Control Profile (VCP)). The functions are then further prefixed with the specific role from each profile where applicable (e.g. `bt_bap_unicast_client_discover()` and `bt_vcp_vol_rend_set_vol()`). There are usually a function per procedure defined by the profile or service specifications, and additional helper or meta functions that do not correspond to procedures.

The structure of the files generally also follow this, where BAP related files are prefixed with `bap` and VCP related files are prefixed with `vcp`. If the file is specific for a profile role, the role is also embedded in the file name.

Generic Audio Framework (GAF) The Generic Audio Framework (GAF) is considered the middleware of the Bluetooth LE Audio architecture. The GAF contains the profiles and services that allows higher layer applications and profiles to set up streams, change volume, control media and telephony and more. The GAF builds on GATT, GAP and isochronous channels (ISO).

GAF uses GAP to connect, advertise and synchronize to other devices. GAF uses GATT to configure streams, associate streams with content (e.g. media or telephony), control volume and more. GAF uses ISO for the audio streams themselves, both as unicast (connected) audio streams or broadcast (unconnected) audio streams.

GAF mandates the use of the LC3 codec, but also supports other codecs.

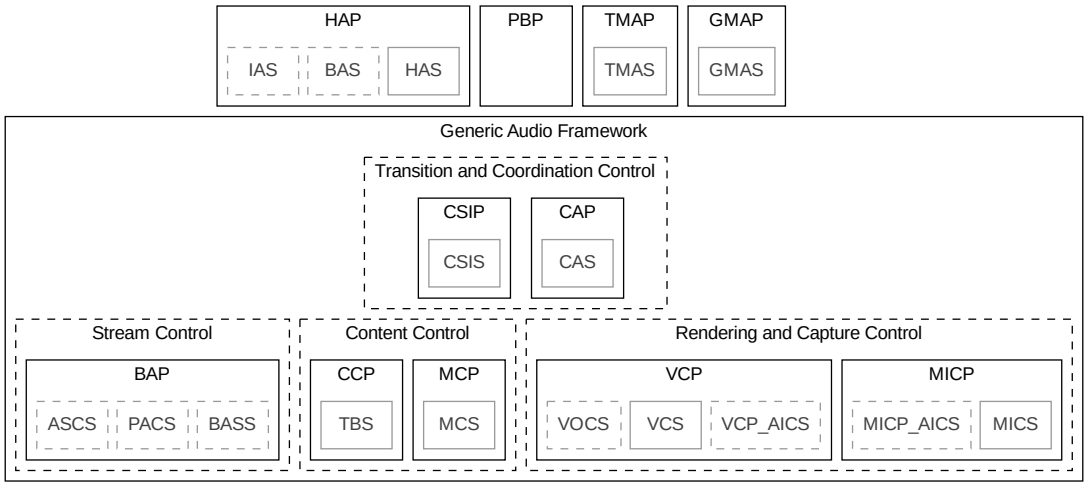


Fig. 16: Generic Audio Framework (GAF)

The top-level profiles TMAP and HAP are not part of the GAF, but rather provide top-level requirements for how to use the GAF.

GAF and the top layer profiles have been implemented in Zephyr with the following structure.

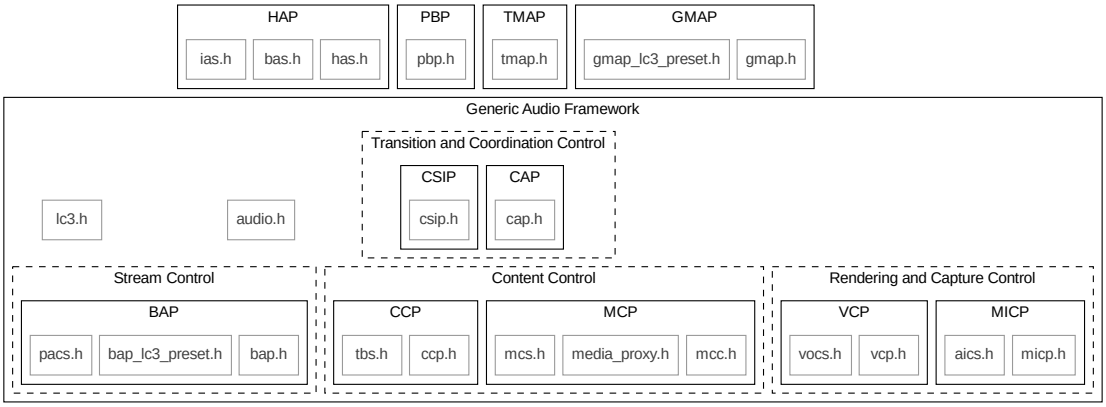


Fig. 17: Zephyr Generic Audio Framework

Profile Dependencies The LE Audio profiles depend on other profiles and services, as outlined in the following tables. In these tables 'Server' refers to acting in the GATT server role, and 'Client'

refers to acting in the GATT client role for the specific service. If a profile role depends on another profile that depends on a service, then that dependency is implicitly also applied to that profile. For example, if the CAP Acceptor uses the BAP Unicast Server role, then the requirements on the ASCS Server and PACS Server also apply to the CAP Acceptor.

The dependencies for Stream Control (BAP) are in the following table.

Table 15: BAP dependencies

	Unicast Server	Unicast Client	Broadcast Source	Broadcast Sink	Scan Delegator	Broadcast Assistant
BAP Scan Delegator				M		
ASCS Client		M				
ASCS Server	M					
PACS Client		M				O
PACS Server	M			M		
BASS Client						M
BASS Server					M	

Note:

- As the table shows, the Broadcast Source role has no dependencies on other LE Audio profiles or services

The dependencies for Content Control (MCP and CCP) are in the following tables.

Table 16: MCP dependencies

	Media Control Server	Media Control Client
GMCS Server	M	
GMCS Client		M
MCS Server	O	
MCS Client		O
OTS Server	O	
OTS Client		O

Table 17: CCP dependencies

	Call Control Server	Call Control Client
GTBS Server	M	
GTBS Client		M
TBS Server	M	
TBS Client		M

The dependencies for Rendering Control (MICP and VCP) are in the following tables.

Table 18: MICP dependencies

	Microphone Controller	Microphone Device
MICS Server	M	
MICS Client		M
AICS Server	O	
AICS Client		O

Table 19: VCP dependencies

	Volume Renderer	Volume Controller
VCS Server	M	
VCS Client		M
VOCS Server	O	
VOCS Client		O
AICS Server	O	
AICS Client		O

The last element in GAF is Transition and Coordination Control (CAP and CSIP) with the dependencies from the following tables.

Table 20: CAP dependencies

	Acceptor	Initiator	Commander
CAS Server	M		C.8
CAS Client		M	M
BAP Unicast Client		C.1	
BAP Unicast Server	C.2		
BAP Broadcast Source		C.1	
BAP Broadcast Sink	C.2		
BAP Broadcast Assistant			C.4, C.6
BAP Scan Delegator	C.3		C.6
VCP Volume Controller			C.6
VCP Volume Renderer	O		
MICP Microphone Controller			C.6
MICP Microphone Device	O		
CCP Call Control Server		O	
CCP Call Control Client	O		C.6
MCP Media Control Server		O	
MCP Media Control Client	O		C.6
CSIP Set Coordinator		C.5	M
CSIP Set Member	C.7		

Notes:

- C.1: Support at least one of BAP Unicast Client or BAP Broadcast Source
- C.2: Support at least one of BAP Unicast Server or BAP Broadcast Sink
- C.3: Mandatory if BAP Broadcast Sink
- C.4: Mandatory if BAP Scan Delegator
- C.5: Mandatory if BAP Unicast Client
- C.6: Support at least one
- C.7: Mandatory if part of a coordinated set
- C.8: Mandatory if the Commander transmits CAP announcements

Table 21: CSIP dependencies

	Set Member	Set Coordinator
CSIS Server	M	
CSIS Client		M

The dependencies of the higher level profiles (GMAP, HAP, PBP and TMAP) are listed in the following tables.

Table 22: GMAP dependencies

	Unicast Game Gateway	Unicast Game Terminal	Broadcast Game Sender	Broadcast Game Receiver
GMAS Server	M	M	O	M
GMAS Client	M	O	O	O
CAP Initiator	M		M	
CAP Acceptor		M		M
CAP Commander	M		M	
BAP Broadcast Source			M	
BAP Broadcast Sink				M
BAP Unicast Client	M			
BAP Unicast Server		M		
VCP Volume Controller	M			
VCP Volume Renderer		C.1		M
MICP Microphone Controller	O			
MICP Microphone Device		C.2		

Notes:

- C.1 Mandatory if the UGT supports the UGT Sink feature
- C.2 Optional if the UGT supports the UGT Source feature

Table 23: HAP dependencies

	Hearing Aid	Hearing Aid Unicast Client	Hearing Aid Remote Controller
HAS Client			M
HAS Server	M		
CAP Initiator		M	
CAP Acceptor	M		
CAP Commander			M
BAP Unicast Client		M	
BAP Unicast Server	M		
VCP Volume Controller			M
VCP Volume Renderer	M		
VOCS Server	C.1		
AICS Server	O		
MICP Microphone Controller			O
MICP Microphone Device	C.2		
CCP Call Control Client	O		
CCP Call Control Server		O	
CSIP Set Coordinator		M	M
CSIP Set Member	C.3		
BAS Server	C.4		
IAS Server	O		

Notes:

- C.1 Mandatory if the HA supports the Volume Baslance feature and is part of a Binaural Hearing Aid Set
- C.2 Mandatory if the HA supports the BAP Audio Source Role
- C.3 Mandatory if the HA is capable of being part of a Binaural Hearing Aid set
- C.4 If equipped with batteries
- C.5 If CCP Call Control Server is supported

Table 24: PBP dependencies

	Public Broadcast Source	Public Broadcast sink	Public Broadcast Assistant
CAP Initiator	M		
CAP Acceptor		M	
CAP Commander			M
BAP Broadcast Assistant			M

Table 25: TMAP dependencies

	Call Gateway	Call Terminal	Unicast Media Sender	Unicast Media Receiver	Broadcast Media Sender	Broadcast Media Receiver
TMAS Server	M	M	M	M	O	M
TMAS Client	O	O	O	O	O	O
CAP Initiator	M		M		M	
CAP Acceptor		M		M		M
CAP Commander	M	O	M	O	O	O
BAP Broadcast Source					M	
BAP Broadcast Sink						M
BAP Unicast Client	M		M			
BAP Unicast Server		M		M		
VCP Volume Controller	M		M			
VCP Volume Renderer		C.1		M		M
MCP Media Control Server			M			
CCP Call Control Server	M					

Notes:

- C.1 Mandatory to support if the BAP Unicast Server is acting as an Audio Sink

Bluetooth Audio Stack Status The following table shows the current status and support of the profiles in the Bluetooth Audio Stack.

Table 26: Bluetooth Audio Profile status

Module	Role	Version	Added in Release	Status	Remaining
VCP	Volume Renderer	1.0.0	2.6	• Feature complete • Shell Module • BSIM test	• Sample Application
	Volume Controller	1.0.0	2.6	• Feature complete • Shell Module • BSIM test	• Sample Application
MICP	Microphone Device	1.0.0	2.7	• Feature complete • Shell Module • BSIM test	• Sample Application
	Microphone Controller	1.0.0	2.7	• Feature complete • Shell Module • BSIM test	• Sample Application
CSIP	Set Member	1.0.1	3.0	• Feature complete • Shell Module • BSIM test	• Sample Application
	Set Coordinator	1.0.1	3.0	• Feature complete • Shell Module • BSIM test	• Sample Application

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
CCP	Call Control Server	1.0.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor (in progress) • Sample Application (in progress)
	Call Control Client	1.0.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor (in progress) • Sample Application (in progress)
MCP	Media Control Server	1.0.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor • Support for multiple instances and connections • Sample Application
	Media Control Client	1.0.0	3.0	• Feature complete • Shell Module • BSIM test	• API refactor • Sample Application

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
BAP	Unicast Server	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Unicast Client	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Source	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Sink	1.0.1	3.0	• Feature complete • Shell Module • BSIM test • Sample Application	
	Scan Delegator	1.0.1	3.3	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Assistant	1.0.1	3.3	• Feature complete • Shell Module • BSIM test • Sample Application	

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
CAP	Acceptor	1.0.1	3.2	• Feature complete • Shell Module • BSIM test • Sample Application	
	Initiator	1.0.1	3.3	• Feature complete • Shell Module • BSIM test • Sample Application	
	Commander			• Feature complete • BSIM test	• Shell Module • Sample Application

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
HAP	Hearing Aid	1.0.0	3.1	• Feature complete • Shell Module • BSIM test • Sample Application	
	Hearing Aid Unicast Client	1.0.0	3.1	• Feature complete • Shell Module • BSIM test • Sample Application	
	Hearing Aid Remote Controller			• WIP	• Feature complete • Shell Module • BSIM test • Sample Application

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
TMAP	Call Gateway	1.0.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Call Terminal	1.0.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Unicast Media Sender	1.0.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Unicast Media Receiver	1.0.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Media Sender	1.0.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	
	Broadcast Media Receiver	1.0.0	3.4	• Feature complete • Shell Module • BSIM test • Sample Application	

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
PBP	Public Broadcast Source	1.0.2	3.5	• Feature complete • Shell Module • BSIM test • Sample Application	
	Public Broadcast Sink	1.0.2	3.5	• Feature complete • Shell Module • BSIM test • Sample Application	
	Public Broadcast Assistant				• Feature complete • Shell Module • BSIM test • Sample Application

continues on next page

Table 26 – continued from previous page

Module	Role	Version	Added in Release	Status	Remaining
GMAP	Unicast Game Gateway	1.0.1	3.5	• Feature complete • Shell Module • BSIM test	• Sample Application
	Unicast Game Terminal	1.0.1	3.5	• Feature complete • Shell Module • BSIM test	• Sample Application
	Broadcast Game Sender	1.0.1	3.5	• Feature complete • Shell Module • BSIM test	• Sample Application
	Broadcast Game Receiver	1.0.1	3.5	• Feature complete • Shell Module • BSIM test	• Sample Application

Using the Bluetooth Audio Stack To use any of the profiles in the Bluetooth Audio Stack, including the top-level profiles outside of GAF, `CONFIG_BT_AUDIO` shall be enabled. This Kconfig option allows the enabling of the individual profiles inside of the Bluetooth Audio Stack. Each profile can generally be enabled on its own, but enabling higher-layer profiles (such as CAP, TMAP and HAP) will typically require enabling some of the lower layer profiles.

It is, however, possible to create a device that uses e.g. only Stream Control (with just the BAP), without using any of the content control or rendering/capture control profiles, or vice versa. Using the higher layer profiles will however typically provide a better user experience and better interoperability with other devices.

Common Audio Profile (CAP) The Common Audio Profile introduces restrictions and requirements on the lower layer profiles. The procedures in CAP works on one or more streams for one or more devices. Is it thus possible via CAP to do a single function call to setup multiple streams across multiple devices.

The figure below shows a complete structure of the procedures in CAP and how they correspond to procedures from the other profiles. The circles with I, A and C show whether the procedure has active involvement or requirements from the CAP Initiator, CAP Accept and CAP Commander roles respectively.

The API reference for CAP can be found in *Common Audio Profile*.

Stream Control (BAP) Stream control is implemented by the Basic Audio Profile. This profile defines multiple roles:

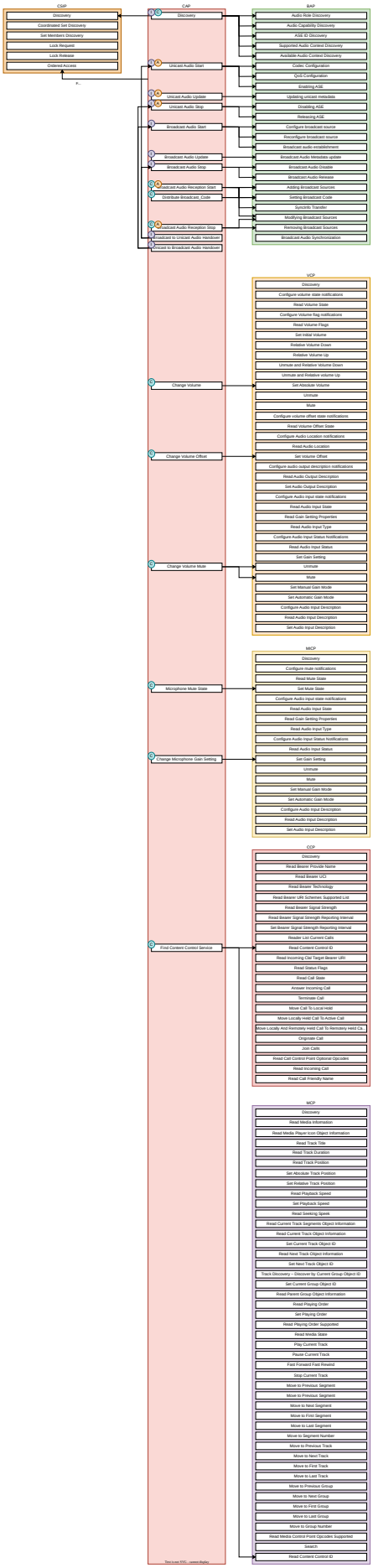


Fig. 18: Common Audio Profile Procedures

- Unicast Client
- Unicast Server
- Broadcast Source
- Broadcast Sink
- Scan Delegator
- Broadcast Assistant

Each role can be enabled individually, and it is possible to support more than one role.

Notes about the stream control services There are 3 services primarily used by stream control using the Basic Audio Profile.

Audio Stream Control Service (ASCS) ASCS is a service used exclusively for setting up unicast streams, and is located on the BAP Unicast Server device. The service exposes one or more endpoints that can either be a sink or source endpoint, from the perspective of the Unicast Server. That means a sink endpoint is always audio from the Unicast Client to the Unicast Server, and a source endpoint is always from the Unicast Server to the Unicast Client.

Unlike most other GATT services, ASCS requires that each characteristic in the service has unique data per client. This means that if a Unicast Server is connected to multiple Unicast Clients, the Unicast Clients are not able to see or control the endpoints configured by the other clients. For example if a person's smartphone is streaming audio to a headset, then the same person will not be able to see or control that stream from their smartwatch.

Broadcast Audio Scan Service (BASS) BASS is a service that is exclusively used by the Scan Delegator and Broadcast Assistant. The main purpose of the service is to offload scanning from low power peripherals to e.g. phones and PCs. Unlike ASCS where the data is required to be unique per client, the data in BASS (called receive states) are (usually) shared among all connected clients. That means it is possible for a person to tell their headphones to synchronize to a Broadcast Source using their phone, and then later tell their headphones to stop synchronizing using their smartwatch.

A Broadcast Assistant can be any device, and may only support this one role without any audio capabilities. This allows legacy devices that do not support periodic advertisements or isochronous channels to still provide an interface and scan offloading for peripherals. The Bluetooth SIG has provided a guide on how to develop such legacy Broadcast Assistants that can be found at https://www.bluetooth.com/bluetooth-resources/developing-auracast-receivers-with-an-assistant-application-for-legacy-smartphones/. An important note about this guide is that many operating systems (especially on phones), do not allow generic usage of the BASS UUID, effectively making it impossible to implement your own Broadcast Assistant, because you cannot access the BASS.

Published Audio Capabilities Service (PACS) PACS is used to expose a device's audio capabilities in Published Audio Capabilities (PAC) records. PACS is used by nearly all roles, where the Unicast Client and Broadcast Assistant will act as PACS clients, and Unicast Server and Broadcast Sink will act as PACS servers. These records contain information about the codec, and which values are supported by each codec. The values for the LC3 codec are defined by the Bluetooth Assigned numbers (https://www.bluetooth.com/specifications/assigned-numbers/), and the values for other codecs such as SBC are left undefined/implementation specific for BAP.

PACS also usually shares the same data between each connected client, but by using functions such as `bt_pacs_conn_set_available_contexts_for_conn()`, it is possible to set specific values for specific clients.

The API reference for stream control can be found in *Bluetooth Audio*.

Rendering and Capture Control Rendering and capture control is implemented by the Volume Control Profile (VCP) and Microphone Control Profile (MICP).

The VCP implementation supports the following roles

- Volume Control Service (VCS) Server
- Volume Control Service (VCS) Client

The MICP implementation supports the following roles

- Microphone Control Profile (MICP) Microphone Device (server)
- Microphone Control Profile (MICP) Microphone Controller (client)

The API reference for volume control can be found in *Bluetooth Volume Control*.

The API reference for Microphone Control can be found in *Bluetooth Microphone Control*.

Content Control Content control is implemented by the Call Control Profile (CCP) and Media Control Profile (MCP).

The CCP implementation is not yet implemented in Zephyr.

The MCP implementation supports the following roles

- Media Control Service (MCS) Server via the Media Proxy module
- Media Control Client (MCC)

The API reference for media control can be found in *Bluetooth Media Control*.

Generic TBS and Generic MCS Both the Telephone Bearer Service (TBS) used by CCP and the Media Control Service (MCS) used by MCP have the concept of generic instances of the services called Generic TBS (GTBS) and Generic MCS (GMCS).

While these share a common name prefix, the behavior of these two may be significantly different.

Generic TBS The TBS spec defines GTBS as

> GTBS provides a single point of access and exposes a representation of its internal telephone bearers into a single telephone bearer. This service provides telephone status and control of the device as a single unit with a single set of characteristics. It is left up to the implementation to determine what telephone bearer a characteristic of GTBS represents at any time. There is no specified manner of representing a characteristic from each individual TBS that resides on the device to the same characteristic of the GTBS.
>
> For example, if there is more than one TBS on a device and each has a unique telephone bearer name (e.g., Name1 and Name2), the way the GTBS represents the telephone bearer name is left up to the implementation. GTBS is suited for clients that do not need to access or control all the information available on specific telephone bearers.

This means that a GTBS instance represents one or more telephone bearers. A telephone bearer could be any application on a device that can handle (telephone) calls, such as the default Call application on a smartphone, but also other applications such as Signal, Discord, Teams, Slack, etc.

GTBS may be standalone (i.e. the device only has a GTBS instance without any TBS instances), and the behavior of the GTBS is mostly left up to the implementation. In Zephyr the implementation of GTBS is that it contains some generic information, such as the provider name which is defined to simply be “Generic TBS”, but the majority of the information in the GTBS instance in Zephyr has been implemented to be a union of the data of the other bearers. For example if you have a bearer for regular phone calls and Teams and have an active call in both bearers, then each of

those bearers will report a single call, but the GTBS instance will report 2 calls, making it possible for a simple Call Control Client to control all calls from a single bearer. Similarly the supported URIs for each bearer are also made into a union in GTBS, and when placing a call using the GTBS the server will pick the most suited bearer depending on the URI. For example calls with URI `tel` would go to the regular phone application, and calls with the URI `skype` would go to the Teams application.

In conclusion the GTBS implementation in Zephyr is a union of the non-generic telephone bearers.

Generic MCS The MCS spec defines GMCS as

> The GMCS provides status and control of media playback for the device as a single unit. An MCS instance describes and controls the media playback for a specific media player within the device. A device implements MCS instances to allow clients to access the separate internal media player entities.

and where the behavior of GMCS is defined as

> … the behavior of MCS and GMCS is identical, and all the characteristics and the characteristics' behaviors are the same. The term "MCS" is used throughout the document. Unless otherwise specifically stated in this specification, the same meaning applies to GMCS as well.

This means that a GMCS instance works the same way as an MCS instance, and it follows that GMCS

> controls the media playback for a specific media player within the device

A media player on a device could be anything that plays media, such as a Spotify or Youtube application on a smartphone. Thus if a device has multiple MCS instances, then each of these control media for that specific application, but the GMCS also controls media playback for a specific media player. GMCS can thus be considered a pointer to a specific MCS instance, and control either e.g. Spotify or Youtube, but not both.

The MCS spec does however provide an example of GMCS where a device can

> Implement a GMCS that provides status and control of media playback for the device as a whole.

Which may indicate that an implementation may use GMCS to represent all media players with GMCS and not a specific media player as stated above. In the case where a device does not have any MCS instances and only GMCS, then GMCS will point to a generic instance.

The Zephyr implementation of MCS and GMCS is incomplete, and currently only supports instantiating a single instance that can either be an MCS or GMCS. This means that the implementation is neither complete nor spec-compliant.

Difference between GTBS and GMCS The definitions and implementations of GTBS and GMCS as stated above are notably different. GTBS works as a union between the other TBS instances (if any), and GMCS works as a pointer to a specific MCS instance (if any). This effectively means that a simple Call Control Client can control all calls just using GTBS, but a Media Control Client may only be able to control a single player using GMCS.

Coordinated Sets Coordinated Sets is implemented by the Coordinated Sets Identification Profile (CSIP).

The CSIP implementation supports the following roles

- Coordinated Set Identification Service (CSIP) Set Member
- Coordinated Set Identification Service (CSIP) Set Coordinator

The API reference for media control can be found in *Bluetooth Coordinated Sets*.

Specification correctness and data location The implementations are designed to ensure specification compliance as much as possible. When a specification introduces a requirement with e.g. a **shall** then the implementation should attempt to ensure that this requirement is always followed. Depending on the context of this, the implementation ensures this by rejecting invalid parameters from the application, or from the remote devices.

Some requirements from the specifications are not or can not be handled by the stack itself for various reasons. One reason when the stack cannot handle a requirement is if the data related to the requirement is exclusively controlled by the application. An example of this is the advertising data, where multiple services have requirements for what to advertise and when, but where both the advertising state and data is exclusively controlled by the application.

Oppositely there are also requirements from the specification, where the data related to the requirement is exclusively controlled by the stack. An example of this is the Volume Control Service (VCS) state, where the specifications mandate that the VCP Volume Renderer (VCS server) modify the values without a choice, e.g. when setting the absolutely volume. In cases like this the application is only notified about the change with a callback, but cannot reject the request (the stack will reject any invalid requests).

Generally when the data is simple (like the VCS state which only takes up a few bytes), the data is kept in and controlled by the stack, as this can ensure that the requirements can be handled by the stack, making it easier to use a profile role correctly. When the data is more complex (e.g. the PAC records), the data may be kept by the application and the stack only contains a reference to it. When the data is very application specific (e.g. advertising data), the data is kept in and controlled by the application.

As a rule of thumb, the return types of the callbacks for each profile implementation indicate whether the data is controlled by the stack or the application. For example all the callbacks for the VCP Volume Renderer have the return type of `void`, but the return type of the BAP Unicast Server callbacks are `int`, indicating that the application not only controls a lot of the Unicast Server data, but can also reject the requests. The choice of what the return type of the callbacks often depend on the specifications, and how much control the role has in a given context.

Things worth knowing or considering when using LE Audio This section describes a few things to consider when contributing to or using LE Audio in Zephyr. The things described by this section are not unique to Zephyr as they are defined by the specifications.

Security requirements All LE Audio services require Security Level 2 but where the key must be 128-bit and derived via an OOB method or via LE Secure connections. There is no Core-spec defined way of reporting this in GATT, as ATT does not have a specific error code for missing OOB method or LE Secure Connections (although there is a way to report wrong key size).

In Zephyr we do not force the device to always use these, as a device that uses LE Audio may also use other profiles and services that do not require such security. We guard all access to services using a custom security check implemented in subsys/bluetooth/audio/audio.c, where all LE Audio services must use the internal `BT_AUDIO_CHRC` macro for proper security verification.

Access to the LTK for encrypted SIRKs in CSIS The Coordinated Set Identification Service (CSIS) may encrypt the SIRK (set identity resolving key). The process of encrypting the SIRK requires the LTK as the encryption key, which is typically not exposed to higher layer implementations such as CSIS. This does not have any effect on the security though.

MTU requirements The Basic Audio Profile (BAP) has a requirement that both sides shall support a minimum ATT_MTU of at least 64 on the unenhanced ATT bearer or at least one enhanced ATT bearer. The requirement comes from the preferred (or sometimes mandatory) use of GATT Write Without Response, and where support for Write Long Characteristic Value is optional in most cases.

If a ASCS device supports values larger than the minimum ATT_MTU of 64 octets, then it shall support Read long Characteristic Value by setting `CONFIG_BT_ATT_PREPARE_COUNT` to a non-zero value.

LE Audio resources

This section contains some links and reference to resources that are useful for either contributors to the LE Audio Stack in Zephyr, LE Audio application developers or both.

The LE audio channel on Discord Zephyr has a specific Discord channel for LE Audio development, which is open to all. Find it here at https://discordapp.com/channels/720317445772017664/1207326649591271434 or simply search for "bt-audio" from within Discord. Since the `#bt-audio` channel is open for all, we cannot discuss any specifications that are in development in that channel. For discussions that require a Bluetooth SIG membership we refer to the `#bt-sig` Discord channel found at https://discordapp.com/channels/720317445772017664/869172014018097162.

Zephyr weekly meetings Anyone who is a Bluetooth SIG member and a Zephyr member can join the weekly meetings where we discuss and plan the development of LE Audio in Zephyr. You can find the time of the meetings by joining the Bluetooth-sig group at https://lists.zephyrproject.org/g/Bluetooth-sig.

Github project LE Audio in Zephyr has its own Github project available at https://github.com/orgs/zephyrproject-rtos/projects/26. The project is mostly automated, and the LE Audio contributors almost only rely on the automated workflows to present the state of development. Anyone is able to pick any of the open issues and work on it. If you cannot assign the issue to yourself, please leave a comment in the issue itself or ping the Discord channel for help.

Bluetooth SIG errata for LE Audio There are many specifications for LE Audio, and several of them are still being updated and developed. To get an overview of the errata for the LE Audio specifications you can visit

- Generic Audio (GA) errata https://bluetooth.atlassian.net/wiki/spaces/GA/pages/1634402349/GAWG+Errata+Lists
- Hearing Aid (HA) errata https://bluetooth.atlassian.net/wiki/spaces/HA/pages/1634140216/HA+WG+Errata+List
- Audio, Telephony and Automotive (ATA) errata https://bluetooth.atlassian.net/wiki/spaces/ATA/pages/1668481034/ATA+Errata+Lists

Access to errata requires a Bluetooth SIG membership.

Bluetooth SIG working groups for LE Audio There are 3 working groups in the Bluetooth SIG related to LE Audio:

- Generic Audio (GA) https://www.bluetooth.org/groups/group.aspx?gId=665
- Hearing Aid (HA) https://www.bluetooth.org/groups/group.aspx?gId=605
- Audio, Telephony, and Automotive (ATA) https://www.bluetooth.org/groups/group.aspx?gId=659

By joining these groups you will also get emails from their respective mailing lists, where multiple questions and discussions are handled. The working groups also have scheduled weekly meetings, where issues and the development of the specifications are handled.

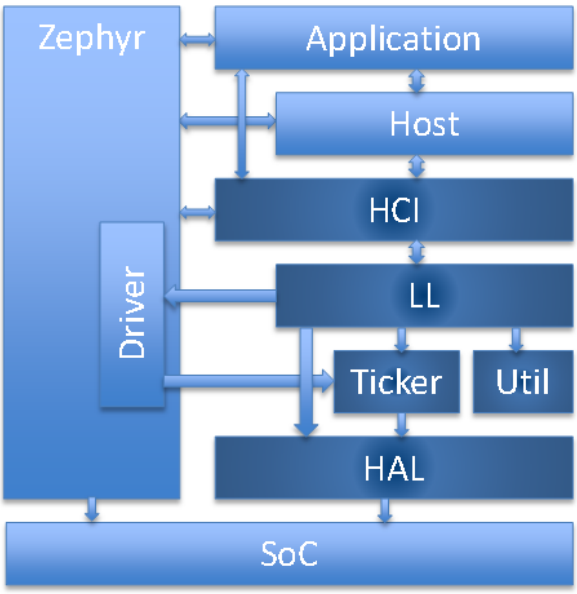
Access to the Bluetooth SIG working groups requires a Bluetooth SIG membership.

The LE Audio Book There is a free ebook on LE Audio at https://www.bluetooth.com/bluetooth-resources/le-audio-book/. The book was released in January 2022, and thus before some of the specifications were finalized, but also before some of the released updates to the specifications. Nevertheless the book still provides a good explanation for many of the concepts and ideas, but please refer to the individual specifications for technical information.

Bluetooth SIG informational papers, reports and guides The Bluetooth SIG occasionally release new informational papers, report and guides. These can be found at https://www.bluetooth.com/bluetooth-resources/?tags=le-audio&keyword. Here you will also find the aforementioned LE Audio book, among many other good resources.

Samples All of the LE Audio samples are located in `samples/bluetooth/audio`.

LE Controller



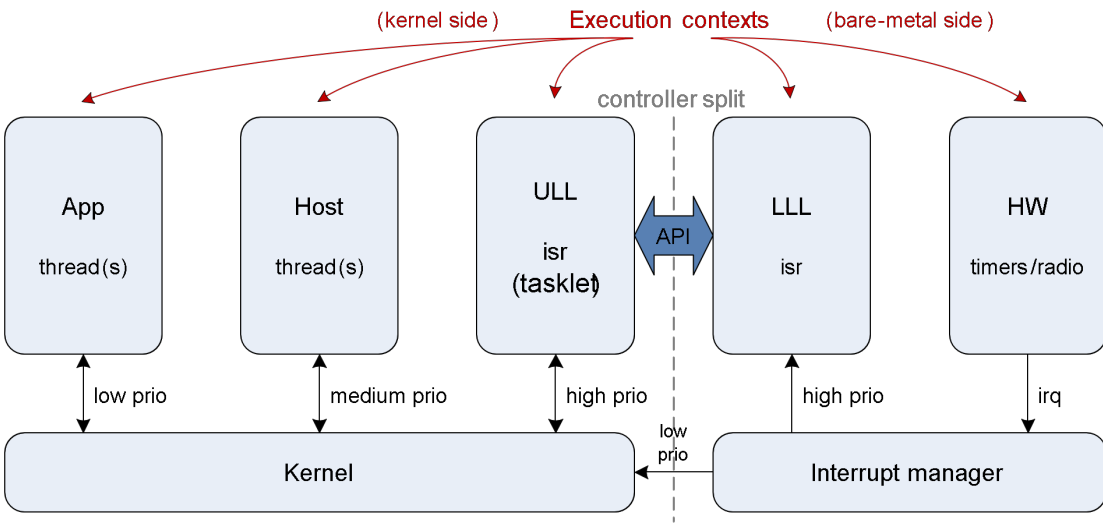
Overview

1. HCI
   - Host Controller Interface, Bluetooth standard
   - Provides Zephyr Bluetooth HCI Driver
2. HAL
   - Hardware Abstraction Layer
   - Vendor Specific, and Zephyr Driver usage
3. Ticker
   - Soft real time radio/resource scheduling
4. LL_SW

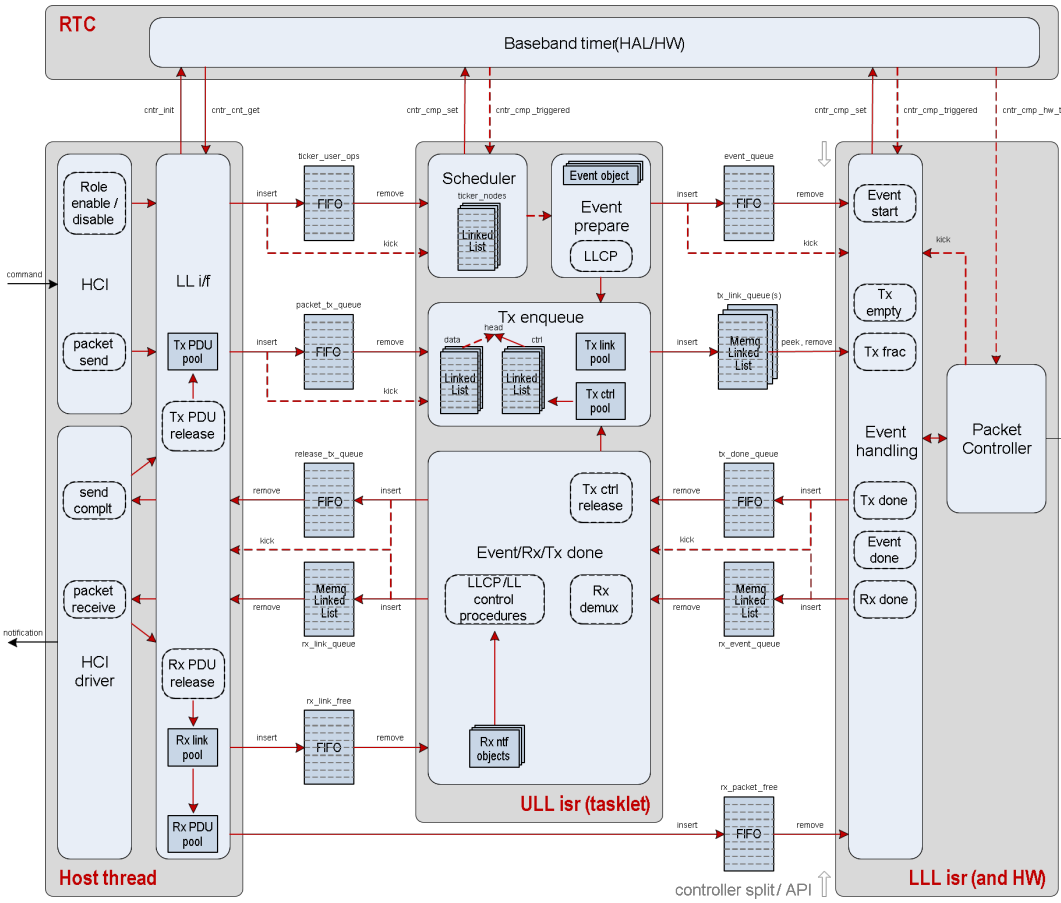
- Software-based Link Layer implementation
- States and roles, control procedures, packet controller

5. Util
   - Bare metal memory pool management
   - Queues of variable count, lockless usage
   - FIFO of fixed count, lockless usage
   - Mayfly concept based deferred ISR executions

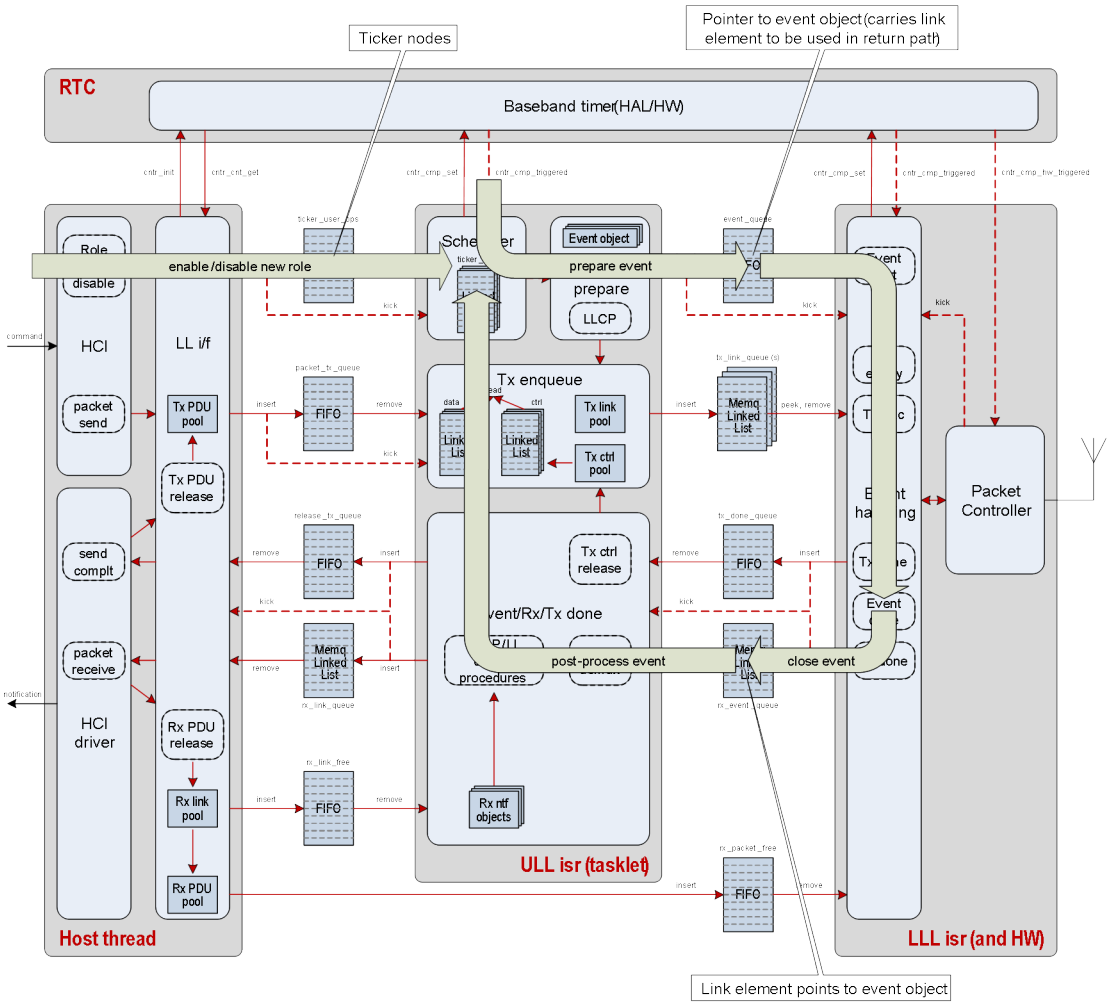
Architecture



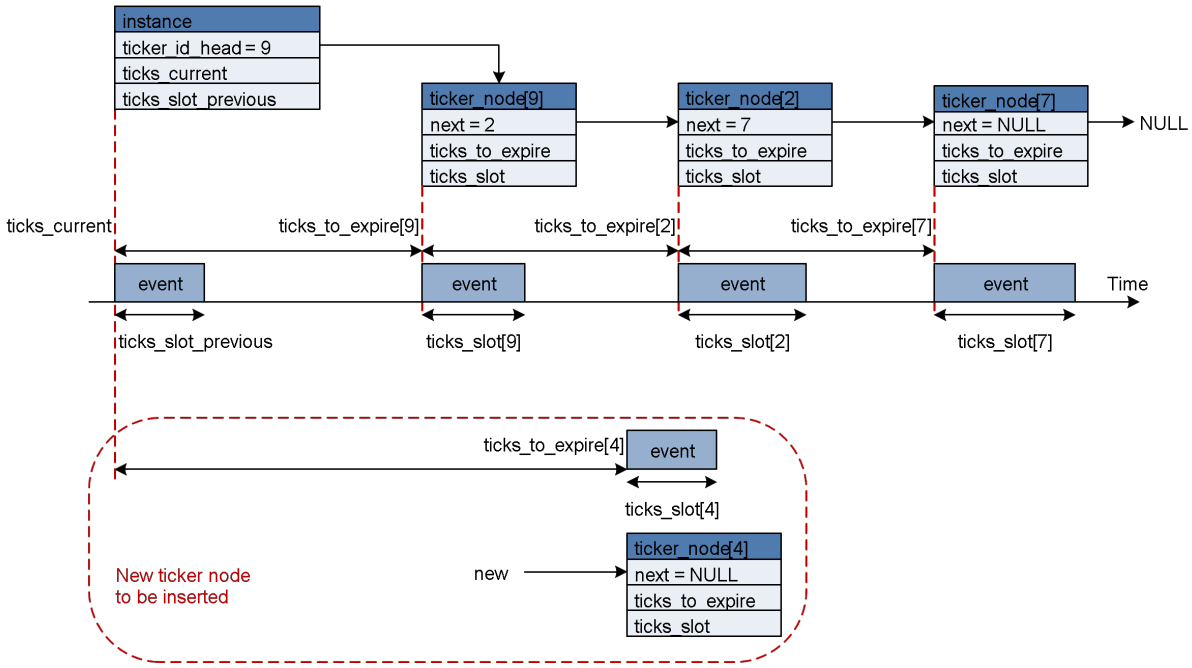
Execution Overview



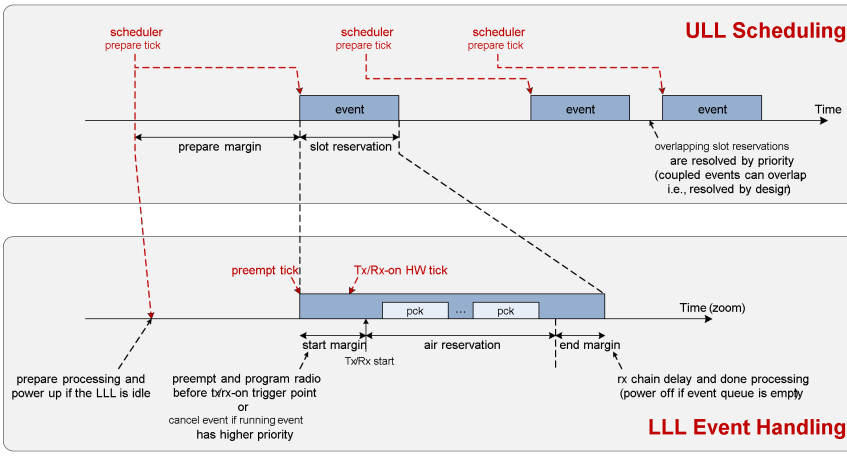
Architecture Overview



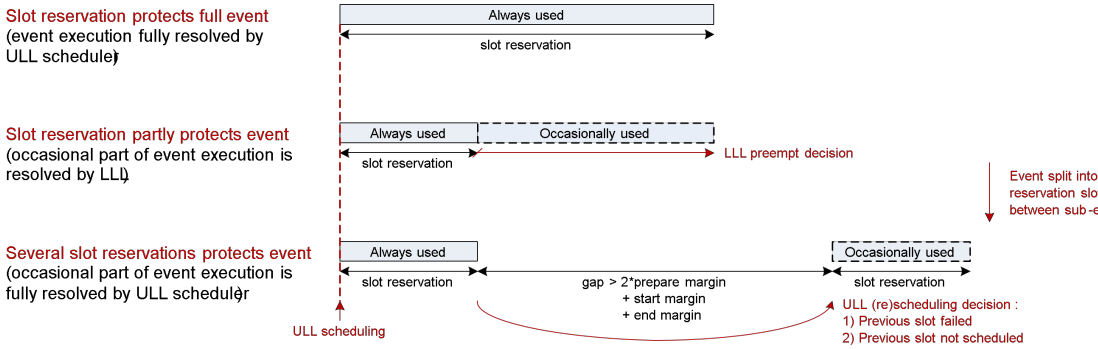
Scheduling



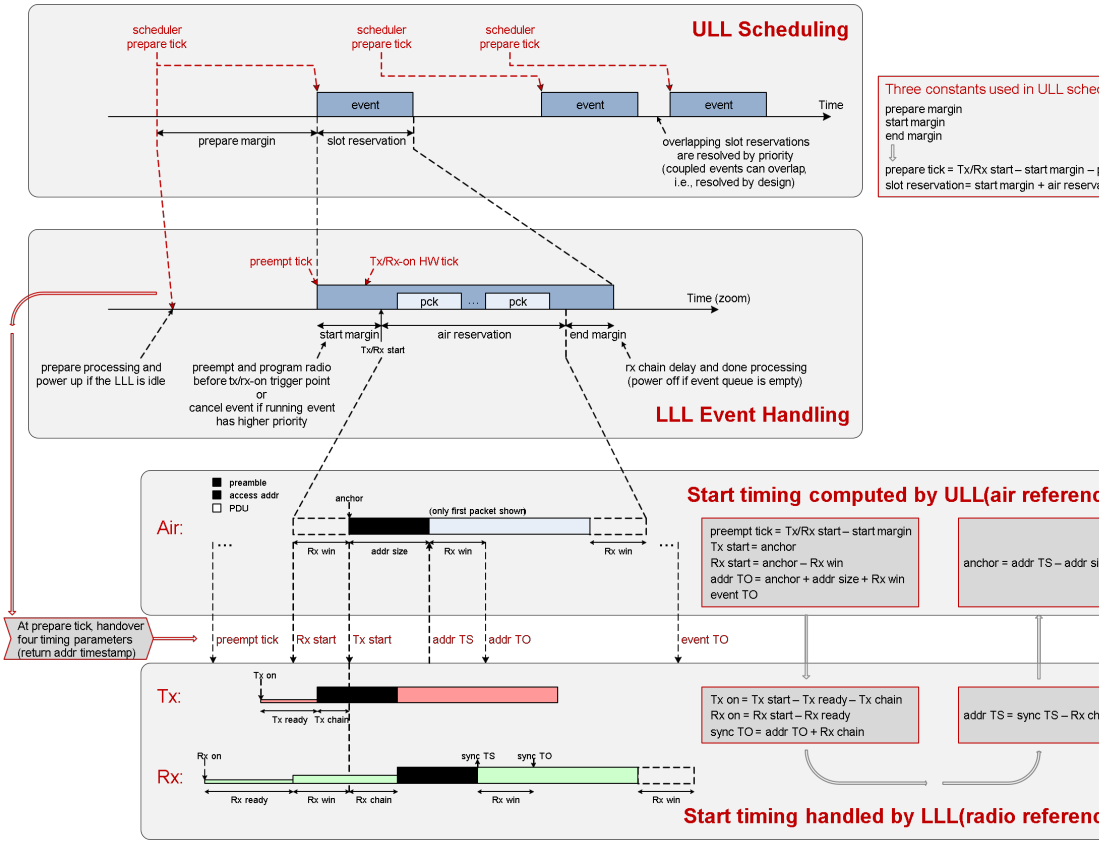
Ticker



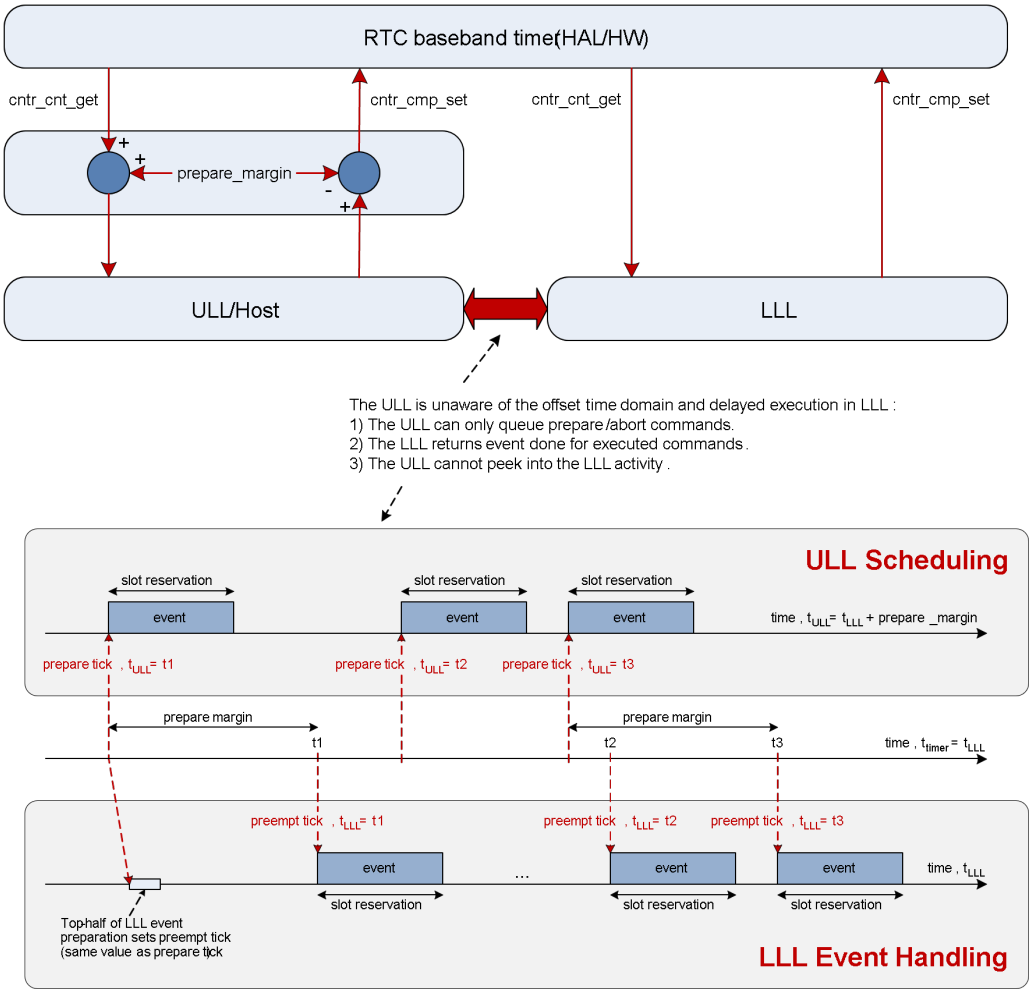
Upper Link Layer and Lower Link Layer



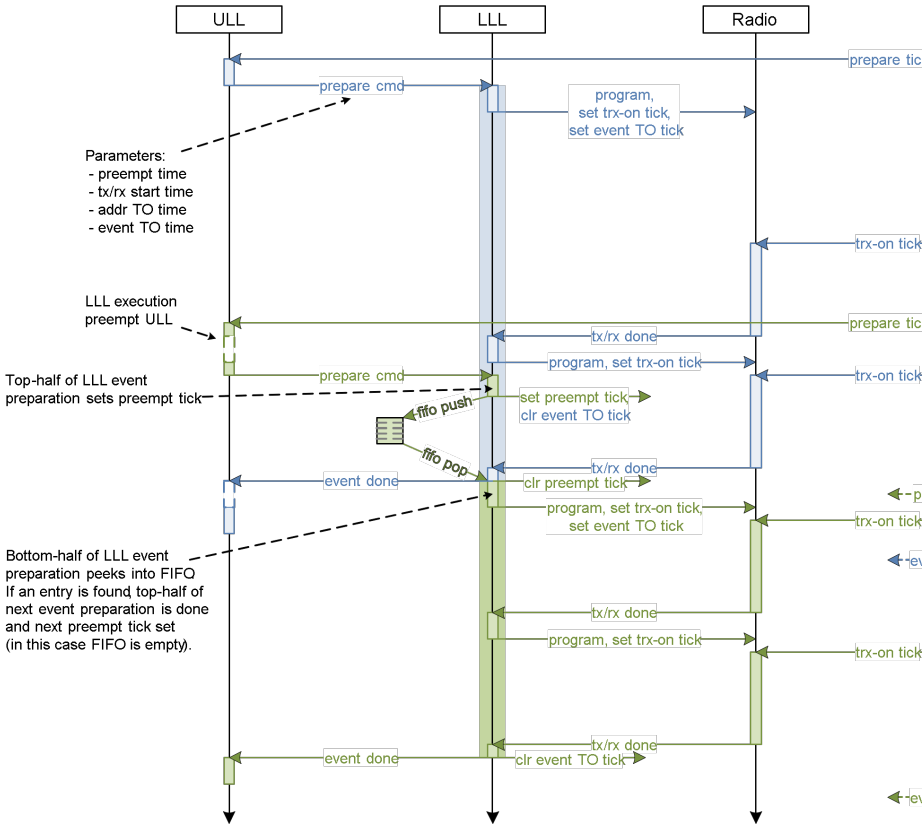
Scheduling Variants



ULL and LLL Timing

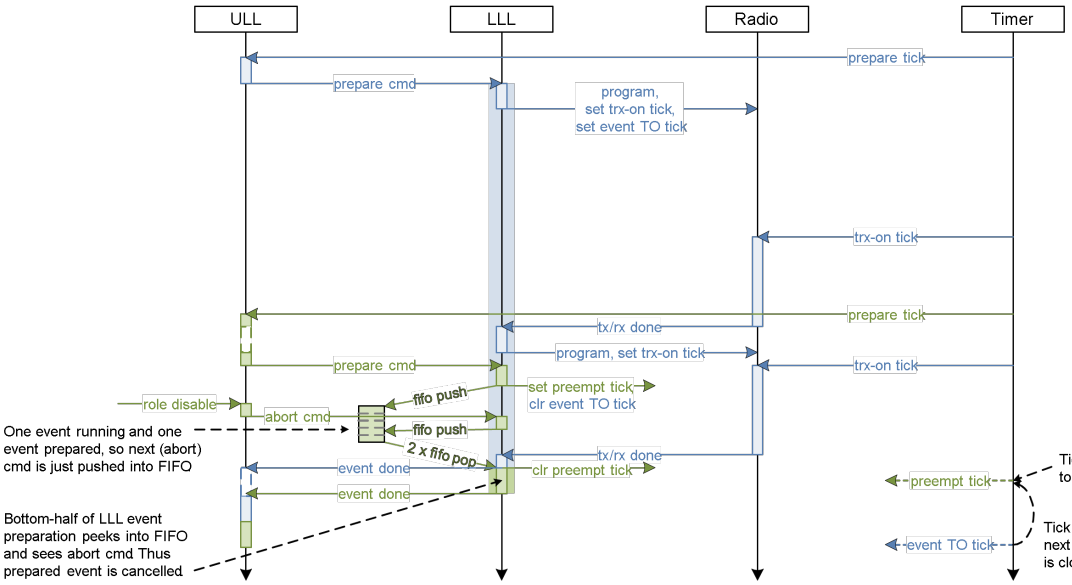


Event Handling

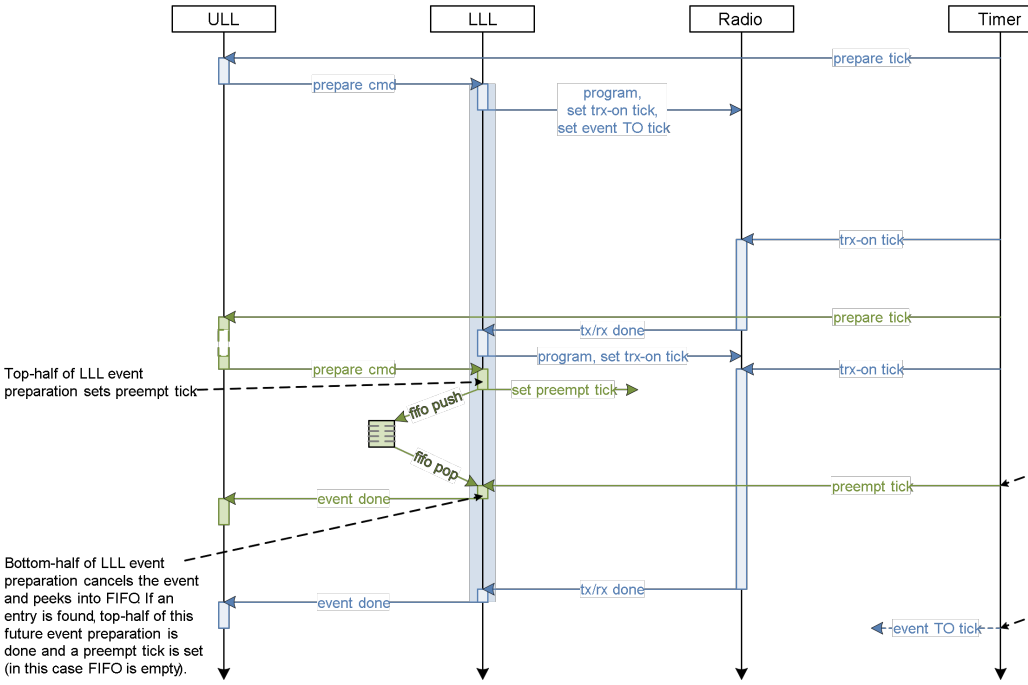


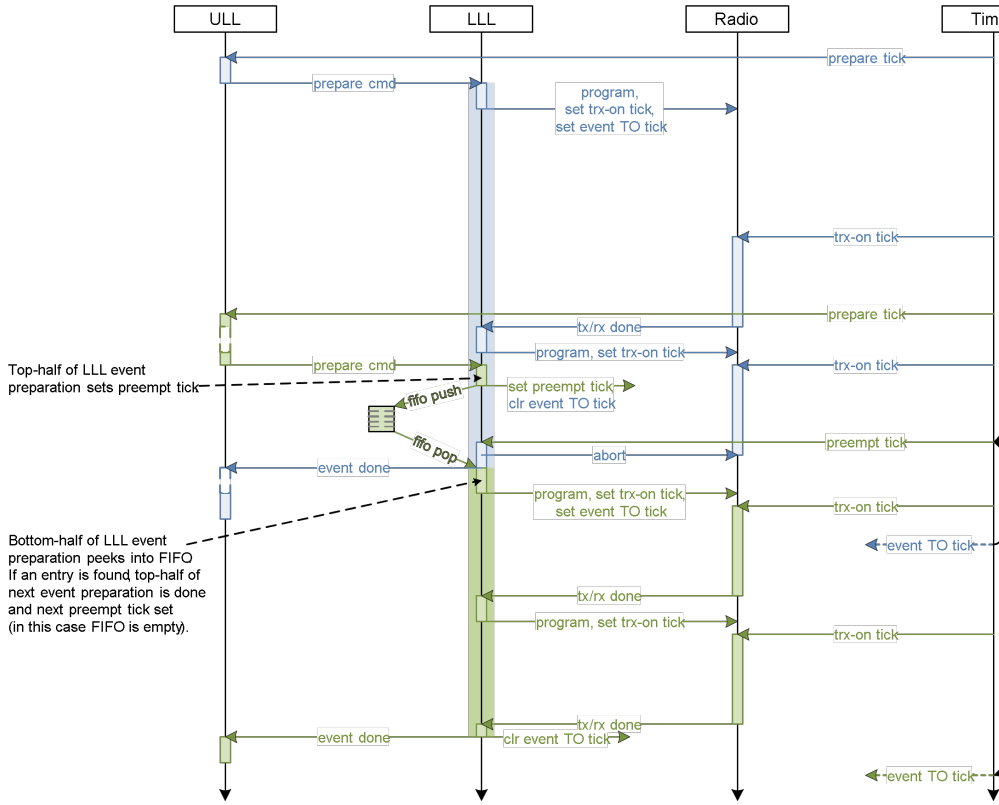
Scheduling Closely Spaced Events

Aborting Active Event



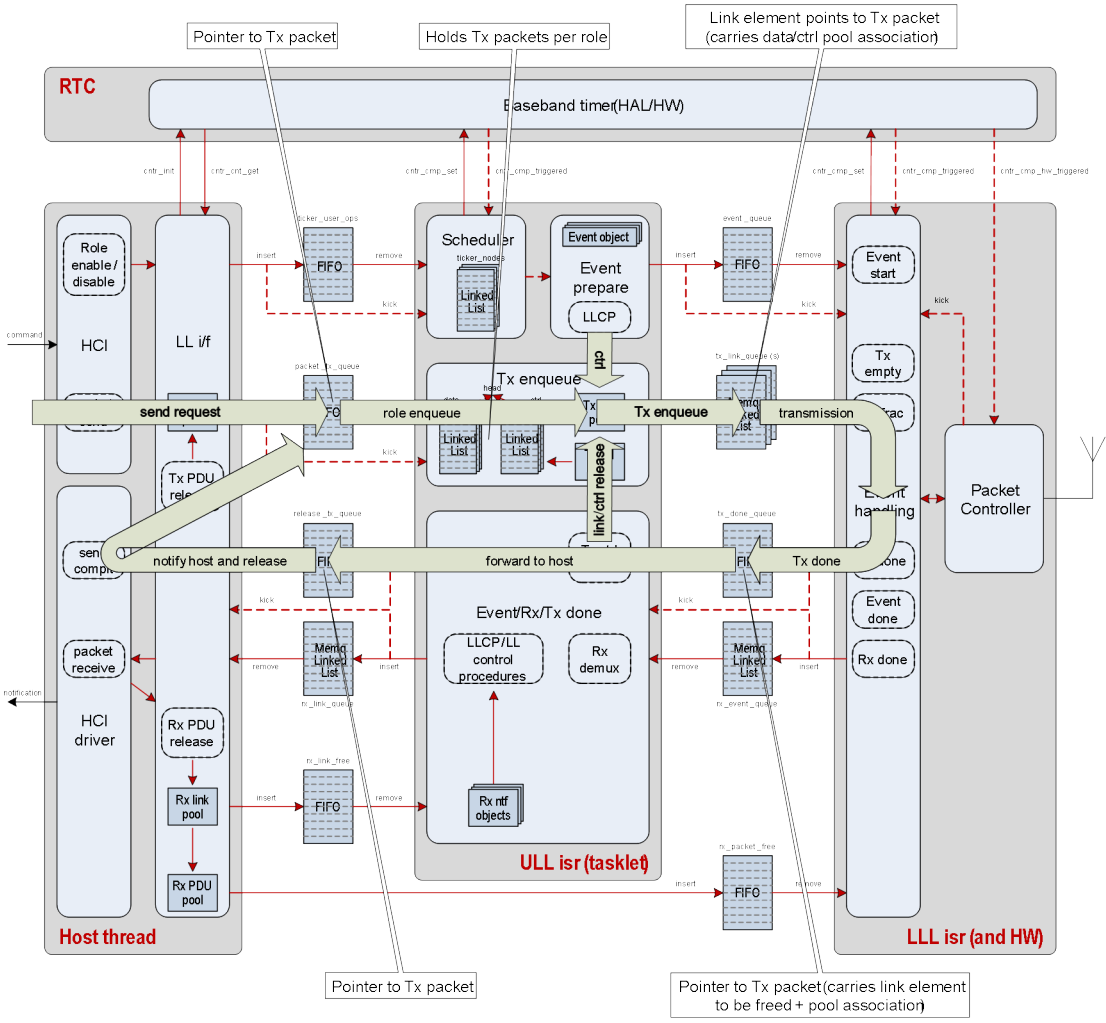
Cancelling Pending Event



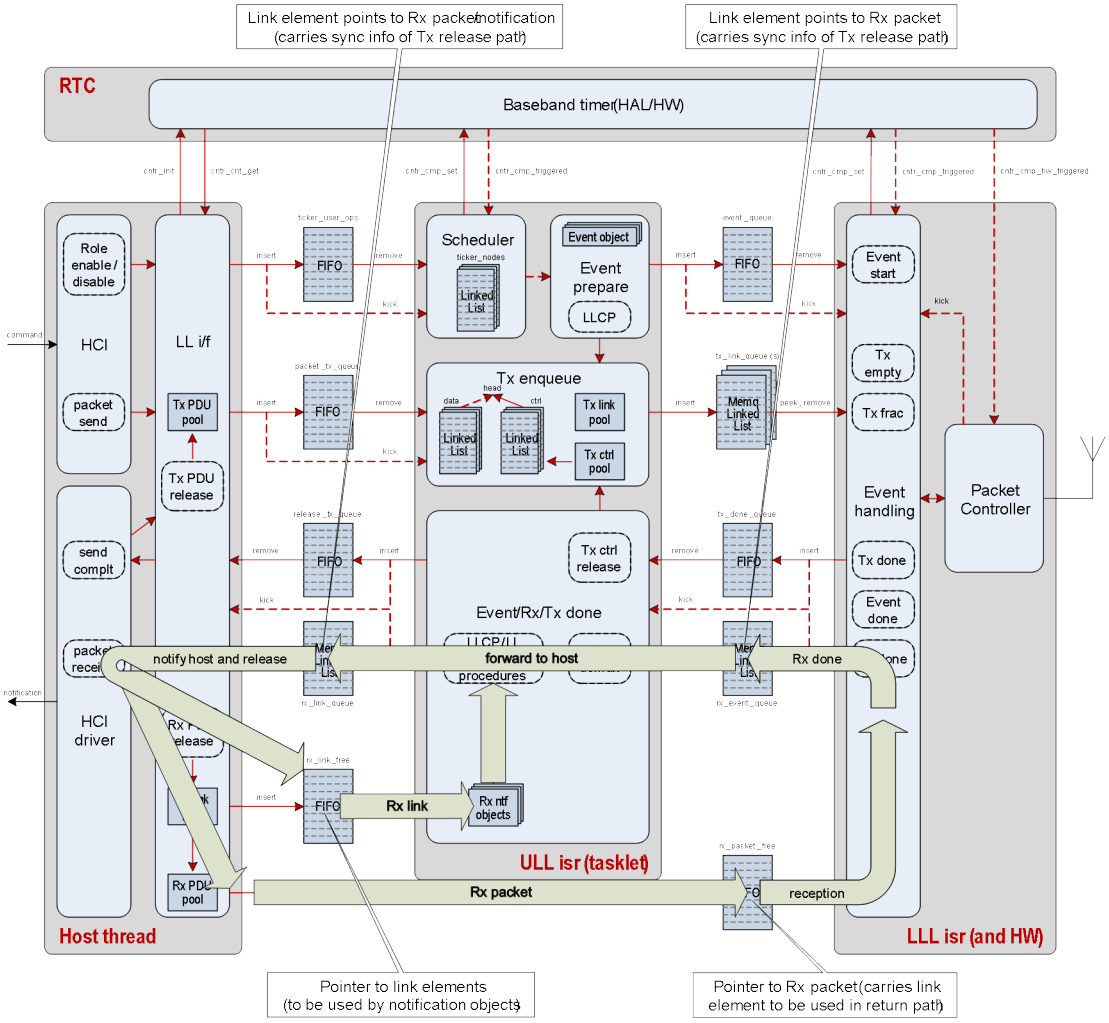


Pre-emption of Active Event

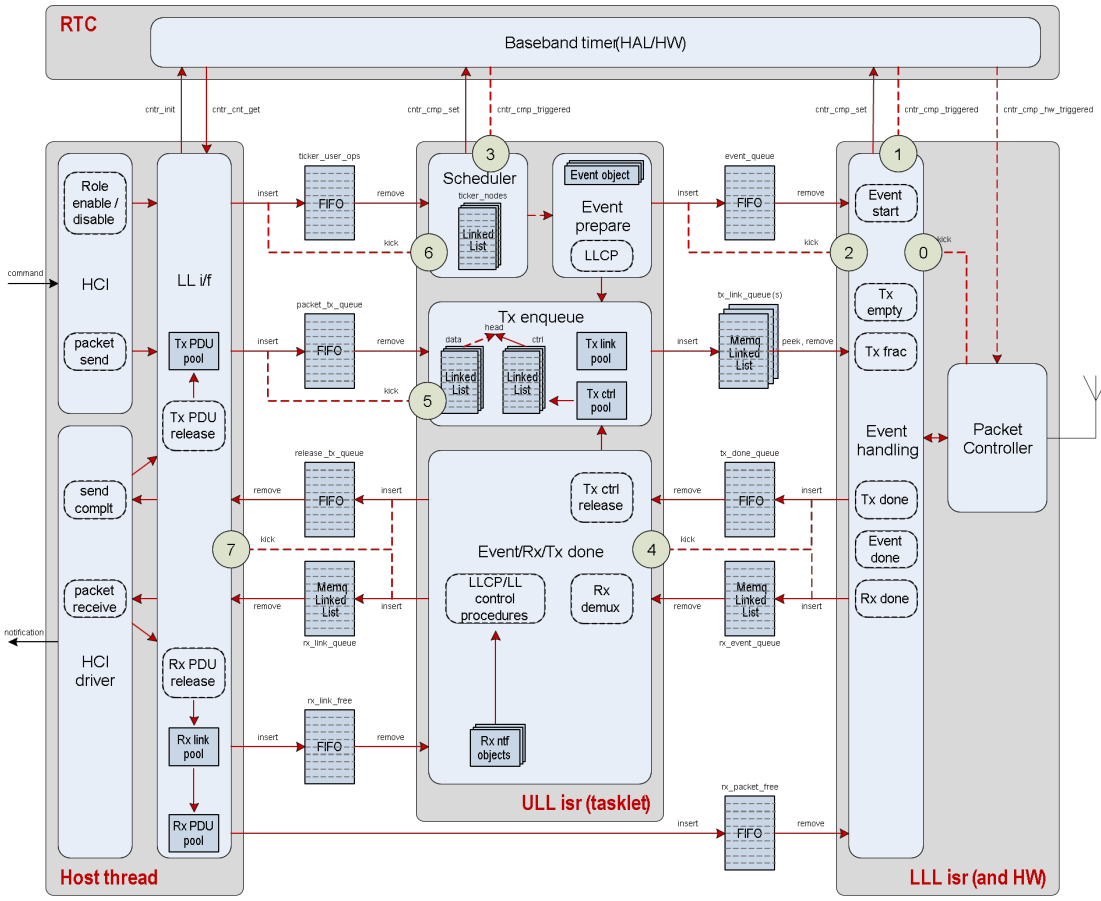
Data Flow



Transmit Data Flow



Receive Data Flow



Execution Priorities

- Event handle (0, 1) < Event preparation (2, 3) < Event/Rx done (4) < Tx request (5) < Role management (6) < Host (7).
- LLL is vendor ISR, ULL is Mayfly ISR concept, Host is kernel thread.

Link Layer Control Procedures Following is a brief fly in on the main concepts in the implementation of control procedures handling in ULL.

Three main execution contexts

- **HCI/LLCP API**
  - Miscellaneous procedure initiation API
  - From ull_llcp.c::ull_cp_<proc>() for initiating local procedures
  - Interface with running procedures, local and remote
- **lll_prepare context to drive ull_cp_run()**
  - LLCP main state machine entry/tick'er
  - From ull_peripheral.c/ull_central.c::ticker_cb via ull_conn_llcp()
- **rx_demux context to drive ull_cp_tx_ack() and ull_cp_rx()**
  - LLCP tx ack handling and PDU reception
  - From ull_conn.c::ull_conn_rx()
  - Handles passing PDUs into running procedures as well as possibly initiating remote procedures

Data structures and PDU helpers

- **struct llcp_struct**
  - Main LLCP data store
  - Defined in ull_conn_types.h and declared as part of struct ll_conn
  - Holds local and remote procedure request queues as well as conn specific LLCP data
  - Basic conn-level abstraction
- **struct proc_ctx**
  - General procedure context data, contains miscellaneous procedure data and state as well as sys_snode_t for queueing
  - Defined in ull_llcp_internal.h, declared/instantiated through ull_llcp.c::create_procedure()
  - Also holds node references used in tx_ack as well as rx_node retention mechanisms
- **struct llcp_mem_pool**
  - Mem pool used to implement procedure contexts resource - instantiated in both a local and a remote version
  - Used through ull_llcp.c::create_procedure()
- **Miscellaneous pdu gymnastics**
  - Encoding and decoding of control pdus done by ull_llcp_pdu.c::llcp_pdu_encode/decode_<PDU>()
  - Miscellaneous pdu validation handled by ull_llcp.c::pdu_validate_<PDU>() via ull_llcp.c::pdu_is_valid()

LLCP local and remote request/procedure state machines

- **ull_llcp_local.c**
  - State machine handling local initiated procedures
  - Naming concept: lr _<...> => local request machine
  - Local procedure queue handling
  - Local run/rx/tx_ack switch
- **ull_llcp_remote.c**
  - Remote versions of the above
  - Naming concept: rr_<...> => remote request machine
  - Also handling of remote procedure initiation by llcp_rx_new()
  - Miscellaneous procedure collision handling (in rr_st_idle())
- **ull_llcp_common/conn_upd/phy/enc/cc/chmu.c**
  - Individual procedure implementations (ull_llcp_common.c collects the simpler ones)
  - Naming concept: lp_<...> => local initiated procedure, rp_<...> => remote initiated procedure
  - Handling of procedure flow from init (possibly through instant) to completion and host notification if applicable

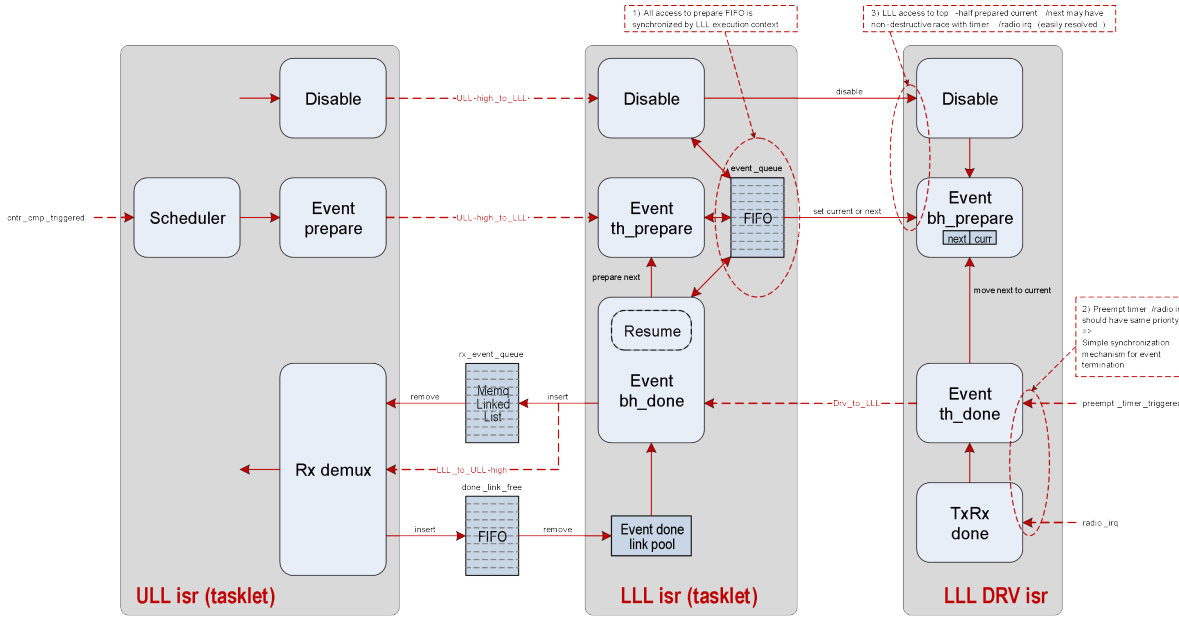
Miscellaneous concepts

- **Procedure collision handling**
  - See BT spec for explanation
  - Basically some procedures can exist in parallel but some can't - for instance only one instant based at a time
  - Spec states rules for how to handle/resolve collisions when they happen
- **Termination handling**
  - Specific rules apply re. how termination is handled.
  - Since we have resource handling re. procedure contexts and terminate must always be available this is handled as a special case
  - Note also - there are miscellaneous cases where connection termination is triggered on invalid peer behavior
- **New remote procedure handling**
  - Table new_proc_lut[] maps LLCP PDUs to procedures/roles used in llcp_rr_new()
  - Note - for any given connection, there can only ever be ONE remote procedure in the remote procedure queue
- **Miscellaneous minors**
  - pause/resume concepts - there are two (see spec for details)
  - procedure execution can be paused by the encryption procedure
  - data TX can be paused by PHY, DLE and ENC procedure
  - RX node retention - ensures no waiting for allocation of RX node when needed for notification

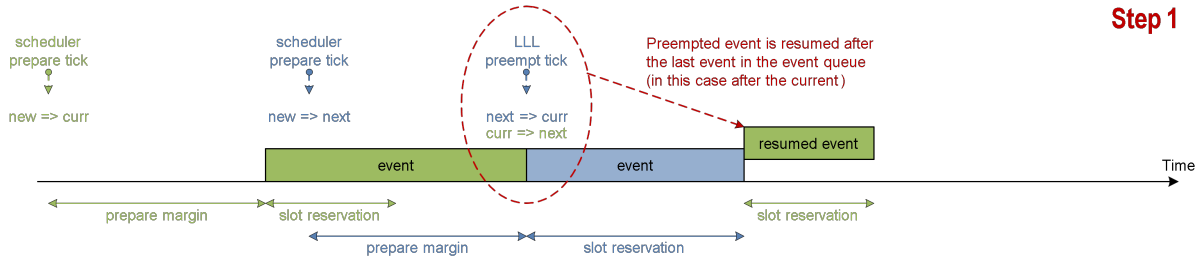
Miscellaneous unit test concepts

- **Individual ZTEST unit test for each procedure**
  - zephyr/tests/bluetooth/controller/ctrl_<proc>
- **Rx node handling is mocked**
  - Different configs are handled by separate conf files (see ctrl_conn_update for example)
  - ZTEST(periph_rem_no_param_req, test_conn_update_periph_rem_accept_no_param_req)
- **Emulated versions of rx_demux/prepare context used in unit tests - testing ONLY procedure PDU flow**
  - event_prepare()/event_done() helpers emulating LLL prepare/done flow
  - lt_rx()/lt_tx() 'lower tester' emulation of rx/tx
  - ut_rx_node() 'upper tester' emulation of notification flow handling
  - Bunch of helpers to generate and parse PDUs, as well as miscellaneous mocked ull_stuff()

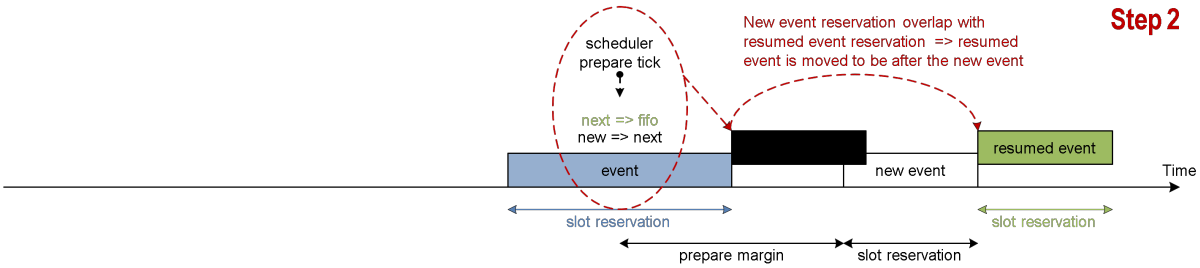
Lower Link Layer



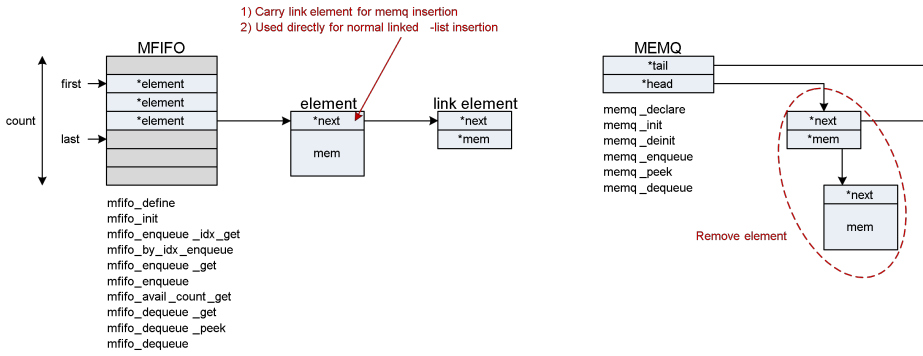
LLL Execution



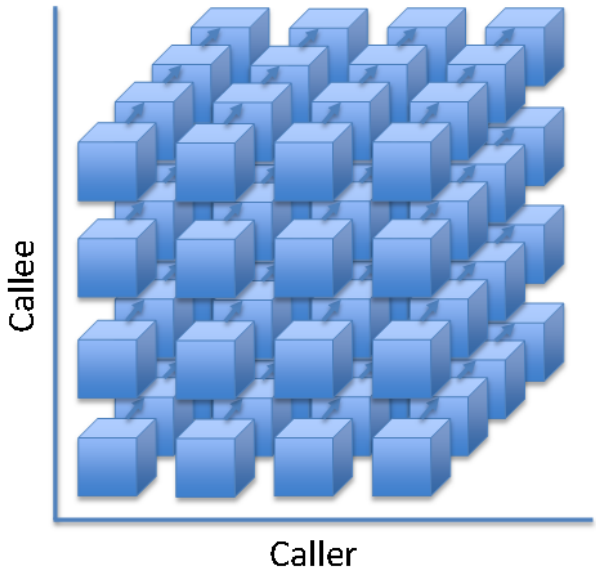
LLL Resume



Bare metal utilities

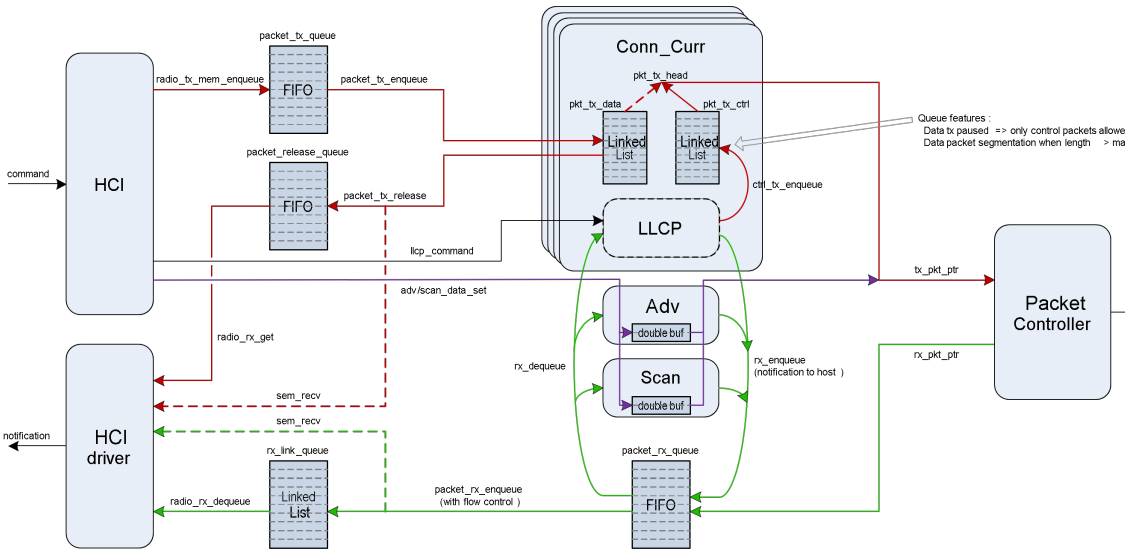


Memory FIFO and Memory Queue



Mayfly

- Mayfly are multi-instance scalable ISR execution contexts
- What a Work is to a Thread, Mayfly is to an ISR
- List of functions executing in ISRs
- Execution priorities map to IRQ priorities
- Facilitate cross execution context scheduling
- Race-to-idle execution
- Lock-less, bare metal



Legacy Controller

Bluetooth Low Energy Controller - Vendor Specific Details

Hardware Requirements

Nordic Semiconductor The Nordic Semiconductor Bluetooth Low Energy Controller implementation requires the following hardware peripherals.

Table 27: SoC Peripheral Use

Resource	nRF Peripheral	# instances	Zephyr Driver Accessible	Description
Clock	NRF_CLOCK	1	Yes	• A Low Frequency Clock (LFCLOCK) or sleep clock, for low power consumption between Bluetooth radio events • A High Frequency Clock (HFCLOCK) or active clock, for high precision packet timing and software based transceiver state switching with inter-frame space (tIFS) timing inside Bluetooth radio events
RTC [a]	NRF_RTC0	1	**No**	• Uses 2 capture/compare registers
Timer	NRF_TIMER0 or NRF_TIMER4 and NRF_TIMER1	2 or 1[Page 891, 1]	**No**	• 2 instances, one each for packet timing and tIFS software switching, respectively • 7 capture/compare registers (3 mandatory, 1 optional for ISR profiling, 4 for single timer tIFS switching) on first instance • 4 capture/compare registers for second instance, if single tIFS timer is not used.
PPI [b]	NRF_PPI	21 channels (20[2]), and 2 channel groups[3]	Yes[4]	• Used for radio mode switching to achieve tIFS timings, for PA/LNA control
DPPI [c]	NRF_DPPI	20 channels, and 2 channel groups[Page 89]	Yes[Page 8]	• Used for radio mode switching to achieve tIFS timings, for PA/LNA control
SWI [d]	NRF_SWI4 and NRF_SWI5, or NRF_SWI2 and NRF_SWI3[5]	2	**No**	• 2 instances, for Lower Link Layer and Upper Link Layer Low priority execution context
Radio	NRF_RADIO	1	**No**	• 2.4 GHz radio transceiver with multiple radio standards such as 1 Mbps, 2 Mbps and Coded PHY S2/S8 Long Range Bluetooth Low Energy technology
RNG [e]	NRF_RNG	1	Yes	
ECB [f]	NRF_ECB	1	**No**	
CBC-CCM [g]	NRF_CCM	1	**No**	
AAR [h]	NRF_AAR	1	**No**	
GPIO [i]	NRF_GPIO	2 GPIO pins for PA and LNA, 1 each	Yes	• Additionally, 10 Debug GPIO pins (optional)
GPIOTE [j]	NRF_GPIOTE	1	Yes	• Used for PA/LNA
TEMP [k]	NRF_TEMP	1	Yes	• For RC sourced LFCLOCK calibration
UART [l]	NRF_UART0	1	Yes	• For HCI interface in Controller only builds
IPC [m]	NRF_IPC[Page 8]	1	Yes	• For HCI interface in Controller only builds

Application Development

Bluetooth applications are developed using the common infrastructure and approach that is described in the *Application Development* section of the documentation.

Additional information that is only relevant to Bluetooth applications can be found on this page.

- *Thread safety*
- *Hardware setup*
  - *Embedded*
  - *Host on Linux with an external Controller*
  - *Simulated nRF5x with BabbleSim*
- *Initialization*
- *Bluetooth Application Example*
- *More Examples*

Thread safety Calling into the Bluetooth API is intended to be thread safe, unless otherwise noted in the documentation of the API function. The effort to ensure that this is the case for all API calls is an ongoing one, but the overall goal is formally stated in this paragraph. Bug reports and Pull Requests that move the subsystem in the direction of such goal are welcome.

Hardware setup This section describes the options you have when building and debugging Bluetooth applications with Zephyr. Depending on the hardware that is available to you, the requirements you have and the type of development you prefer you may pick one or another setup to match your needs.

There are 3 possible setups:

1. *Embedded*
2. *External controller*
   - *QEMU host*
   - *native_sim host*
3. *Simulated nRF5x with BabbleSim*

Embedded This setup relies on all software running directly on the embedded platform(s) that the application is targeting. All the *Configurations* and *Build Types* are supported but you might need to build Zephyr more than once if you are using a dual-chip configuration or if you have multiple cores in your SoC each running a different build type (e.g., one running the Host, the other the Controller).

To start developing using this setup follow the *Getting Started Guide*, choose one (or more if you are using a dual-chip solution) boards that support Bluetooth and then *run the application*).

There is a way to access the *HCI* traffic between the Host and Controller, even if there is no physical transport. See *Embedded HCI tracing* for instructions.

[1] `CONFIG_BT_CTLR_SW_SWITCH_SINGLE_TIMER =y`
[0] `CONFIG_BT_CTLR_TIFS_HW =n`
[2] When not using pre-defined PPI channels
[3] For software-based tIFS switching
[4] Drivers that use nRFx interfaces
[5] For nRF53x Series

> **Note**
>
> This is currently only available on GNU/Linux

Host on Linux with an external Controller This setup relies on a "dual-chip" *configuration* which is comprised of the following devices:

1. A *Host-only* application running in the *QEMU* emulator or the native_sim native port of Zephyr
2. A Controller, which can be one of the following types:
   - A commercially available Controller
   - A *Controller-only* build of Zephyr
   - A *Virtual controller*

> **Warning**
>
> Certain external Controllers are either unable to accept the Host to Controller flow control parameters that Zephyr sets by default (Qualcomm), or do not transmit any data from the Controller to the Host (Realtek). If you see a message similar to:
>
> ```
> <wrn> bt_hci_core: opcode 0x0c33 status 0x12
> ```
>
> when booting your sample of choice (make sure you have enabled `CONFIG_LOG` in your `prj.conf` before running the sample), or if there is no data flowing from the Controller to the Host, then you need to disable Host to Controller flow control. To do so, set `CONFIG_BT_HCI_ACL_FLOW_CONTROL=n` in your `prj.conf`.

QEMU You can run the Zephyr Host on the *QEMU emulator* and have it interact with a physical external Bluetooth Controller.

Refer to *Running on QEMU or native_sim* for full instructions on how to build and run an application in this setup.

> **Note**
>
> This is currently only available on GNU/Linux

native_sim The native_sim target builds your Zephyr application with the Zephyr kernel, and some minimal HW emulation as a native Linux executable.

This executable is a normal Linux program, which can be debugged and instrumented like any other, and it communicates with a physical or virtual external Controller. Refer to:

- *Running on QEMU or native_sim* for the physical controller
- *Running on a Virtual Controller and native_sim* for the virtual controller

> **Note**
>
> This is currently only available on GNU/Linux

Simulated nRF5x with BabbleSim The nrf52_bsim and nrf5340bsim boards, are simulated target boards which emulate the necessary peripherals of a nRF52/53 SOC to be able to develop and test Bluetooth LE applications. These boards, use:

- BabbleSim to simulate the nRF5x modem and the radio environment.
- The POSIX arch and native simulator to emulate the processor, and run natively on your host.
- Models of the nrf5x HW

Just like with the native_sim target, the build result is a normal Linux executable. You can find more information on how to run simulations with one or several devices in either of these boards's documentation.

With the nrf52_bsim, typically you do *Combined builds*, but it is also possible to build the controller with one of the bluetooth_hci_uart samples in one simulated device, and the host with the H4 driver instead of the integrated controller in another simulated device.

With the nrf5340bsim, you can build with either, both controller and host on its network core, or, with the network core running only the controller, the application core running the host and your application, and the HCI transport over IPC.

Initialization The Bluetooth subsystem is initialized using the `bt_enable()` function. The caller should ensure that function succeeds by checking the return code for errors. If a function pointer is passed to `bt_enable()`, the initialization happens asynchronously, and the completion is notified through the given function.

Bluetooth Application Example A simple Bluetooth beacon application is shown below. The application initializes the Bluetooth Subsystem and enables non-connectable advertising, effectively acting as a Bluetooth Low Energy broadcaster.

```

/*
 * Set Advertisement data. Based on the Eddystone specification:
 * https://github.com/google/eddystone/blob/master/protocol-specification.md
 * https://github.com/google/eddystone/tree/master/eddystone-url
 */
static const struct bt_data ad[] = {
        BT_DATA_BYTES(BT_DATA_FLAGS, BT_LE_AD_NO_BREDR),
        BT_DATA_BYTES(BT_DATA_UUID16_ALL, 0xaa, 0xfe),
        BT_DATA_BYTES(BT_DATA_SVC_DATA16,
                      0xaa, 0xfe, /* Eddystone UUID */
                      0x10, /* Eddystone-URL frame type */
                      0x00, /* Calibrated Tx power at 0m */
                      0x00, /* URL Scheme Prefix http://www. */
                      'z', 'e', 'p', 'h', 'y', 'r',
                      'p', 'r', 'o', 'j', 'e', 'c', 't',
                      0x08) /* .org */
};

/* Set Scan Response data */
static const struct bt_data sd[] = {
        BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
};
```

(continues on next page)

(continued from previous page)

```

static void bt_ready(int err)
{
        bt_addr_le_t addr = {0};
        size_t count = 1;

        if (err) {
                printk("Bluetooth init failed (err %d)\n", err);
                return;
        }

        printk("Bluetooth initialized\n");

        /* Start advertising */
        err = bt_le_adv_start(BT_LE_ADV_NCONN_IDENTITY, ad, ARRAY_SIZE(ad),
                              sd, ARRAY_SIZE(sd));
        if (err) {
                printk("Advertising failed to start (err %d)\n", err);
                return;
        }


        /* For connectable advertising you would use
         * bt_le_oob_get_local().  For non-connectable non-identity
         * advertising an non-resolvable private address is used;
         * there is no API to retrieve that.
         */

        bt_id_get(&addr, &count);

        printk("Beacon started, advertising as %s\n", bt_addr_le_str(&addr));
}

int main(void)
{
        int err;

        printk("Starting Beacon Demo\n");

        /* Initialize the Bluetooth Subsystem */
        err = bt_enable(bt_ready);
        if (err) {
                printk("Bluetooth init failed (err %d)\n", err);
        }
        return 0;
}
```

The key APIs employed by the beacon sample are `bt_enable()` that's used to initialize Bluetooth and then `bt_le_adv_start()` that's used to start advertising a specific combination of advertising and scan response data.

More Examples More sample Bluetooth applications are available in `samples/bluetooth/`.

API

Bluetooth Classic Host and profiles

Service Discovery Protocol (SDP)

API Reference

Service Discovery Protocol (SDP)

Bluetooth Logical Link Control and Adaptation Protocol (L2CAP) for BR/EDR L2CAP BR/EDR provides support for Bluetooth Classic L2CAP (Logical Link Control and Adaptation Protocol) features including ECHO request/response and connectionless data channels.

ECHO Request/Response The L2CAP ECHO feature allows testing the connection by sending ECHO requests and receiving ECHO responses. Applications can register callbacks to monitor ECHO packets and send ECHO data. The feature is enabled through the configuration option: `CONFIG_BT_CLASSIC`.

Registering ECHO Callbacks To monitor ECHO request/response packets, register a `bt_l2cap_br_echo_cb` callback structure:

```
static void echo_req_cb(struct bt_conn *conn, uint8_t identifier, struct net_buf␣
↪*buf)
{
    /* Handle ECHO request */
}

static void echo_rsp_cb(struct bt_conn *conn, struct net_buf *buf)
{
    /* Handle ECHO response */
}

static struct bt_l2cap_br_echo_cb echo_cb = {
    .req = echo_req_cb,
    .rsp = echo_rsp_cb,
};

bt_l2cap_br_echo_cb_register(&echo_cb);
```

Sending ECHO Request To send an ECHO request, allocate a buffer with `BT_L2CAP_BR_ECHO_REQ_RESERVE` bytes reserved for the L2CAP header:

```
struct net_buf *buf;

buf = net_buf_alloc(&pool, K_FOREVER);
net_buf_reserve(buf, BT_L2CAP_BR_ECHO_REQ_RESERVE);
net_buf_add_mem(buf, data, data_len);

bt_l2cap_br_echo_req(conn, buf);
```

Sending ECHO Response To send an ECHO response (typically in response to a received ECHO request), allocate a buffer with `BT_L2CAP_BR_ECHO_RSP_RESERVE` bytes reserved for the L2CAP header:

```
struct net_buf *buf;

buf = net_buf_alloc(&pool, K_FOREVER);
net_buf_reserve(buf, BT_L2CAP_BR_ECHO_RSP_RESERVE);
```

(continues on next page)

(continued from previous page)

```
net_buf_add_mem(buf, data, data_len);

bt_l2cap_br_echo_rsp(conn, buf);
```

The identifier parameter must match the identifier from the received ECHO request to properly correlate the response with the request.

Connectionless Data Channel The connectionless data channel allows sending and receiving data to/from a specific PSM (Protocol/Service Multiplexer) without establishing a connection-oriented L2CAP channel. The feature is enabled through the configuration option: `CONFIG_BT_L2CAP_CONNLESS`.

Registering Connectionless Callbacks To receive connectionless data, register a `bt_l2cap_br_connless_cb` callback structure:

```
static void connless_recv_cb(struct bt_conn *conn, uint16_t psm, struct net_buf␣
↪*buf)
{
    /* Handle received connectionless data */
}

static struct bt_l2cap_br_connless_cb connless_cb = {
    .psm = MY_PSM,  /* Or 0 to receive all */
    .sec_level = BT_SECURITY_L1,
    .recv = connless_recv_cb,
};

bt_l2cap_br_connless_register(&connless_cb);
```

Sending Connectionless Data To send connectionless data, allocate a buffer with `BT_L2CAP_CONNLESS_RESERVE` bytes reserved:

```
struct net_buf *buf;

buf = net_buf_alloc(&pool, K_FOREVER);
net_buf_reserve(buf, BT_L2CAP_CONNLESS_RESERVE);
net_buf_add_mem(buf, data, data_len);

bt_l2cap_br_connless_send(conn, psm, buf);
```

API Reference

Bluetooth Logical Link Control and Adaptation Protocol (L2CAP) for BR/EDR

Serial Port Emulation (RFCOMM)

API Reference

RFCOMM

Hands Free Profile (HFP)

API Reference

Hands Free Profile (HFP) Unit
Hands Free Profile (HFP)

Related code samples

- Hands-freeUse the Hands-Free Profile (HFP) APIs.

Hands Free Profile - Audio Gateway (HFP-AG)
Hands Free Profile - Audio Gateway (HFP-AG)

Related code samples

- Hands-free Audio Gateway (AG)Use the Hands-Free Profile Audio Gateway (AG) APIs.

Advanced Audio Distribution Profile (A2DP)

API Reference
Advanced Audio Distribution Profile (A2DP)

Related code samples

- A2DP SinkUse A2DP (Advanced Audio Distribution Profile) sink functionality.
- A2DP SourceUse A2DP (Advanced Audio Distribution Profile) source functionality.

Audio Video Remote Control Profile (AVRCP)

API Reference
Audio Video Remote Control Profile (AVRCP)

Generic Object Exchange Profile (GOEP)

API Reference

Generic Object Exchange Profile (GOEP)
Generic Object Exchange Profile (GOEP)

IrDA Object Exchange Protocol (OBEX)
IrDA Object Exchange Protocol (OBEX)

Basic Imaging Profile (BIP)

API Reference
Basic Imaging Profile (BIP)

Bluetooth LE Audio

Bluetooth Audio

API Reference
Bluetooth Audio

Related code samples

- Basic Audio Profile (BAP) Broadcast Audio AssistantUse BAP Broadcast Assistant functionality.
- Basic Audio Profile (BAP) Broadcast Audio SinkUse BAP Broadcast Sink functionality.
- Basic Audio Profile (BAP) Broadcast Audio SourceUse BAP Broadcast Source functionality.
- Basic Audio Profile (BAP) Unicast Audio ClientUse BAP Unicast Client functionality.
- Basic Audio Profile (BAP) Unicast Audio ServerUse BAP Unicast Server functionality.
- Common Audio Profile (CAP) AcceptorAdvertise audio availability to CAP Initiators using the CAP Acceptor role.
- Public Broadcast Profile (PBP) Public Broadcast SinkUse PBP Public Broadcast Sink functionality.
- Public Broadcast Profile (PBP) Public Broadcast SourceUse PBP Public Broadcast Source functionality.
- Telephone and Media Audio Profile (TMAP) Broadcast Media Receiver (BMR)Implement the TMAP Broadcast Media Receiver (BMR) role.
- Telephone and Media Audio Profile (TMAP) Broadcast Media Sender (BMS)Implement the LE Audio TMAP Broadcast Media Sender (BMS) role.
- Telephone and Media Audio Profile (TMAP) CentralImplement the TMAP Call Gateway (CG) and Unicast Media Sender (UMS) roles.
- Telephone and Media Audio Profile (TMAP) PeripheralImplement the TMAP Call Terminal (CT) and Unicast Media Receiver (UMR) roles.

Codec config parsing APIs

Basic Audio Profile

API Reference
Bluetooth Basic Audio Profile

Related code samples

- Basic Audio Profile (BAP) Broadcast Audio AssistantUse BAP Broadcast Assistant functionality.
- Basic Audio Profile (BAP) Broadcast Audio SinkUse BAP Broadcast Sink functionality.
- Basic Audio Profile (BAP) Broadcast Audio SourceUse BAP Broadcast Source functionality.
- Basic Audio Profile (BAP) Unicast Audio ClientUse BAP Unicast Client functionality.
- Basic Audio Profile (BAP) Unicast Audio ServerUse BAP Unicast Server functionality.
- Common Audio Profile (CAP) AcceptorAdvertise audio availability to CAP Initiators using the CAP Acceptor role.
- Common Audio Profile (CAP) HandoverConnect to a CAP acceptor and perform CAP handover procedures

- Common Audio Profile (CAP) InitiatorConnect to CAP Acceptors and setup unicast audio streaming or broadcast audio streams.
- Public Broadcast Profile (PBP) Public Broadcast SinkUse PBP Public Broadcast Sink functionality.
- Public Broadcast Profile (PBP) Public Broadcast SourceUse PBP Public Broadcast Source functionality.
- Telephone and Media Audio Profile (TMAP) Broadcast Media Receiver (BMR)Implement the TMAP Broadcast Media Receiver (BMR) role.
- Telephone and Media Audio Profile (TMAP) Broadcast Media Sender (BMS)Implement the LE Audio TMAP Broadcast Media Sender (BMS) role.
- Telephone and Media Audio Profile (TMAP) CentralImplement the TMAP Call Gateway (CG) and Unicast Media Sender (UMS) roles.
- Telephone and Media Audio Profile (TMAP) PeripheralImplement the TMAP Call Terminal (CT) and Unicast Media Receiver (UMR) roles.

BAP Unicast Client APIs

BAP Unicast Server APIs

BAP Broadcast APIs

BAP Broadcast Sink APIs

BAP Broadcast Source APIs

Common Audio Profile

API Reference
Common Audio Profile (CAP)

Related code samples

- Common Audio Profile (CAP) AcceptorAdvertise audio availability to CAP Initiators using the CAP Acceptor role.
- Common Audio Profile (CAP) HandoverConnect to a CAP acceptor and perform CAP handover procedures
- Common Audio Profile (CAP) InitiatorConnect to CAP Acceptors and setup unicast audio streaming or broadcast audio streams.
- Telephone and Media Audio Profile (TMAP) Broadcast Media Sender (BMS)Implement the LE Audio TMAP Broadcast Media Sender (BMS) role.
- Telephone and Media Audio Profile (TMAP) CentralImplement the TMAP Call Gateway (CG) and Unicast Media Sender (UMS) roles.

Bluetooth Coordinated Sets

API Reference
Coordinated Set Identification Profile (CSIP)

Related code samples

- Telephone and Media Audio Profile (TMAP) PeripheralImplement the TMAP Call Terminal (CT) and Unicast Media Receiver (UMR) roles.

Bluetooth Media

API Reference

Media Control Service
Media Control Service (MCS)

Media Proxy
Media Proxy

Media Control Client
Media Control Client (MCC)

Related code samples

- Telephone and Media Audio Profile (TMAP) PeripheralImplement the TMAP Call Terminal (CT) and Unicast Media Receiver (UMR) roles.

Bluetooth Microphone Control

API Reference
Microphone Control Profile (MICP)

Bluetooth Audio Volume Control

API Reference
Volume Control Profile (VCP)

Related code samples

- Telephone and Media Audio Profile (TMAP) Broadcast Media Receiver (BMR)Implement the TMAP Broadcast Media Receiver (BMR) role.
- Telephone and Media Audio Profile (TMAP) CentralImplement the TMAP Call Gateway (CG) and Unicast Media Sender (UMS) roles.
- Telephone and Media Audio Profile (TMAP) PeripheralImplement the TMAP Call Terminal (CT) and Unicast Media Receiver (UMR) roles.

Bluetooth LE Host

Bluetooth standard services

Battery Service
Battery Service (BAS)

Related code samples

- ESP PeripheralExpose environmental information using the Environmental Sensing Profile (ESP).
- Health Thermometer (Peripheral)Expose a Health Thermometer (HT) GATT Service generating dummy temperature values.
- Heart-rate Monitor (Peripheral)Expose a Heart Rate (HR) GATT Service generating dummy heart-rate values.

Current Time Service
Current Time Service (CTS)

Elapsed Time Service
Elapsed Time Service (ETS)

Related code samples

- Elapsed Time Service (ETS) PeripheralExpose an Elapsed Time Service (ETS) GATT Service.

Heart Rate Service
Heart Rate Service (HRS)

Related code samples

- HCI Power ControlDynamically control the Tx power of a Bluetooth LE Controller using HCI vendor-specific commands.
- Heart-rate Monitor (Peripheral)Expose a Heart Rate (HR) GATT Service generating dummy heart-rate values.

Immediate Alert Service
Immediate Alert Service (IAS)

Object Transfer Service
Object Transfer Service (OTS)

Related code samples

- Peripheral Object Transfer Service (OTS)Expose an Object Transfer Service (OTS) GATT Service.

Generic Access Profile (GAP)

API Reference
Generic Access Profile (GAP)

Related code samples

- Bluetooth Low Energy Power Management on STM32WBPerform Bluetooth LE operations with Zephyr power management enabled on STM32WB.
- Central Periodic Advertising Sync Transfer (PAST)Use the Periodic Advertising Sync Transfer (PAST) feature as the sender.
- Channel SoundingUse Channel Sounding functionality.
- Direction Finding Periodic Advertising BeaconImplement Bluetooth LE Direction Finding CTE Broadcaster functionality.
- Direction Finding Periodic Advertising LocatorImplement Bluetooth LE Direction Finding CTE Locator functionality.
- Nordic flash write during Bluetooth LE activityWrite to flash while the Bluetooth LE radio is active.
- ObserverScan for Bluetooth devices nearby and print their information.
- Periodic AdvertisingUse Bluetooth LE Periodic Advertising functionality.
- Periodic Advertising Connection Procedure (Initiator)Initiate a connection to a device using the Periodic Advertising Connection Procedure.
- Periodic Advertising Connection Procedure (Responder)Respond to periodic advertising and establish a connection.
- Periodic Advertising SynchronizationUse Bluetooth LE Periodic Advertising Synchronization functionality.
- Periodic Advertising Synchronization Transfer PeripheralUse the Periodic Advertising Sync Transfer (PAST) feature as the receiver.
- Periodic Advertising with Responses (PAwR) AdvertiserUse Bluetooth LE Periodic Advertising with Responses (PAwR) Advertiser functionality.
- Periodic Advertising with Responses (PAwR) SynchronizationImplement Bluetooth LE Periodic Advertising with Responses Synchronization.
- Scan & AdvertiseCombine Bluetooth LE Broadcaster & Observer roles to advertise and scan for devices simultaneously.

Device Address

Defines and Assigned Numbers

Generic Attribute Profile (GATT) The GATT layer manages the service database providing APIs for service registration and attribute declaration.

The GATT Client initiates commands and requests towards the GATT Server, and can receive responses, indications and notifications sent by the server. It is enabled through the configuration option: `CONFIG_BT_GATT_CLIENT`

The GATT Server accepts incoming commands and requests from the GATT Client, and sends responses, indications and notifications to the client.

Services can be registered using the `bt_gatt_service_register()` API which takes the `bt_gatt_service` struct that provides the list of attributes the service contains. The helper macro `BT_GATT_SERVICE()` can be used to declare a service.

Attributes can be declared using the `bt_gatt_attr` struct or using one of the helper macros:

`BT_GATT_PRIMARY_SERVICE`
: Declares a Primary Service.

`BT_GATT_SECONDARY_SERVICE`
Declares a Secondary Service.

`BT_GATT_INCLUDE_SERVICE`
Declares a Include Service.

`BT_GATT_CHARACTERISTIC`
Declares a Characteristic.

`BT_GATT_DESCRIPTOR`
Declares a Descriptor.

`BT_GATT_ATTRIBUTE`
Declares an Attribute.

`BT_GATT_CCC`
Declares a Client Characteristic Configuration.

`BT_GATT_CEP`
Declares a Characteristic Extended Properties.

`BT_GATT_CUD`
Declares a Characteristic User Format.

Each attribute contain a `uuid`, which describes their type, a `read` callback, a `write` callback and a set of permission. Both read and write callbacks can be set to NULL if the attribute permission don't allow their respective operations.

> **Note**
>
> 32-bit UUIDs are not supported in GATT. All 32-bit UUIDs shall be converted to 128-bit UUIDs when the UUID is contained in an ATT PDU.

> **Note**
>
> Attribute `read` and `write` callbacks are called directly from RX Thread thus it is not recommended to block for long periods of time in them.

Attribute value changes can be notified using `bt_gatt_notify()` API, alternatively there is `bt_gatt_notify_cb()` where it is possible to pass a callback to be called when it is necessary to know the exact instant when the data has been transmitted over the air. Indications are supported by `bt_gatt_indicate()` API.

Discover procedures can be initiated with the use of `bt_gatt_discover()` API which takes the `bt_gatt_discover_params` struct which describes the type of discovery. The parameters also serves as a filter when setting the `uuid` field only attributes which matches will be discovered, in contrast setting it to NULL allows all attributes to be discovered.

> **Note**
>
> Caching discovered attributes is not supported.

Read procedures are supported by `bt_gatt_read()` API which takes the `bt_gatt_read_params` struct as parameters. In the parameters one or more attributes can be set, though setting multiple handles requires the option: `CONFIG_BT_GATT_READ_MULTIPLE`

Write procedures are supported by `bt_gatt_write()` API and takes `bt_gatt_write_params` struct as parameters. In case the write operation don't require a response

`bt_gatt_write_without_response()` or `bt_gatt_write_without_response_cb()` APIs can be used, with the later working similarly to `bt_gatt_notify_cb()`.

Subscriptions to notification and indication can be initiated with use of `bt_gatt_subscribe()` API which takes `bt_gatt_subscribe_params` as parameters. Multiple subscriptions to the same attribute are supported so there could be multiple `notify` callback being triggered for the same attribute. Subscriptions can be removed with use of `bt_gatt_unsubscribe()` API.

Note

When subscriptions are removed `notify` callback is called with the data set to NULL.

API Reference

Generic Attribute Profile (GATT)

Related code samples

- Bluetooth logging backendSend log messages over Bluetooth using the Bluetooth logging backend.
- Bluetooth Mesh badgeImplement a smart badge using the reel board and Bluetooth Mesh.
- Cycling Speed and Cadence (CSC) PeripheralExpose a Cycling Speed and Cadence (CSC) GATT Service.
- DIS PeripheralExpose device information using the Device Information Service (DIS).
- ESP PeripheralExpose environmental information using the Environmental Sensing Profile (ESP).
- HID PeripheralImplement a Bluetooth HID peripheral (generic mouse)
- MTU UpdateConfigure and exchange MTU between two devices.
- PeripheralImplement basic Bluetooth LE Peripheral role functionality (advertising and exposing GATT services).
- Peripheral Accept ListAdvertise and accept connections only from devices on an accept list.
- Peripheral GATT WriteWrite a value to a characteristic using GATT Write Without Response.
- ST Bluetooth LE Sensor DemoExport vendor-specific GATT services over Bluetooth.

GATT Server

GATT Server APIs

GATT Client

GATT Client APIs

Attribute Protocol (ATT)

API Reference

Attribute Protocol (ATT)

Bluetooth Mesh

Bluetooth Mesh Profile The Bluetooth Mesh profile adds secure wireless multi-hop communication for Bluetooth Low Energy. This module implements the Bluetooth Mesh Protocol Specification v1.1.1.

Read more about Bluetooth Mesh on the Bluetooth SIG Website.

Core The core provides functionality for managing the general Bluetooth Mesh state.

Low Power Node The Low Power Node (LPN) role allows battery powered devices to participate in a mesh network as a leaf node. An LPN interacts with the mesh network through a Friend node, which is responsible for relaying any messages directed to the LPN. The LPN saves power by keeping its radio turned off, and only wakes up to either send messages or poll the Friend node for any incoming messages.

The radio control and polling is managed automatically by the mesh stack, but the LPN API allows the application to trigger the polling at any time through `bt_mesh_lpn_poll()`. The LPN operation parameters, including poll interval, poll event timing and Friend requirements is controlled through the `CONFIG_BT_MESH_LOW_POWER` option and related configuration options.

When using the LPN feature with logging, it is strongly recommended to only use the `CONFIG_LOG_MODE_DEFERRED` option. Log modes other than the deferred may cause unintended delays during processing of log messages. This in turns will affect scheduling of the receive delay and receive window. The same limitation applies for the `CONFIG_BT_MESH_FRIEND` option.

Replay Protection List The Replay Protection List (RPL) is used to hold recently received sequence numbers from elements within the mesh network to perform protection against replay attacks.

To keep a node protected against replay attacks after reboot, it needs to store the entire RPL in the persistent storage before it is powered off. Depending on the amount of traffic in a mesh network, storing recently seen sequence numbers can make flash wear out sooner or later. To mitigate this, `CONFIG_BT_MESH_RPL_STORE_TIMEOUT` can be used. This option postpones storing of RPL entries in the persistent storage.

This option, however, doesn't completely solve the issue as the node may get powered off before the timer to store the RPL is fired. To ensure that messages can not be replayed, the node can initiate storage of the pending RPL entry (or entries) at any time (or sufficiently before power loss) by calling `bt_mesh_rpl_pending_store()`. This is up to the node to decide, which RPL entries are to be stored in this case.

Setting `CONFIG_BT_MESH_RPL_STORE_TIMEOUT` to -1 allows to completely switch off the timer, which can help to significantly reduce flash wear out. This moves the responsibility of storing RPL to the user application and requires that sufficient power backup is available from the time this API is called until all RPL entries are written to the flash.

Finding the right balance between `CONFIG_BT_MESH_RPL_STORE_TIMEOUT` and calling `bt_mesh_rpl_pending_store()` may reduce a risk of security vulnerability and flash wear out.

Persistent storage The mesh stack uses the *Settings Subsystem* for storing the device configuration persistently. When the stack configuration changes and the change needs to be stored persistently, the stack schedules a work item. The delay between scheduling the work item and submitting it to the workqueue is defined by the `CONFIG_BT_MESH_STORE_TIMEOUT` option. Once storing of data is scheduled, it can not be rescheduled until the work item is processed. Exceptions are made in certain cases as described below.

When IV index, Sequence Number, CDB configuration or a Replay Protection List reset have to be stored, the work item is submitted to the workqueue without the delay. If the work item was previously scheduled, it will be rescheduled without the delay.

The Replay Protection List and the Solicitation Replay Protection List are stored by a separate work item, scheduled with the delay defined by `CONFIG_BT_MESH_RPL_STORE_TIMEOUT`. Pending stores of other kinds do not shorten this delay. If the option is set to `-1`, the replay protection lists are stored together with the rest of the stack configuration.

When a work item is running, the stack will store all pending configuration that the work item is responsible for.

Work item execution context The `CONFIG_BT_MESH_SETTINGS_WORKQ` option configures the context from which the work item is executed. This option is enabled by default, and results in stack using a dedicated cooperative thread to process the work item. This allows the stack to process other incoming and outgoing messages, as well as other work items submitted to the system workqueue, while the stack configuration is being stored.

When this option is disabled, the work item is submitted to the system workqueue. This means that the system workqueue is blocked for the time it takes to store the stack's configuration. It is not recommended to disable this option as this will make the device non-responsive for a noticeable amount of time.

Advertisement identity All mesh stack bearers advertise data with the `BT_ID_DEFAULT` local identity. The value is preset in the mesh stack implementation. When Bluetooth® Low Energy (LE) and Bluetooth Mesh coexist on the same device, the application should allocate and configure another local identity for Bluetooth LE purposes before starting the communication.

API reference

Bluetooth Mesh

Related code samples

- Bluetooth Mesh badgeImplement a smart badge using the reel board and Bluetooth Mesh.
- MeshUse basic Bluetooth Mesh functionality.
- Mesh DemoImplement a Bluetooth Mesh demo application.
- Mesh ModelsSetup a Bluetooth Mesh node with various models (generic OnOff, generic Level, ...).
- Mesh OnOff ModelControl LEDs on a mesh network using the Bluetooth Mesh OnOff model.
- Mesh ProvisionerProvision a node and configure it using the Bluetooth Mesh APIs.

Access layer The access layer is the application's interface to the Bluetooth Mesh network. The access layer provides mechanisms for compartmentalizing the node behavior into elements and models, which are implemented by the application.

Mesh models The functionality of a mesh node is represented by models. A model implements a single behavior the node supports, like being a light, a sensor or a thermostat. The mesh models are grouped into *elements*. Each element is assigned its own unicast address, and may only contain one of each type of model. Conventionally, each element represents a single aspect of the mesh node behavior. For instance, a node that contains a sensor, two lights and a power outlet would spread this functionality across four elements, with each element instantiating all the models required for a single aspect of the supported behavior.

The node's element and model structure is specified in the node composition data, which is passed to `bt_mesh_init()` during initialization. The Bluetooth SIG have defined a set of foundation models (see *Mesh models*) and a set of models for implementing common behavior in the

Bluetooth Mesh Model Specification. All models not specified by the Bluetooth SIG are vendor models, and must be tied to a Company ID.

Mesh models have several parameters that can be configured either through initialization of the mesh stack or with the *Configuration Server*:

Opcode list The opcode list contains all message opcodes the model can receive, as well as the minimum acceptable payload length and the callback to pass them to. Models can support any number of opcodes, but each opcode can only be listed by one model in each element.

The full opcode list must be passed to the model structure in the composition data, and cannot be changed at runtime. The end of the opcode list is determined by the special `BT_MESH_MODEL_OP_END` entry. This entry must always be present in the opcode list, unless the list is empty. In that case, `BT_MESH_MODEL_NO_OPS` should be used in place of a proper opcode list definition.

AppKey list The AppKey list contains all the application keys the model can receive messages on. Only messages encrypted with application keys in the AppKey list will be passed to the model.

The maximum number of supported application keys each model can hold is configured with the `CONFIG_BT_MESH_MODEL_KEY_COUNT` configuration option. The contents of the AppKey list is managed by the *Configuration Server*.

Subscription list A model will process all messages addressed to the unicast address of their element (given that the utilized application key is present in the AppKey list). Additionally, the model will process packets addressed to any group or virtual address in its subscription list. This allows nodes to address multiple nodes throughout the mesh network with a single message.

The maximum number of supported addresses in the Subscription list each model can hold is configured with the `CONFIG_BT_MESH_MODEL_GROUP_COUNT` configuration option. The contents of the subscription list is managed by the *Configuration Server*.

Model publication The models may send messages in two ways:

- By specifying a set of message parameters in a `bt_mesh_msg_ctx`, and calling `bt_mesh_model_send()`.
- By setting up a `bt_mesh_model_pub` structure and calling `bt_mesh_model_publish()`.

When publishing messages with `bt_mesh_model_publish()`, the model will use the publication parameters configured by the *Configuration Server*. This is the recommended way to send unprompted model messages, as it passes the responsibility of selecting message parameters to the network administrator, which likely knows more about the mesh network than the individual nodes will.

To support publishing with the publication parameters, the model must allocate a packet buffer for publishing, and pass it to `bt_mesh_model_pub.msg`. The Config Server may also set up period publication for the publication message. To support this, the model must populate the `bt_mesh_model_pub.update` callback. The `bt_mesh_model_pub.update` callback will be called right before the message is published, allowing the model to change the payload to reflect its current state.

By setting `bt_mesh_model_pub.retr_update` to 1, the model can configure the `bt_mesh_model_pub.update` callback to be triggered on every retransmission. This can, for example, be used by models that make use of a Delay parameter, which can be adjusted for every retransmission. The `bt_mesh_model_pub_is_retransmission()` function can be used to differentiate a first publication and a retransmission. The `BT_MESH_PUB_MSG_TOTAL` and `BT_MESH_PUB_MSG_NUM` macros can be used to return total number of transmissions and the retransmission number within one publication interval.

Extended models The Bluetooth Mesh specification allows the mesh models to extend each other. When a model extends another, it inherits that model's functionality, and extension can be used to construct complex models out of simple ones, leveraging the existing model functionality to avoid defining new opcodes. Models may extend any number of models, from any element. When one model extends another in the same element, the two models will share subscription lists. The mesh stack implements this by merging the subscription lists of the two models into one, combining the number of subscriptions the models can have in total. Models may extend models that extend others, creating an "extension tree". All models in an extension tree share a single subscription list per element it spans.

Model extensions are done by calling `bt_mesh_model_extend()` during initialization. A model can only be extended by one other model, and extensions cannot be circular. Note that binding of node states and other relationships between the models must be defined by the model implementations.

The model extension concept adds some overhead in the access layer packet processing, and must be explicitly enabled with `CONFIG_BT_MESH_MODEL_EXTENSIONS` to have any effect.

Model data storage Mesh models may have data associated with each model instance that needs to be stored persistently. The access API provides a mechanism for storing this data, leveraging the internal model instance encoding scheme. Models can store one user defined data entry per instance by calling `bt_mesh_model_data_store()`. To be able to read out the data the next time the device reboots, the model's `bt_mesh_model_cb.settings_set` callback must be populated. This callback gets called when model specific data is found in the persistent storage. The model can retrieve the data by calling the `read_cb` passed as a parameter to the callback. See the *Settings* module documentation for details.

When model data changes frequently, storing it on every change may lead to increased wear of flash. To reduce the wear, the model can postpone storing of data by calling `bt_mesh_model_data_store_schedule()`. The stack will schedule a work item with delay defined by the `CONFIG_BT_MESH_STORE_TIMEOUT` option. When the work item is running, the stack will call the `bt_mesh_model_cb.pending_store` callback for every model that has requested storing of data. The model can then call `bt_mesh_model_data_store()` to store the data.

If `CONFIG_BT_MESH_SETTINGS_WORKQ` is enabled, the `bt_mesh_model_cb.pending_store` callback is called from a dedicated thread. This allows the stack to process other incoming and outgoing messages while model data is being stored. It is recommended to use this option and the `bt_mesh_model_data_store_schedule()` function when large amount of data needs to be stored.

Composition Data The Composition Data provides information about a mesh device. A device's Composition Data holds information about the elements on the device, the models that it supports, and other features. The Composition Data is split into different pages, where each page contains specific feature information about the device. In order to access this information, the user may use the *Configuration Server* model or, if supported, the *Large Composition Data Server* model.

Composition Data Page 0 Composition Data Page 0 provides the fundamental information about a device, and is mandatory for all mesh devices. It contains the element and model composition, the supported features, and manufacturer information.

Composition Data Page 1 Composition Data Page 1 provides information about the relationships between models, and is mandatory for all mesh devices. A model may extend and/or correspond to one or more models. A model can extend another model by calling `bt_mesh_model_extend()`, or correspond to another model by calling `bt_mesh_model_correspond()`. `CONFIG_BT_MESH_MODEL_EXTENSION_LIST_SIZE` specifies how many model relations can be stored in the composition on a device, and this number should reflect the number of `bt_mesh_model_extend()` and `bt_mesh_model_correspond()` calls.

Composition Data Page 2 Composition Data Page 2 provides information for supported mesh profiles. Mesh profile specifications define product requirements for devices that want to support a specific Bluetooth SIG defined profile. Currently supported profiles can be found in section 3.12 in Bluetooth SIG Assigned Numbers. Composition Data Page 2 is only mandatory for devices that claim support for one or more mesh profile(s).

Composition Data Pages 128, 129 and 130 Composition Data Pages 128, 129 and 130 mirror Composition Data Pages 0, 1 and 2 respectively. They are used to represent the new content of the mirrored pages when the Composition Data will change after a firmware update. See *Composition Data and Models Metadata* for details.

Delayable messages The delayable message functionality is enabled with Kconfig option `CONFIG_BT_MESH_ACCESS_DELAYABLE_MSG`. This is an optional functionality that implements specification recommendations for messages that are transmitted by a model in a response to a received message, also called response messages.

Response messages should be sent with the following random delays:

- Between 20 and 50 milliseconds if the received message was sent to a unicast address
- Between 20 and 500 milliseconds if the received message was sent to a group or virtual address

The delayable message functionality is triggered if the `bt_mesh_msg_ctx.rnd_delay` flag is set. The delayable message functionality stores messages in the local memory while they are waiting for the random delay expiration.

If the transport layer doesn't have sufficient memory to send a message at the moment the random delay expires, the message is postponed for another 10 milliseconds. If the transport layer cannot send a message for any other reason, the delayable message functionality raises the `bt_mesh_send_cb.start` callback with a transport layer error code.

If the delayable message functionality cannot find enough free memory to store an incoming message, it will send messages with delay close to expiration to free memory.

When the mesh stack is suspended or reset, messages not yet sent are removed and the `bt_mesh_send_cb.start` callback is raised with an error code.

> **Note**
>
> When a model sends several messages in a row, it may happen that the messages are not sent in the order they were passed to the access layer. This is because some messages can be delayed for a longer time than the others.
>
> Disable the randomization by setting the `bt_mesh_msg_ctx.rnd_delay` to `false`, when a set of messages originated by the same model needs to be sent in a certain order.

Delayable publications The delayable publication functionality implements the specification recommendations for message publication delays in the following cases:

- Between 20 to 500 milliseconds when the Bluetooth Mesh stack starts or when the publication is triggered by the `bt_mesh_model_publish()` function
- Between 20 to 50 milliseconds for periodically published messages

This feature is optional and enabled with the `CONFIG_BT_MESH_DELAYABLE_PUBLICATION` Kconfig option. When enabled, each model can enable or disable the delayable publication by setting the `bt_mesh_model_pub.delayable` bit field to `1` or `0` correspondingly. This bit field can be changed at any time.

API reference
Access layer

Mesh models

Foundation models The Bluetooth Mesh specification defines foundation models that can be used by network administrators to configure and diagnose mesh nodes.

Bridge Configuration Client The Bridge Configuration Client is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled through the `CONFIG_BT_MESH_BRG_CFG_CLI` option.

The Bridge Configuration Client model provides functionality for configuring the subnet bridge functionality of another Mesh node containing the *Bridge Configuration Server*. The device key of the node containing the target Bridge Configuration Server is used for access layer security.

If present, the Bridge Configuration Client model must only be instantiated on the primary element.

API reference
Bridge Configuration Client Model

Bridge Configuration Server The Bridge Configuration Server model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_BRG_CFG_SRV` configuration option. This model extends the *Configuration Server* model.

The Bridge Configuration Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used for supporting and configuring the Subnet Bridge feature.

The Bridge Configuration Server model relies on a *Bridge Configuration Client* to configure it. The Bridge Configuration Server model only accepts messages encrypted with the node's device key.

If present, the Bridge Configuration Server model must be instantiated on the primary element.

The Bridge Configuration Server model provides access to the following three states:

- Subnet Bridge
- Bridging Table
- Bridging Table Size

For more information about the states, see *Subnet Bridge states*.

API reference
Bridge Configuration Server Model

Configuration Client The Configuration Client model is a foundation model defined by the Bluetooth Mesh specification. It provides functionality for configuring most parameters of a mesh node, including encryption keys, model configuration and feature enabling.

The Configuration Client model communicates with a *Configuration Server* model using the device key of the target node. The Configuration Client model may communicate with servers on other nodes or self-configure through the local Configuration Server model.

All configuration functions in the Configuration Client API have `net_idx` and `addr` as their first parameters. These should be set to the network index and primary unicast address that the target node was provisioned with.

The Configuration Client model is optional, and it must only be instantiated on the primary element if present in the Composition Data.

API reference
Configuration Client Model

Configuration Server The Configuration Server model is a foundation model defined by the Bluetooth Mesh specification. The Configuration Server model controls most parameters of the mesh node. It does not have an API of its own, but relies on a *Configuration Client* to control it.

The Configuration Server model is mandatory on all Bluetooth Mesh nodes, and must only be instantiated on the primary element.

API reference
Configuration Server Model

Health Client The Health Client model interacts with a Health Server model to read out diagnostics and control the node's attention state.

All message passing functions in the Health Client API have `cli` as their first parameter. This is a pointer to the client model instance to be used in this function call. The second parameter is the `ctx` or message context. Message context contains netkey index, appkey index and unicast address that the target node uses.

The Health Client model is optional, and may be instantiated on any element. However, if a Health Client model is instantiated on an element other than the primary, an instance must also be present on the primary element.

See *Health faults* for a list of specification defined fault values.

API reference
Health Client Model

Health Server The Health Server model provides attention callbacks and node diagnostics for *Health Client* models. It is primarily used to report faults in the mesh node and map the mesh nodes to their physical location.

If present, the Health Server model must be instantiated on the primary element.

Faults The Health Server model may report a list of faults that have occurred in the device's lifetime. Typically, the faults are events or conditions that may alter the behavior of the node, like power outages or faulty peripherals. Faults are split into warnings and errors. Warnings indicate conditions that are close to the limits of what the node is designed to withstand, but not necessarily damaging to the device. Errors indicate conditions that are outside of the node's design limits, and may have caused invalid behavior or permanent damage to the device.

Fault values `0x01` to `0x7f` are reserved for the Bluetooth Mesh specification, and the full list of specification defined faults are available in *Health faults*. Fault values `0x80` to `0xff` are vendor specific. The list of faults are always reported with a company ID to help interpreting the vendor specific faults.

Attention state The attention state is used to make the device call attention to itself through some physical behavior like blinking, playing a sound or vibrating. The attention state may be used during provisioning to let the user know which device they're provisioning, as well as through the Health models at runtime.

The attention state is always assigned a timeout in the range of one to 255 seconds when enabled. The Health Server API provides two callbacks for the application to run their attention calling behavior: `bt_mesh_health_srv_cb.attn_on` is called at the beginning of the attention period, `bt_mesh_health_srv_cb.attn_off` is called at the end.

The remaining time for the attention period may be queried through `bt_mesh_health_srv.attn_timer`.

API reference
Health Server Model

Health faults Fault values defined by the Bluetooth Mesh specification.

Health faults

Large Composition Data Client The Large Composition Data Client model is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled through the `CONFIG_BT_MESH_LARGE_COMP_DATA_CLI` option.

The Large Composition Data Client model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and supports the functionality of reading pages of Composition Data that do not fit in a Config Composition Data Status message and reading the metadata of the model instances on a node that supports the *Large Composition Data Server* model.

The Large Composition Data Client model communicates with a Large Composition Data Server model using the device key of the node containing the target Large Composition Data Server model instance.

If present, the Large Composition Data Client model must only be instantiated on the primary element.

API reference
Large Composition Data Client model

Large Composition Data Server The Large Composition Data Server model is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled through the `CONFIG_BT_MESH_LARGE_COMP_DATA_SRV` option.

The Large Composition Data Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of exposing pages of Composition Data that do not fit in a Config Composition Data Status message and to expose metadata of the model instances.

The Large Composition Data Server does not have an API of its own and relies on a *Large Composition Data Client* to control it. The model only accepts messages encrypted with the node's device key.

If present, the Large Composition Data Server model must only be instantiated on the primary element.

Models metadata The Large Composition Data Server model allows each model to have a list of model's specific metadata that can be read by the Large Composition Data Client model. The metadata list can be associated with the `bt_mesh_model` through the `bt_mesh_model.metadata` field. The metadata list consists of one or more entries defined by the `bt_mesh_models_metadata_entry` structure. Each entry contains the length and ID of the metadata, and a pointer to the raw data. Entries can be created using the `BT_MESH_MODELS_METADATA_ENTRY` macro. The `BT_MESH_MODELS_METADATA_END` macro marks the end of the metadata list and must always be present. If the model has no metadata, the helper macro `BT_MESH_MODELS_METADATA_NONE` can be used instead.

API reference
Large Composition Data Server model

On-Demand Private Proxy Client The On-Demand Private Proxy Client model is a foundation model defined by the Bluetooth Mesh specification. The model is optional, and is enabled with the `CONFIG_BT_MESH_OD_PRIV_PROXY_CLI` option.

The On-Demand Private Proxy Client model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to set and retrieve the On-Demand Private GATT Proxy state. The state defines how long a node will advertise Mesh Proxy Service with Private Network Identity type after it receives a Solicitation PDU.

The On-Demand Private Proxy Client model communicates with an On-Demand Private Proxy Server model using the device key of the node containing the target On-Demand Private Proxy Server model instance.

If present, the On-Demand Private Proxy Client model must only be instantiated on the primary element.

Configurations The On-Demand Private Proxy Client model behavior can be configured with the transmission timeout option `CONFIG_BT_MESH_OD_PRIV_PROXY_CLI_TIMEOUT`. The `CONFIG_BT_MESH_OD_PRIV_PROXY_CLI_TIMEOUT` controls how long the Client waits for a state response message to arrive in milliseconds. This value can be changed at runtime using `bt_mesh_od_priv_proxy_cli_timeout_set()`.

API reference
Bluetooth Mesh On-Demand Private GATT Proxy Client

On-Demand Private Proxy Server The On-Demand Private Proxy Server model is a foundation model defined by the Bluetooth Mesh specification. It is enabled with the `CONFIG_BT_MESH_OD_PRIV_PROXY_SRV` option.

The On-Demand Private Proxy Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and supports the configuration of advertising with Private Network Identity type of a node that is a recipient of Solicitation PDUs by managing its On-Demand Private GATT Proxy state.

When enabled, the *Solicitation PDU RPL Configuration Server* is also enabled. The On-Demand Private Proxy Server is dependent on the *Private Beacon Server* to be present on the node.

The On-Demand Private Proxy Server does not have an API of its own, and relies on a *On-Demand Private Proxy Client* to control it. The On-Demand Private Proxy Server model only accepts messages encrypted with the node's device key.

If present, the On-Demand Private Proxy Server model must only be instantiated on the primary element.

API reference
Bluetooth Mesh On-Demand Private GATT Proxy Server

Opcodes Aggregator Client The Opcodes Aggregator Client model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_OP_AGG_CLI` option.

The Opcodes Aggregator Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of dispatching a sequence of access layer messages to nodes supporting the *Opcodes Aggregator Server* model.

The Opcodes Aggregator Client model communicates with an Opcodes Aggregator Server model using the device key of the target node or the application keys configured by the Configuration Client.

If present, the Opcodes Aggregator Client model must only be instantiated on the primary element.

The Opcodes Aggregator Client model is implicitly bound to the device key on initialization. It should be bound to the same application keys as the client models that are used to produce the sequence of messages.

To be able to aggregate a message from a client model, it should support an asynchronous API, for example through callbacks.

API reference

Opcodes Aggregator Client model

Opcodes Aggregator Server The Opcodes Aggregator Server model is a foundation model defined by the Bluetooth mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_OP_AGG_SRV` option.

The Opcodes Aggregator Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of processing a sequence of access layer messages.

The Opcodes Aggregator Server model accepts messages encrypted with the node's device key or the application keys.

If present, the Opcodes Aggregator Server model must only be instantiated on the primary element.

The targeted server models should be bound to the same application key that is used to encrypt the sequence of access layer messages sent to the Opcodes Aggregator Server.

The Opcodes Aggregator Server handles aggregated messages and dispatches them to the respective models and their message handlers. Current implementation assumes that responses are sent from the same execution context as the received message and doesn't allow to send a postponed response, for example from a work queue.

API reference

Opcodes Aggregator Server model

Private Beacon Client The Private Beacon Client model is a foundation model defined by the Bluetooth mesh specification. It is enabled with the `CONFIG_BT_MESH_PRIV_BEACON_CLI` option.

The Private Beacon Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and provides functionality for configuring the *Private Beacon Server* models.

The Private Beacons feature adds privacy to the different Bluetooth Mesh beacons by periodically randomizing the beacon input data. This protects the mesh node from being tracked by devices outside the mesh network, and hides the network's IV index, IV update and the Key Refresh state.

The Private Beacon Client model communicates with a *Private Beacon Server* model using the device key of the target node. The Private Beacon Client model may communicate with servers on other nodes or self-configure through the local Private Beacon Server model.

All configuration functions in the Private Beacon Client API have `net_idx` and `addr` as their first parameters. These should be set to the network index and the primary unicast address the target node was provisioned with.

If present, the Private Beacon Client model must only be instantiated on the primary element.

API reference

Bluetooth Mesh Private Beacon Client

Private Beacon Server The Private Beacon Server model is a foundation model defined by the Bluetooth mesh specification. It is enabled with `CONFIG_BT_MESH_PRIV_BEACON_SRV` option.

The Private Beacon Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and controls the mesh node's Private Beacon state, Private GATT Proxy state and Private Node Identity state.

The Private Beacons feature adds privacy to the different Bluetooth Mesh beacons by periodically randomizing the beacon input data. This protects the mesh node from being tracked by devices outside the mesh network, and hides the network's IV index, IV update and the Key Refresh state. The Private Beacon Server must be instantiated for the device to support sending of the private beacons, but the node will process received private beacons without it.

The Private Beacon Server does not have an API of its own, but relies on a *Private Beacon Client* to control it. The Private Beacon Server model only accepts messages encrypted with the node's device key.

The application can configure the initial parameters of the Private Beacon Server model through the `bt_mesh_priv_beacon_srv` instance passed to `BT_MESH_MODEL_PRIV_BEACON_SRV`. Note that if the mesh node stored changes to this configuration in the settings subsystem, the initial values may be overwritten upon loading.

If present, the Private Beacon Server model must only be instantiated on the primary element.

API reference

Bluetooth Mesh Private Beacon Server

Remote Provisioning Client The Remote Provisioning Client model is a foundation model defined by the Bluetooth mesh specification. It is enabled with the `CONFIG_BT_MESH_RPR_CLI` option.

The Remote Provisioning Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1. This model provides functionality to remotely provision devices into a mesh network, and perform Node Provisioning Protocol Interface procedures by interacting with mesh nodes that support the *Remote Provisioning Server* model.

The Remote Provisioning Client model communicates with a Remote Provisioning Server model using the device key of the node containing the target Remote Provisioning Server model instance.

If present, the Remote Provisioning Client model must be instantiated on the primary element.

Scanning The scanning procedure is used to scan for unprovisioned devices located nearby the Remote Provisioning Server. The Remote Provisioning Client starts a scan procedure by using the `bt_mesh_rpr_scan_start()` call:

```
static void rpr_scan_report(struct bt_mesh_rpr_cli *cli,
            const struct bt_mesh_rpr_node *srv,
            struct bt_mesh_rpr_unprov *unprov,
            struct net_buf_simple *adv_data)
{

}

struct bt_mesh_rpr_cli rpr_cli = {
   .scan_report = rpr_scan_report,
};

const struct bt_mesh_rpr_node srv = {
   .addr = 0x0004,
   .net_idx = 0,
```

(continues on next page)

(continued from previous page)

```
    .ttl = BT_MESH_TTL_DEFAULT,
};

struct bt_mesh_rpr_scan_status status;
uint8_t *uuid = NULL;
uint8_t timeout = 10;
uint8_t max_devs = 3;

bt_mesh_rpr_scan_start(&rpr_cli, &srv, uuid, timeout, max_devs, &status);
```

The above example shows pseudo code for starting a scan procedure on the target Remote Provisioning Server node. This procedure will start a ten-second, multiple-device scanning where the generated scan report will contain a maximum of three unprovisioned devices. If the UUID argument was specified, the same procedure would only scan for the device with the corresponding UUID. After the procedure completes, the server sends the scan report that will be handled in the client's `bt_mesh_rpr_cli.scan_report` callback.

Additionally, the Remote Provisioning Client model also supports extended scanning with the `bt_mesh_rpr_scan_start_ext()` call. Extended scanning supplements regular scanning by allowing the Remote Provisioning Server to report additional data for a specific device. The Remote Provisioning Server will use active scanning to request a scan response from the unprovisioned device if it is supported by the unprovisioned device.

Provisioning The Remote Provisioning Client starts a provisioning procedure by using the `bt_mesh_provision_remote()` call:

```
struct bt_mesh_rpr_cli rpr_cli;

const struct bt_mesh_rpr_node srv = {
    .addr = 0x0004,
    .net_idx = 0,
    .ttl = BT_MESH_TTL_DEFAULT,
};

uint8_t uuid[16] = { 0xaa };
uint16_t addr = 0x0006;
uint16_t net_idx = 0;

bt_mesh_provision_remote(&rpr_cli, &srv, uuid, net_idx, addr);
```

The above example shows pseudo code for remotely provisioning a device through a Remote Provisioning Server node. This procedure will attempt to provision the device with the corresponding UUID, and assign the address 0x0006 to its primary element using the network key located at index zero.

> **Note**
>
> During the remote provisioning, the same `bt_mesh_prov` callbacks are triggered as for ordinary provisioning. See section *Provisioning* for further details.

Re-provisioning In addition to scanning and provisioning functionality, the Remote Provisioning Client also provides means to reconfigure node addresses, device keys and Composition Data on devices that support the *Remote Provisioning Server* model. This is provided through the Node Provisioning Protocol Interface (NPPI) which supports the following three procedures:

- Device Key Refresh procedure: Used to change the device key of the Target node without a need to reconfigure the node.
- Node Address Refresh procedure: Used to change the node's device key and unicast address.
- Node Composition Refresh procedure: Used to change the device key of the node, and to add or delete models or features of the node.

The three NPPI procedures can be initiated with the `bt_mesh_reprovision_remote()` call:

```
struct bt_mesh_rpr_cli rpr_cli;
struct bt_mesh_rpr_node srv = {
   .addr = 0x0006,
   .net_idx = 0,
   .ttl = BT_MESH_TTL_DEFAULT,
};

bool composition_changed = false;
uint16_t new_addr = 0x0009;

bt_mesh_reprovision_remote(&rpr_cli, &srv, new_addr, composition_changed);
```

The above example shows pseudo code for triggering a Node Address Refresh procedure on the Target node. The specific procedure is not chosen directly, but rather through the other parameters that are inputted. In the example we can see that the current unicast address of the Target is 0x0006, while the new address is set to 0x0009. If the two addresses were the same, and the `composition_changed` flag was set to true, this code would instead trigger a Node Composition Refresh procedure. If the two addresses were the same, and the `composition_changed` flag was set to false, this code would trigger a Device Key Refresh procedure.

API reference

Remote Provisioning Client model

Remote Provisioning Server The Remote Provisioning Server model is a foundation model defined by the Bluetooth mesh specification. It is enabled with the `CONFIG_BT_MESH_RPR_SRV` option.

The Remote Provisioning Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and is used to support the functionality of remotely provisioning devices into a mesh network.

The Remote Provisioning Server does not have an API of its own, but relies on a *Remote Provisioning Client* to control it. The Remote Provisioning Server model only accepts messages encrypted with the node's device key.

If present, the Remote Provisioning Server model must be instantiated on the primary element.

Note that after refreshing the device key, node address or Composition Data through a Node Provisioning Protocol Interface (NPPI) procedure, the `bt_mesh_prov.reprovisioned` callback is triggered. See section *Remote Provisioning Client* for further details.

How to integrate the model into an application To add the Remote Provisioning Server model in your application, do the following:

1. Enable Kconfig in your project configuration:

   ```
   CONFIG_BT_MESH_RPR_SRV=y
   ```

2. Add the model instance into the model list `bt_mesh_elem.models` of the primary element using the `BT_MESH_MODEL_RPR_SRV` macro, for example:

```
static const struct bt_mesh_model models[] = {
        BT_MESH_MODEL_CFG_SRV,
        BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
        BT_MESH_MODEL_RPR_SRV,
        /* ... */
};

static const struct bt_mesh_elem elements[] = {
        BT_MESH_ELEM(0, models, BT_MESH_MODEL_NONE),
};
```

3. Enable PB-Remote by calling `bt_mesh_prov_enable()` with `BT_MESH_PROV_REMOTE`:

```
err = bt_mesh_prov_enable(BT_MESH_PROV_REMOTE);
if (err) {
        printk("PB-Remote enable failed (err %d)\n", err);
}
```

Limitations The following limitations apply to Remote Provisioning Server model:

- Provisioning of unprovisioned device using PB-GATT is not supported.
- All Node Provisioning Protocol Interface (NPPI) procedures are supported. However, if the composition data of a device gets changed after device firmware update (see *firmware effect*), it is not possible for the device to remain provisioned. The device should be unprovisioned if its composition data is expected to change.

API reference

Remote provisioning server

SAR Configuration Client The SAR Configuration Client model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_SAR_CFG_CLI` configuration option.

The SAR Configuration Client model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and it supports the configuration of the lower transport layer behavior of a node that supports the *SAR Configuration Server* model.

The model can send messages to query or change the states supported by the SAR Configuration Server (SAR Transmitter and SAR Receiver) using SAR Configuration messages.

The SAR Transmitter procedure is used to determine and configure the SAR Transmitter state of a SAR Configuration Server. Function calls `bt_mesh_sar_cfg_cli_transmitter_get()` and `bt_mesh_sar_cfg_cli_transmitter_set()` are used to get and set the SAR Transmitter state of the Target node respectively.

The SAR Receiver procedure is used to determine and configure the SAR Receiver state of a SAR Configuration Server. Function calls `bt_mesh_sar_cfg_cli_receiver_get()` and `bt_mesh_sar_cfg_cli_receiver_set()` are used to get and set the SAR Receiver state of the Target node respectively.

For more information about the two states, see *SAR states*.

An element can send any SAR Configuration Client message at any time to query or change the states supported by the SAR Configuration Server model of a peer node. The SAR Configuration Client model only accepts messages encrypted with the device key of the node supporting the SAR Configuration Server model.

If present, the SAR Configuration Client model must only be instantiated on the primary element.

API reference
Bluetooth Mesh SAR Configuration Client Model

SAR Configuration Server The SAR Configuration Server model is a foundation model defined by the Bluetooth Mesh specification. It is an optional model, enabled with the `CONFIG_BT_MESH_SAR_CFG_SRV` configuration option.

The SAR Configuration Server model is introduced in the Bluetooth Mesh Protocol Specification version 1.1, and it supports the configuration of the *segmentation and reassembly (SAR)* behavior of a Bluetooth Mesh node. The model defines a set of states and messages for the SAR configuration.

The SAR Configuration Server model defines two states, SAR Transmitter state and SAR Receiver state. For more information about the two states, see *SAR states*.

The model also supports the SAR Transmitter and SAR Receiver get and set messages.

The SAR Configuration Server model does not have an API of its own, but relies on a *SAR Configuration Client* to control it. The SAR Configuration Server model only accepts messages encrypted with the node's device key.

If present, the SAR Configuration Server model must only be instantiated on the primary element.

API reference
Bluetooth Mesh SAR Configuration Server Model

Solicitation PDU RPL Configuration Client The Solicitation PDU RPL Configuration Client model is a foundation model defined by the Bluetooth mesh specification. The model is optional, and is enabled through the `CONFIG_BT_MESH_SOL_PDU_RPL_CLI` option.

The Solicitation PDU RPL Configuration Client model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and supports the functionality of removing addresses from the solicitation replay protection list (SRPL) of a node that supports the *Solicitation PDU RPL Configuration Server* model.

The Solicitation PDU RPL Configuration Client model communicates with a Solicitation PDU RPL Configuration Server model using the application keys configured by the Configuration Client.

If present, the Solicitation PDU RPL Configuration Client model must only be instantiated on the primary element.

Configurations The Solicitation PDU RPL Configuration Client model behavior can be configured with the transmission timeout option `CONFIG_BT_MESH_SOL_PDU_RPL_CLI_TIMEOUT`. The `CONFIG_BT_MESH_SOL_PDU_RPL_CLI_TIMEOUT` controls how long the Solicitation PDU RPL Configuration Client waits for a response message to arrive in milliseconds. This value can be changed at runtime using `bt_mesh_sol_pdu_rpl_cli_timeout_set()`.

API reference
Bluetooth Mesh Solicitation PDU RPL Client

Solicitation PDU RPL Configuration Server The Solicitation PDU RPL Configuration Server model is a foundation model defined by the Bluetooth mesh specification. The model is enabled if the node has the *On-Demand Private Proxy Server* enabled.

The Solicitation PDU RPL Configuration Server model was introduced in the Bluetooth Mesh Protocol Specification version 1.1, and manages the Solicitation Replay Protection List (SRPL) saved on the device. The SRPL is used to reject Solicitation PDUs that are already processed by a node. When a valid Solicitation PDU message is successfully processed by a node, the SSRC field and SSEQ field of the message are stored in the node's SRPL.

The Solicitation PDU RPL Configuration Server does not have an API of its own, and relies on a *Solicitation PDU RPL Configuration Client* to control it. The model only accepts messages encrypted with an application key as configured by the Configuration Client.

If present, the Solicitation PDU RPL Configuration Server model must only be instantiated on the primary element.

Configurations For the Solicitation PDU RPL Configuration Server model, the `CONFIG_BT_MESH_PROXY_SRPL_SIZE` option can be configured to set the size of the SRPL.

API reference

Bluetooth Mesh Solicitation PDU RPL Server

Model specification models In addition to the foundation models defined in the Bluetooth Mesh specification, the Bluetooth Mesh Model Specification defines several models, some of which are implemented in Zephyr:

BLOB Transfer models The Binary Large Object (BLOB) Transfer models implement the Bluetooth Mesh Binary Large Object Transfer Model specification version 1.0 and provide functionality for sending large binary objects from a single source to many Target nodes over the Bluetooth Mesh network. It is the underlying transport method for the *Device Firmware Update (DFU)*, but may be used for other object transfer purposes. The implementation is in experimental state.

The BLOB Transfer models support transfers of continuous binary objects of up to 4 GB (2^{32} bytes). The BLOB transfer protocol has built-in recovery procedures for packet losses, and sets up checkpoints to ensure that all targets have received all the data before moving on. Data transfer order is not guaranteed.

BLOB transfers are constrained by the transfer speed and reliability of the underlying mesh network. Under ideal conditions, the BLOBs can be transferred at a rate of up to 1 kbps, allowing a 100 kB BLOB to be transferred in 10-15 minutes. However, network conditions, transfer capabilities and other limiting factors can easily degrade the data rate by several orders of magnitude. Tuning the parameters of the transfer according to the application and network configuration, as well as scheduling it to periods with low network traffic, will offer significant improvements on the speed and reliability of the protocol. However, achieving transfer rates close to the ideal rate is unlikely in actual deployments.

There are two BLOB Transfer models:

BLOB Transfer Server The Binary Large Object (BLOB) Transfer Server model implements reliable receiving of large binary objects. It serves as the backend of the *Firmware Update Server*, but can also be used for receiving other binary images.

BLOBs As described in *BLOB Transfer models*, the binary objects transferred by the BLOB Transfer models are divided into blocks, which are divided into chunks. As the transfer is controlled by the BLOB Transfer Client model, the BLOB Transfer Server must allow blocks to come in any order. The chunks within a block may also come in any order, but all chunks in a block must be received before the next block is started.

The BLOB Transfer Server keeps track of the received blocks and chunks, and will process each block and chunk only once. The BLOB Transfer Server also ensures that any missing chunks are resent by the BLOB Transfer Client.

Usage The BLOB Transfer Server is instantiated on an element with a set of event handler callbacks:

```
static const struct bt_mesh_blob_srv_cb blob_cb = {
    /* Callbacks */
};

static struct bt_mesh_blob_srv blob_srv = {
    .cb = &blob_cb,
};

static const struct bt_mesh_model models[] = {
    BT_MESH_MODEL_BLOB_SRV(&blob_srv),
};
```

A BLOB Transfer Server is capable of receiving a single BLOB transfer at a time. Before the BLOB Transfer Server can receive a transfer, it must be prepared by the user. The transfer ID must be passed to the BLOB Transfer Server through the `bt_mesh_blob_srv_recv()` function before the transfer is started by the BLOB Transfer Client. The ID must be shared between the BLOB Transfer Client and the BLOB Transfer Server through some higher level procedure, like a vendor specific transfer management model.

Once the transfer has been set up on the BLOB Transfer Server, it's ready for receiving the BLOB. The application is notified of the transfer progress through the event handler callbacks, and the BLOB data is sent to the BLOB stream.

The interaction between the BLOB Transfer Server, BLOB stream and application is shown below:

Transfer suspension The BLOB Transfer Server keeps a running timer during the transfer, that is reset on every received message. If the BLOB Transfer Client does not send a message before the transfer timer expires, the transfer is suspended by the BLOB Transfer Server.

The BLOB Transfer Server notifies the user of the suspension by calling the `suspended` callback. If the BLOB Transfer Server is in the middle of receiving a block, this block is discarded.

The BLOB Transfer Client may resume a suspended transfer by starting a new block transfer. The BLOB Transfer Server notifies the user by calling the `resume` callback.

Transfer recovery The state of the BLOB transfer is stored persistently. If a reboot occurs, the BLOB Transfer Server will attempt to recover the transfer. When the Bluetooth Mesh subsystem is started (for instance by calling `bt_mesh_init()`), the BLOB Transfer Server will check for aborted transfers, and call the `recover` callback if there is any. In the recover callback, the user must provide a BLOB stream to use for the rest of the transfer. If the recover callback doesn't return successfully or does not provide a BLOB stream, the transfer is abandoned. If no recover callback is implemented, transfers are always abandoned after a reboot.

After a transfer is successfully recovered, the BLOB Transfer Server enters the suspended state. It will stay suspended until the BLOB Transfer Client resumes the transfer, or the user cancels it.

> **Note**
>
> The BLOB Transfer Client sending the transfer must support transfer recovery for the transfer to complete. If the BLOB Transfer Client has already given up the transfer, the BLOB Transfer Server will stay suspended until the application calls `bt_mesh_blob_srv_cancel()`.

API reference

Bluetooth Mesh BLOB Transfer Server model API

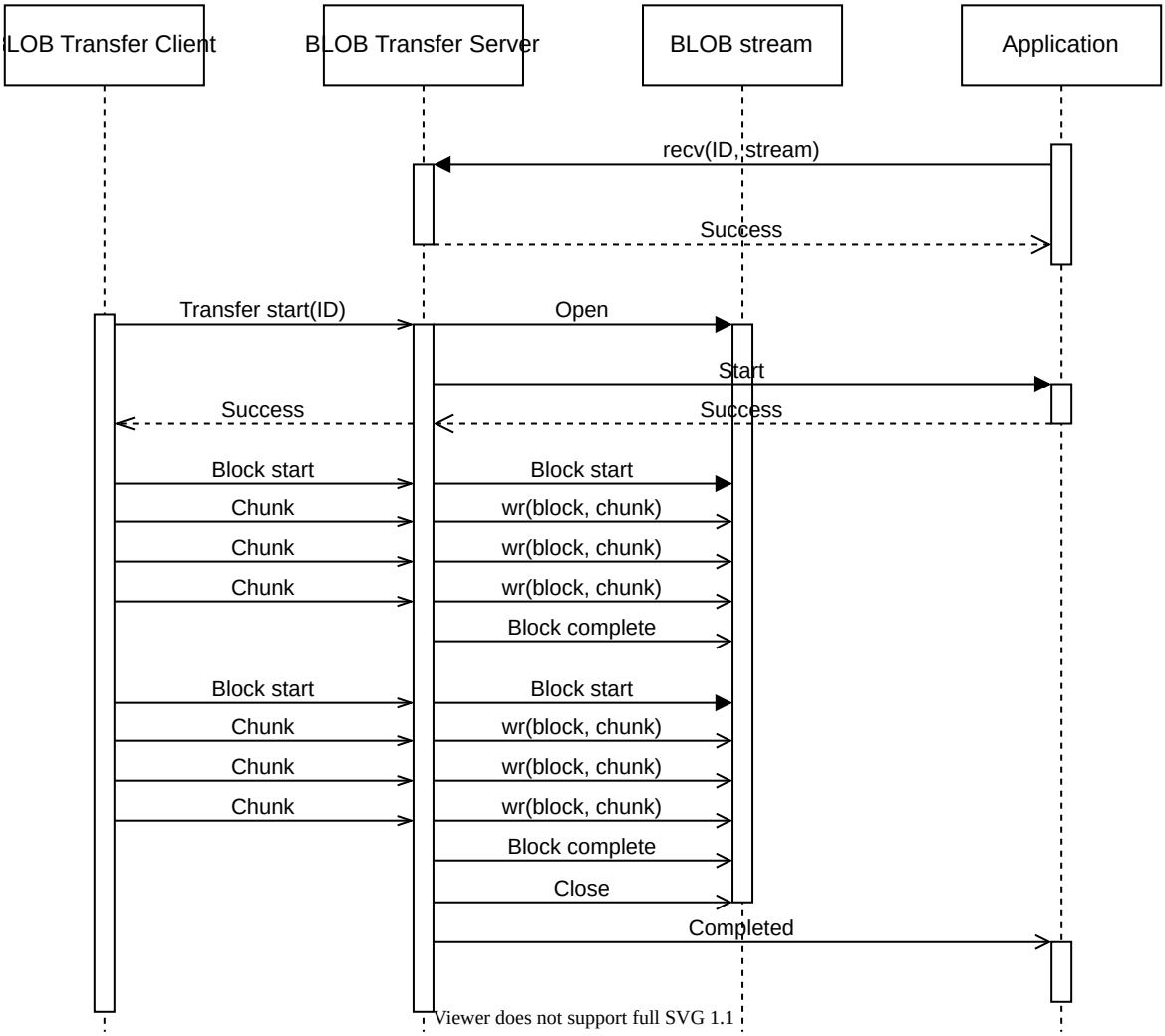


Fig. 19: BLOB Transfer Server model interaction

BLOB Transfer Client The Binary Large Object (BLOB) Transfer Client is the sender of the BLOB transfer. It supports sending BLOBs of any size to any number of Target nodes, in both Push BLOB Transfer Mode and Pull BLOB Transfer Mode.

Usage

Initialization The BLOB Transfer Client is instantiated on an element with a set of event handler callbacks:

```
static const struct bt_mesh_blob_cli_cb blob_cb = {
      /* Callbacks */
};

static struct bt_mesh_blob_cli blob_cli = {
      .cb = &blob_cb,
};

static const struct bt_mesh_model models[] = {
      BT_MESH_MODEL_BLOB_CLI(&blob_cli),
};
```

Transfer context Both the transfer capabilities retrieval procedure and the BLOB transfer uses an instance of a `bt_mesh_blob_cli_inputs` to determine how to perform the transfer. The BLOB Transfer Client Inputs structure must at least be initialized with a list of targets, an application key and a time to live (TTL) value before it is used in a procedure:

```
static struct bt_mesh_blob_target targets[3] = {
        { .addr = 0x0001 },
        { .addr = 0x0002 },
        { .addr = 0x0003 },
};
static struct bt_mesh_blob_cli_inputs inputs = {
        .app_idx = MY_APP_IDX,
        .ttl = BT_MESH_TTL_DEFAULT,
};

sys_slist_init(&inputs.targets);
sys_slist_append(&inputs.targets, &targets[0].n);
sys_slist_append(&inputs.targets, &targets[1].n);
sys_slist_append(&inputs.targets, &targets[2].n);
```

Note that all BLOB Transfer Servers in the transfer must be bound to the chosen application key.

Group address The application may additionally specify a group address in the context structure. If the group is not `BT_MESH_ADDR_UNASSIGNED`, the messages in the transfer will be sent to the group address, instead of being sent individually to each Target node. Mesh Manager must ensure that all Target nodes having the BLOB Transfer Server model subscribe to this group address.

Using group addresses for transferring the BLOBs can generally increase the transfer speed, as the BLOB Transfer Client sends each message to all Target nodes at the same time. However, sending large, segmented messages to group addresses in Bluetooth Mesh is generally less reliable than sending them to unicast addresses, as there is no transport layer acknowledgment mechanism for groups. This can lead to longer recovery periods at the end of each block, and increases the risk of losing Target nodes. Using group addresses for BLOB transfers will generally only pay off if the list of Target nodes is extensive, and the effectiveness of each addressing strategy will vary heavily between different deployments and the size of the chunks.

Transfer timeout If a Target node fails to respond to an acknowledged message within the BLOB Transfer Client's time limit, the Target node is dropped from the transfer. The application can reduce the chances of this by giving the BLOB Transfer Client extra time through the context structure. The extra time may be set in 10-second increments, up to 182 hours, in addition to the base time of 20 seconds. The wait time scales automatically with the transfer TTL.

Note that the BLOB Transfer Client only moves forward with the transfer in following cases:

- All Target nodes have responded.
- A node has been removed from the list of Target nodes.
- The BLOB Transfer Client times out.

Increasing the wait time will increase this delay.

BLOB transfer capabilities retrieval It is generally recommended to retrieve BLOB transfer capabilities before starting a transfer. The procedure populates the transfer capabilities from all Target nodes with the most liberal set of parameters that allows all Target nodes to participate in the transfer. Any Target nodes that fail to respond, or respond with incompatible transfer parameters, will be dropped.

Target nodes are prioritized according to their order in the list of Target nodes. If a Target node is found to be incompatible with any of the previous Target nodes, for instance by reporting a non-overlapping block size range, it will be dropped. Lost Target nodes will be reported through the `lost_target` callback.

The end of the procedure is signalled through the `caps` callback, and the resulting capabilities can be used to determine the block and chunk sizes required for the BLOB transfer.

BLOB transfer The BLOB transfer is started by calling `bt_mesh_blob_cli_send()` function, which (in addition to the aforementioned transfer inputs) requires a set of transfer parameters and a BLOB stream instance. The transfer parameters include the 64-bit BLOB ID, the BLOB size, the transfer mode, the block size in logarithmic representation and the chunk size. The BLOB ID is application defined, but must match the BLOB ID the BLOB Transfer Servers have been started with.

The transfer runs until it either completes successfully for at least one Target node, or it is cancelled. The end of the transfer is communicated to the application through the `end` callback. Lost Target nodes will be reported through the `lost_target` callback.

API reference

Bluetooth Mesh BLOB Transfer Client model API

The BLOB Transfer Client is instantiated on the sender node, and the BLOB Transfer Server is instantiated on the receiver nodes.

Concepts The BLOB transfer protocol introduces several new concepts to implement the BLOB transfer.

BLOBs BLOBs are binary objects up to 4 GB in size, that can contain any data the application would like to transfer through the mesh network. The BLOBs are continuous data objects, divided into blocks and chunks to make the transfers reliable and easy to process. No limitations are put on the contents or structure of the BLOB, and applications are free to define any encoding or compression they'd like on the data itself.

The BLOB transfer protocol does not provide any built-in integrity checks, encryption or authentication of the BLOB data. However, the underlying encryption of the Bluetooth Mesh protocol provides data integrity checks and protects the contents of the BLOB from third parties using network and application level encryption.

Blocks The binary objects are divided into blocks, typically from a few hundred to several thousand bytes in size. Each block is transmitted separately, and the BLOB Transfer Client ensures that all BLOB Transfer Servers have received the full block before moving on to the next. The block size is determined by the transfer's `block_size_log` parameter, and is the same for all blocks in the transfer except the last, which may be smaller. For a BLOB stored in flash memory, the block size is typically a multiple of the flash page size of the Target devices.

Chunks Each block is divided into chunks. A chunk is the smallest data unit in the BLOB transfer, and must fit inside a single Bluetooth Mesh access message excluding the opcode (379 bytes or less). The mechanism for transferring chunks depends on the transfer mode.

When operating in Push BLOB Transfer Mode, the chunks are sent as unacknowledged packets from the BLOB Transfer Client to all targeted BLOB Transfer Servers. Once all chunks in a block have been sent, the BLOB Transfer Client asks each BLOB Transfer Server if they're missing any chunks, and resends them. This is repeated until all BLOB Transfer Servers have received all chunks, or the BLOB Transfer Client gives up.

When operating in Pull BLOB Transfer Mode, the BLOB Transfer Server will request a small number of chunks from the BLOB Transfer Client at a time, and wait for the BLOB Transfer Client to send them before requesting more chunks. This repeats until all chunks have been transferred, or the BLOB Transfer Server gives up.

Read more about the transfer modes in *Transfer modes* section.

BLOB streams In the BLOB Transfer models' APIs, the BLOB data handling is separated from the high-level transfer handling. This split allows reuse of different BLOB storage and transfer strategies for different applications. While the high level transfer is controlled directly by the application, the BLOB data itself is accessed through a *BLOB stream*.

The BLOB stream is comparable to a standard library file stream. Through opening, closing, reading and writing, the BLOB Transfer model gets full access to the BLOB data, whether it's kept in flash, RAM, or on a peripheral. The BLOB stream is opened with an access mode (read or write) before it's used, and the BLOB Transfer models will move around inside the BLOB's data in blocks and chunks, using the BLOB stream as an interface.

Interaction Before the BLOB is read or written, the stream is opened by calling its `open` callback. When used with a BLOB Transfer Server, the BLOB stream is always opened in write mode, and when used with a BLOB Transfer Client, it's always opened in read mode.

For each block in the BLOB, the BLOB Transfer model starts by calling `block_start`. Then, depending on the access mode, the BLOB stream's `wr` or `rd` callback is called repeatedly to move data to or from the BLOB. When the model is done processing the block, it calls `block_end`. When the transfer is complete, the BLOB stream is closed by calling `close`.

Implementations The application may implement their own BLOB stream, or use the implementations provided by Zephyr:

BLOB Flash The BLOB Flash Readers and Writers implement BLOB reading to and writing from flash partitions defined in the *flash map*.

BLOB Flash Reader The BLOB Flash Reader interacts with the BLOB Transfer Client to read BLOB data directly from flash. It must be initialized by calling `bt_mesh_blob_flash_rd_init()` before being passed to the BLOB Transfer Client. Each BLOB Flash Reader only supports one transfer at the time.

BLOB Flash Writer The BLOB Flash Writer interacts with the BLOB Transfer Server to write BLOB data directly to flash. It must be initialized by calling `bt_mesh_blob_flash_rd_init()` before being passed to the BLOB Transfer Server. Each BLOB Flash Writer only supports one transfer at the time, and requires a block size that is a multiple of the flash page size. If a transfer is started with a block size lower than the flash page size, the transfer will be rejected.

The BLOB Flash Writer copies chunk data into a buffer to accommodate chunks that are unaligned with the flash write block size. The buffer data is padded with `0xff` if either the start or length of the chunk is unaligned.

API Reference

Bluetooth Mesh BLOB flash stream

Transfer capabilities Each BLOB Transfer Server may have different transfer capabilities. The transfer capabilities of each device are controlled through the following configuration options:

- `CONFIG_BT_MESH_BLOB_SIZE_MAX`
- `CONFIG_BT_MESH_BLOB_BLOCK_SIZE_MIN`
- `CONFIG_BT_MESH_BLOB_BLOCK_SIZE_MAX`
- `CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX`

The `CONFIG_BT_MESH_BLOB_CHUNK_COUNT_MAX` option is also used by the BLOB Transfer Client and affects memory consumption by the BLOB Transfer Client model structure.

To ensure that the transfer can be received by as many servers as possible, the BLOB Transfer Client can retrieve the capabilities of each BLOB Transfer Server before starting the transfer. The client will transfer the BLOB with the highest possible block and chunk size.

Transfer modes BLOBs can be transferred using two transfer modes, Push BLOB Transfer Mode and Pull BLOB Transfer Mode. In most cases, the transfer should be conducted in Push BLOB Transfer Mode.

In Push BLOB Transfer Mode, the send rate is controlled by the BLOB Transfer Client, which will push all the chunks of each block without any high level flow control. Push BLOB Transfer Mode supports any number of Target nodes, and should be the default transfer mode.

In Pull BLOB Transfer Mode, the BLOB Transfer Server will "pull" the chunks from the BLOB Transfer Client at its own rate. Pull BLOB Transfer Mode can be conducted with multiple Target nodes, and is intended for transferring BLOBs to Target nodes acting as *Low Power Node*. When operating in Pull BLOB Transfer Mode, the BLOB Transfer Server will request chunks from the BLOB Transfer Client in small batches, and wait for them all to arrive before requesting more chunks. This process is repeated until the BLOB Transfer Server has received all chunks in a block. Then, the BLOB Transfer Client starts the next block, and the BLOB Transfer Server requests all chunks of that block.

Transfer timeout The timeout of the BLOB transfer is based on a Timeout Base value. Both client and server use the same Timeout Base value, but they calculate timeout differently.

The BLOB Transfer Server uses the following formula to calculate the BLOB transfer timeout:

```
10 * (Timeout Base + 1) seconds
```

For the BLOB Transfer Client, the following formula is used:

```
(10000 * (Timeout Base + 2)) + (100 * TTL) milliseconds
```

where TTL is time to live value set in the transfer.

API reference This section contains types and defines common to the BLOB Transfer models.

Bluetooth Mesh BLOB model API

Device Firmware Update (DFU) Bluetooth Mesh supports the distribution of firmware images across a mesh network. The Bluetooth mesh DFU subsystem implements the Bluetooth Mesh Device Firmware Update Model specification version 1.0.

Bluetooth Mesh DFU implements a distribution mechanism for firmware images, and does not put any restrictions on the size, format or usage of the images. The primary design goal of the subsystem is to provide the qualifiable parts of the Bluetooth Mesh DFU specification, and leave the usage, firmware validation and deployment to the application.

The DFU specification is implemented in the Zephyr Bluetooth Mesh DFU subsystem as three separate models:

Firmware Update Server The Firmware Update Server model implements the Target node functionality of the *Device Firmware Update (DFU)* subsystem. It extends the *BLOB Transfer Server*, which it uses to receive the firmware image binary from the Distributor node.

Together with the extended BLOB Transfer Server model, the Firmware Update Server model implements all the required functionality for receiving firmware updates over the mesh network, but does not provide any functionality for storing, applying or verifying the images.

Firmware images The Firmware Update Server holds a list of all the updatable firmware images on the device. The full list shall be passed to the server through the `_imgs` parameter in `BT_MESH_DFU_SRV_INIT`, and must be populated before the Bluetooth Mesh subsystem is started. Each firmware image in the image list must be independently updatable, and should have its own firmware ID.

For instance, a device with an upgradable bootloader, an application and a peripheral chip with firmware update capabilities could have three entries in the firmware image list, each with their own separate firmware ID.

Receiving transfers The Firmware Update Server model uses a BLOB Transfer Server model on the same element to transfer the binary image. The interaction between the Firmware Update Server, BLOB Transfer Server and application is described below:

Transfer check The transfer check is an optional pre-transfer check the application can perform on incoming firmware image metadata. The Firmware Update Server performs the transfer check by calling the `check` callback.

The result of the transfer check is a pass/fail status return and the expected `bt_mesh_dfu_effect`. The DFU effect return parameter will be communicated back to the Distributor, and should indicate what effect the firmware update will have on the mesh state of the device.

Composition Data and Models Metadata If the transfer will cause the device to change its Composition Data or become unprovisioned, this should be communicated through the effect parameter of the metadata check.

When the transfer will cause the Composition Data to change, and the *Remote Provisioning Server* is supported, the Composition Data of the new firmware image will be represented by Composition Data Pages 128, 129, and 130. The Models Metadata of the new firmware image will be represented by Models Metadata Page 128. Composition Data Pages 0, 1 and 2, and Models Metadata Page 0, will represent the Composition Data and the Models Metadata of the old firmware image until the device is reprovisioned with Node Provisioning Protocol Interface (NPPI) procedures using the *Remote Provisioning Client*.

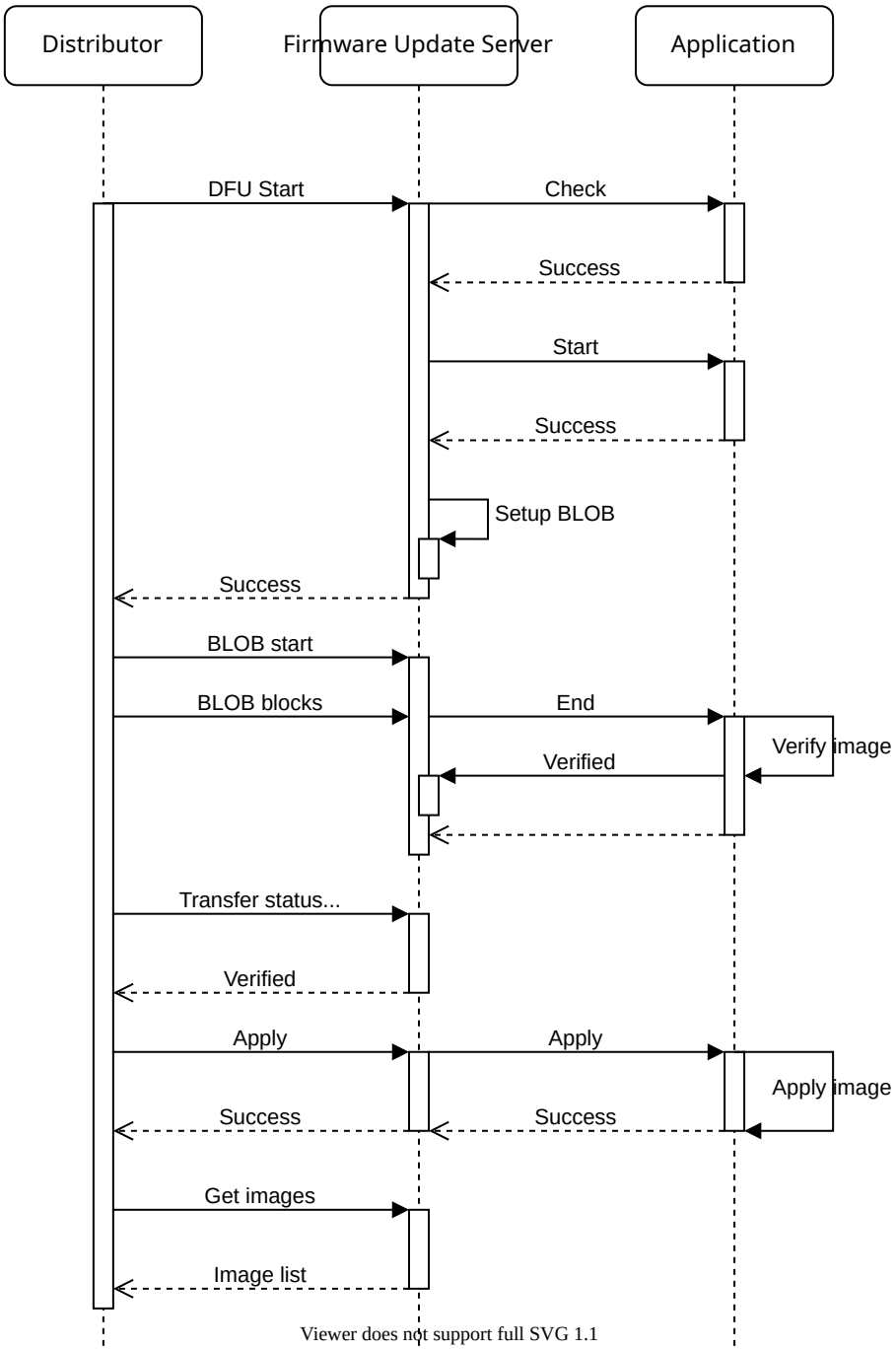


Fig. 20: Bluetooth Mesh Firmware Update Server transfer

The application must call functions `bt_mesh_comp_change_prepare()` and `bt_mesh_models_metadata_change_prepare()` to store the existing Composition Data and Models Metadata pages before booting into the firmware with the updated Composition Data and Models Metadata. The old Composition Data will then be loaded into Composition Data Pages 0, 1 and 2, while the Composition Data in the new firmware will be loaded into Composition Data Pages 128, 129 and 130. The Models Metadata for the old image will be loaded into Models Metadata Page 0, and the Models Metadata for the new image will be loaded into Models Metadata Page 128.

Limitation:

- It is not possible to change the Composition Data of the device and keep the device provisioned and working with the old firmware after the new firmware image is applied.

Start The Start procedure prepares the application for the incoming transfer. It'll contain information about which image is being updated, as well as the update metadata.

The Firmware Update Server `start` callback must return a pointer to the BLOB Writer the BLOB Transfer Server will send the BLOB to.

BLOB transfer After the setup stage, the Firmware Update Server prepares the BLOB Transfer Server for the incoming transfer. The entire firmware image is transferred to the BLOB Transfer Server, which passes the image to its assigned BLOB Writer.

At the end of the BLOB transfer, the Firmware Update Server calls its `end` callback.

Image verification After the BLOB transfer has finished, the application should verify the image in any way it can to ensure that it is ready for being applied. Once the image has been verified, the application calls `bt_mesh_dfu_srv_verified()`.

If the image can't be verified, the application calls `bt_mesh_dfu_srv_rejected()`.

Applying the image Finally, if the image was verified, the Distributor may instruct the Firmware Update Server to apply the transfer. This is communicated to the application through the `apply` callback. The application should swap the image and start running with the new firmware. The firmware image table should be updated to reflect the new firmware ID of the updated image.

When the transfer applies to the mesh application itself, the device might have to reboot as part of the swap. This restart can be performed from inside the apply callback, or done asynchronously. After booting up with the new firmware, the firmware image table should be updated before the Bluetooth Mesh subsystem is started.

The Distributor will read out the firmware image table to confirm that the transfer was successfully applied. If the metadata check indicated that the device would become unprovisioned, the Target node is not required to respond to this check.

API reference
Firmware Update Server model

Firmware Update Client The Firmware Update Client is responsible for distributing firmware updates through the mesh network. The Firmware Update Client uses the *BLOB Transfer Client* as a transport for its transfers.

API reference
Firmware Uppdate Client model

Firmware Distribution Server The Firmware Distribution Server model implements the Distributor role for the *Device Firmware Update (DFU)* subsystem. It extends the *BLOB Transfer Server*, which it uses to receive the firmware image binary from the Initiator node. It also instantiates a *Firmware Update Client*, which it uses to distribute firmware updates throughout the mesh network.

> **Note**
>
> Currently, the Firmware Distribution Server supports out-of-band (OOB) retrieval of firmware images over SMP service only.

The Firmware Distribution Server does not have an API of its own, but relies on a Firmware Distribution Client model on a different device to give it information and trigger image distribution and upload.

Firmware slots The Firmware Distribution Server is capable of storing multiple firmware images for distribution. Each slot contains a separate firmware image with metadata, and can be distributed to other mesh nodes in the network in any order. The contents, format and size of the firmware images are vendor specific, and may contain data from other vendors. The application should never attempt to execute or modify them.

The slots are managed remotely by a Firmware Distribution Client, which can both upload new slots and delete old ones. The application is notified of changes to the slots through the Firmware Distribution Server's callbacks (`bt_mesh_fd_srv_cb`). While the metadata for each firmware slot is stored internally, the application must provide a *BLOB streams* for reading and writing the firmware image.

API reference

Firmware Distribution Server model

Overview

DFU roles The Bluetooth Mesh DFU subsystem defines three different roles the mesh nodes have to assume in the distribution of firmware images:

Target node
: Target node is the receiver and user of the transferred firmware images. All its functionality is implemented by the *Firmware Update Server* model. A transfer may be targeting any number of Target nodes, and they will all be updated concurrently.

Distributor
: The Distributor role serves two purposes in the DFU process. First, it's acting as the Target node in the Upload Firmware procedure, then it distributes the uploaded image to other Target nodes as the Distributor. The Distributor does not select the parameters of the transfer, but relies on an Initiator to give it a list of Target nodes and transfer parameters. The Distributor functionality is implemented in two models, *Firmware Distribution Server* and *Firmware Update Client*. The *Firmware Distribution Server* is responsible for communicating with the Initiator, and the *Firmware Update Client* is responsible for distributing the image to the Target nodes.

Initiator
: The Initiator role is typically implemented by the same device that implements the Bluetooth Mesh *Provisioner* and *Configurator* roles. The Initiator needs a full overview of the potential Target nodes and their firmware, and will control (and initiate) all firmware updates. The Initiator role is not implemented in the Zephyr Bluetooth Mesh DFU subsystem.

Bluetooth Mesh applications may combine the DFU roles in any way they'd like, and even take on multiple instances of the same role by instantiating the models on separate elements. For

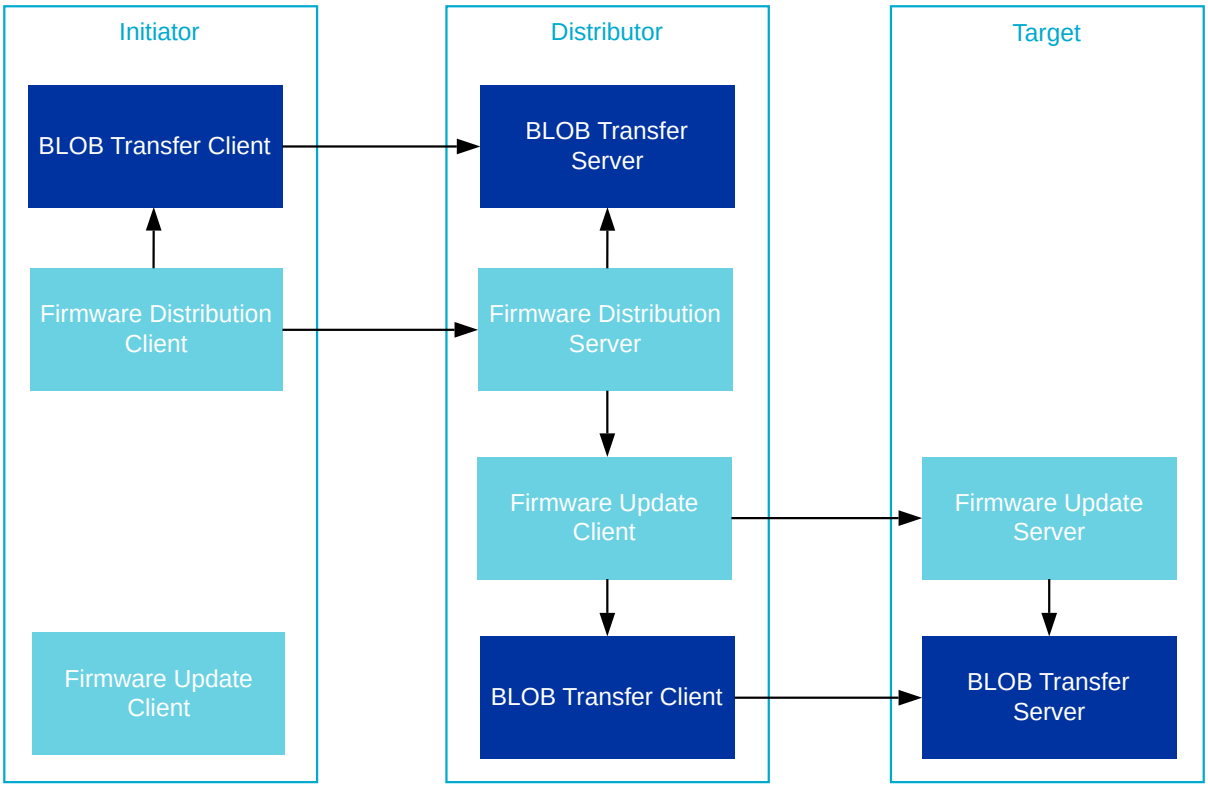


Fig. 21: DFU roles and the associated Bluetooth Mesh models

instance, the Distributor and Initiator role can be combined by instantiating the *Firmware Update Client* on the Initiator node and calling its API directly.

It's also possible to combine the Initiator and Distributor devices into a single device, and replace the Firmware Distribution Server model with a proprietary mechanism that will access the Firmware Update Client model directly, e.g. over a serial protocol.

> **Note**
>
> All DFU models instantiate one or more *BLOB Transfer models*, and may need to be spread over multiple elements for certain role combinations.

Stages The Bluetooth Mesh DFU process is designed to act in three stages:

Upload stage
First, the image is uploaded to a Distributor in a mesh network by an external entity, such as a phone or gateway (the Initiator). During the Upload stage, the Initiator transfers the firmware image and all its metadata to the Distributor node inside the mesh network. The Distributor stores the firmware image and its metadata persistently, and awaits further instructions from the Initiator. The time required to complete the upload process depends on the size of the image. After the upload completes, the Initiator can disconnect from the network during the much more time-consuming Distribution stage. Once the firmware has been uploaded to the Distributor, the Initiator may trigger the Distribution stage at any time.

Firmware Capability Check stage (optional)
Before starting the Distribution stage, the Initiator may optionally check if Target nodes can accept the new firmware. Nodes that do not respond, or respond that they can't receive the new firmware, are excluded from the firmware distribution process.

Distribution stage
Before the firmware image can be distributed, the Initiator transfers the list of Target nodes and their designated firmware image index to the Distributor. Next, it tells the Distributor to start the firmware distributon process, which runs in the background while the Initiator and the mesh network perform other duties. Once the firmware image has been transferred to the Target nodes, the Distributor may ask them to apply the firmware image immediately and report back with their status and new firmware IDs.

Firmware images All updatable parts of a mesh node's firmware should be represented as a firmware image. Each Target node holds a list of firmware images, each of which should be independently updatable and identifiable.

Firmware images are represented as a BLOB (the firmware itself) with the following additional information attached to it:

Firmware ID
The firmware ID is used to identify a firmware image. The Initiator node may ask the Target nodes for a list of its current firmware IDs to determine whether a newer version of the firmware is available. The format of the firmware ID is vendor specific, but generally, it should include enough information for an Initiator node with knowledge of the format to determine the type of image as well as its version. The firmware ID is optional, and its max length is determined by `CONFIG_BT_MESH_DFU_FWID_MAXLEN`.

Firmware metadata
The firmware metadata is used by the Target node to determine whether it should accept an incoming firmware update, and what the effect of the update would be. The metadata format is vendor specific, and should contain all information the Target node needs to verify the image, as well as any preparation the Target node has to make before the image is applied. Typical metadata information can be image signatures, changes to the node's Composition Data and the format of the BLOB. The Target node may perform a metadata check before accepting incoming transfers to determine whether the transfer should be started. The firmware metadata can be discarded by the Target node after the metadata check, as other nodes will never request the metadata from the Target node. The firmware metadata is optional, and its maximum length is determined by `CONFIG_BT_MESH_DFU_METADATA_MAXLEN`.

The Bluetooth Mesh DFU subsystem in Zephyr provides its own metadata format (`bt_mesh_dfu_metadata`) together with a set of related functions that can be used by an end product. The support for it is enabled using the `CONFIG_BT_MESH_DFU_METADATA` option. The format of the metadata is presented in the table below.

Field	Size (Bytes)	Description
New firmware version	8 B	1 B: Major version 1 B: Minor version 2 B: Revision 4 B: Build number
New firmware size	3 B	Size in bytes for a new firmware
New firmware core type	1 B	Bit field: Bit 0: Application core Bit 1: Network core Bit 2: Applications specific BLOB. Other bits: RFU
Hash of incoming composition data	4 B (Optional)	Lower 4 octets of AES-CMAC (app-specific-key, composition data). This field is present, if Bit 0 is set in the New firmware core type field.
New number of elements	2 B (Optional)	Number of elements on the node after firmware is applied. This field is present, if Bit 0 is set in the New firmware core type field.
Application-specific data for new firmware	<variable> (Optional)	Application-specific data to allow application to execute some vendor-specific behaviors using this data before it can respond with a status message.

> **Note**
>
> The AES-CMAC algorithm serves as a hashing function with a fixed key and is not used for encryption in Bluetooth Mesh DFU metadata. The resulting hash is not secure since the key is known.

Firmware URI
The firmware URI gives the Initiator information about where firmware updates for the image can be found. The URI points to an online resource the Initiator can interact with to get new versions of the firmware. This allows Initiators to perform updates for any node in the mesh network by interacting with the web server pointed to in the URI. The URI must point to a resource using the `http` or `https` schemes, and the targeted web server must behave according to the Firmware Check Over HTTPS procedure defined by the specification. The firmware URI is optional, and its max length is determined by `CONFIG_BT_MESH_DFU_URI_MAXLEN`.

> **Note**
>
> The out-of-band distribution mechanism is not supported.

Firmware effect A new image may have the Composition Data Page 0 different from the one allocated on a Target node. This may have an effect on the provisioning data of the node and how the Distributor finalizes the DFU. Depending on the availability of the Remote Provisioning Server model on the old and new image, the device may either boot up unprovisioned after applying the new firmware or require to be re-provisioned. The complete list of available options is defined in `bt_mesh_dfu_effect`:

`BT_MESH_DFU_EFFECT_NONE`
The device stays provisioned after the new firmware is programmed. This effect is chosen if the composition data of the new firmware doesn't change.

`BT_MESH_DFU_EFFECT_COMP_CHANGE_NO_RPR`
This effect is chosen when the composition data changes and the device doesn't support the remote provisioning. The new composition data takes place only after re-provisioning.

`BT_MESH_DFU_EFFECT_COMP_CHANGE`
This effect is chosen when the composition data changes and the device supports the remote provisioning. In this case, the device stays provisioned and the new composition data takes place after re-provisioning using the Remote Provisioning models.

`BT_MESH_DFU_EFFECT_UNPROV`
This effect is chosen if the composition data in the new firmware changes, the device does not support the remote provisioning, and the new composition data takes effect after applying the firmware. The effect can also be chosen, if it is necessary to unprovision the device for application-specific reasons.

When the Target node receives the Firmware Update Firmware Metadata Check message, the Firmware Update Server model calls the `bt_mesh_dfu_srv_cb.check` callback, the application can then process the metadata and provide the effect value. If the effect is `BT_MESH_DFU_EFFECT_COMP_CHANGE`, the application must call functions `bt_mesh_comp_change_prepare()` and `bt_mesh_models_metadata_change_prepare()` to prepare the Composition Data Page and Models Metadata Page contents before applying the new firmware image. See *Composition Data and Models Metadata* for more information.

DFU procedures The DFU protocol is implemented as a set of procedures that must be performed in a certain order.

The Initiator controls the Upload stage of the DFU protocol, and all Distributor side handling of the upload subprocedures is implemented in the *Firmware Distribution Server*.

The Distribution stage is controlled by the Distributor, as implemented by the *Firmware Update Client*. The Target node implements all handling of these procedures in the *Firmware Update Server*, and notifies the application through a set of callbacks.

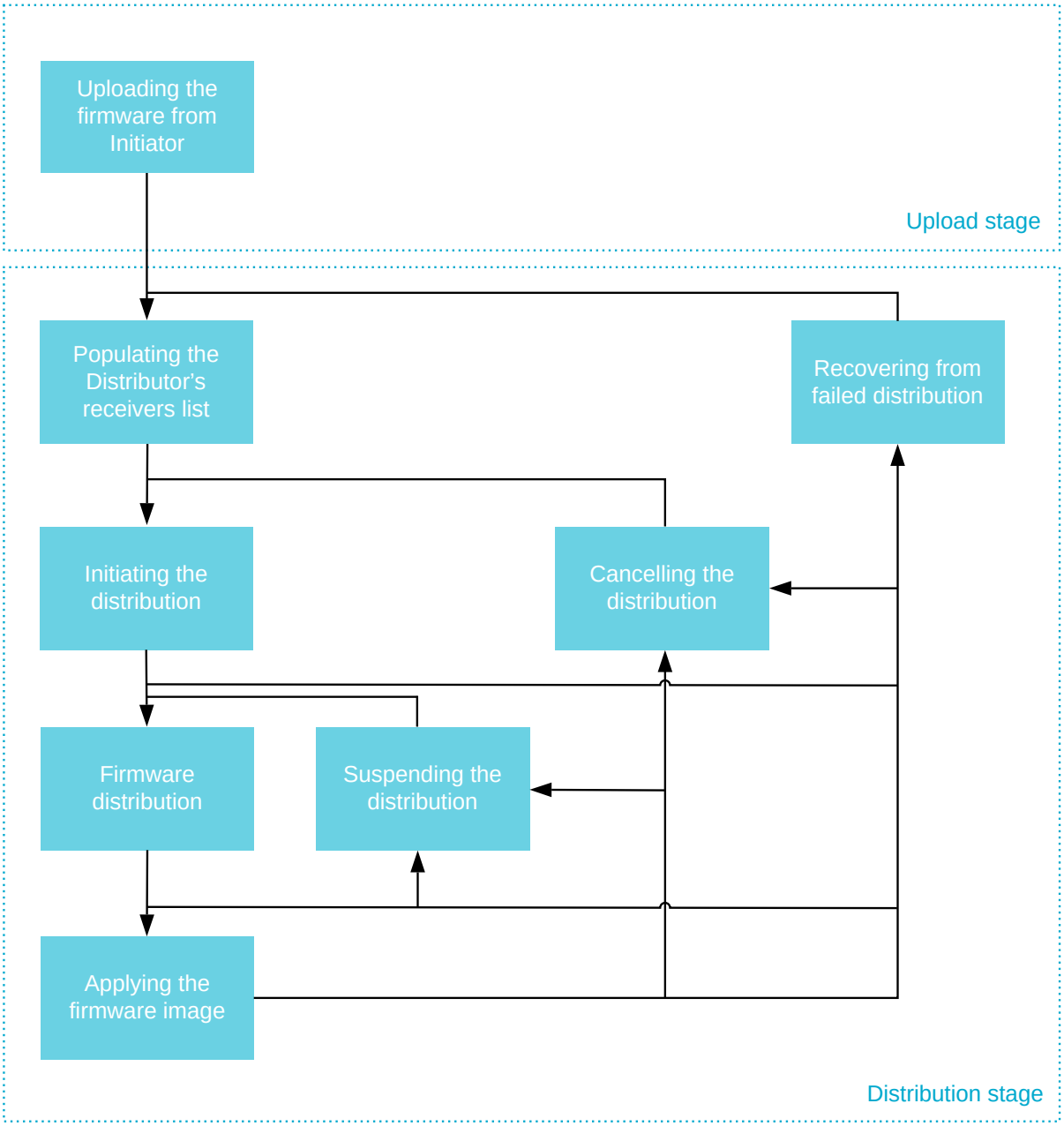


Fig. 22: DFU stages and procedures as seen from the Distributor

Uploading the firmware The Upload Firmware procedure uses the *BLOB Transfer models* to transfer the firmware image from the Initiator to the Distributor. The Upload Firmware procedure works in two steps:

1. The Initiator generates a BLOB ID, and sends it to the Distributor's Firmware Distribution Server along with the firmware information and other input parameters of the BLOB transfer. The Firmware Distribution Server stores the information, and prepares its BLOB Transfer Server for the incoming transfer before it responds with a status message to the Initiator.

2. The Initiator's BLOB Transfer Client model transfers the firmware image to the Distributor's BLOB Transfer Server, which stores the image in a predetermined flash partition.

When the BLOB transfer finishes, the firmware image is ready for distribution. The Initiator may upload several firmware images to the Distributor, and ask it to distribute them in any order or at any time. Additional procedures are available for querying and deleting firmware images from the Distributor.

The following Distributor's capabilities related to firmware images can be configured using the configuration options:

- `CONFIG_BT_MESH_DFU_SLOT_CNT`: Amount of image slots available on the device.
- `CONFIG_BT_MESH_DFD_SRV_SLOT_MAX_SIZE`: Maximum allowed size for each image.
- `CONFIG_BT_MESH_DFD_SRV_SLOT_SPACE`: Available space for all images.

Populating the Distributor's receivers list Before the Distributor can start distributing the firmware image, it needs a list of Target nodes to send the image to. The Initiator gets the full list of Target nodes either by querying the potential targets directly, or through some external authority. The Initiator uses this information to populate the Distributor's receivers list with the address and relevant firmware image index of each Target node. The Initiator may send one or more Firmware Distribution Receivers Add messages to build the Distributor's receivers list, and a Firmware Distribution Receivers Delete All message to clear it.

The maximum number of receivers that can be added to the Distributor is configured through the `CONFIG_BT_MESH_DFD_SRV_TARGETS_MAX` configuration option.

Initiating the distribution Once the Distributor has stored a firmware image and received a list of Target nodes, the Initiator may initiate the distribution procedure. The BLOB transfer parameters for the distribution are passed to the Distributor along with an update policy. The update policy decides whether the Distributor should request that the firmware is applied on the Target nodes or not. The Distributor stores the transfer parameters and starts distributing the firmware image to its list of Target nodes.

Firmware distribution The Distributor's Firmware Update Client model uses its BLOB Transfer Client model's broadcast subsystem to communicate with all Target nodes. The firmware distribution is performed with the following steps:

1. The Distributor's Firmware Update Client model generates a BLOB ID and sends it to each Target node's Firmware Update Server model, along with the other BLOB transfer parameters, the Target node firmware image index and the firmware image metadata. Each Target node performs a metadata check and prepares their BLOB Transfer Server model for the transfer, before sending a status response to the Firmware Update Client, indicating if the firmware update will have any effect on the Bluetooth Mesh state of the node.
2. The Distributor's BLOB Transfer Client model transfers the firmware image to all Target nodes.
3. Once the BLOB transfer has been received, the Target nodes' applications verify that the firmware is valid by performing checks such as signature verification or image checksums against the image metadata.
4. The Distributor's Firmware Update Client model queries all Target nodes to ensure that they've all verified the firmware image.

If the distribution procedure completed with at least one Target node reporting that the image has been received and verified, the distribution procedure is considered successful.

> **Note**
>
> The firmware distribution procedure only fails if *all* Target nodes are lost. It is up to the Initiator to request a list of failed Target nodes from the Distributor and initiate additional attempts to update the lost Target nodes after the current attempt is finished.

Suspending the distribution The Initiator can also request the Distributor to suspend the firmware distribution. In this case, the Distributor will stop sending any messages to Target nodes. When the firmware distribution is resumed, the Distributor will continue sending the firmware from the last successfully transferred block.

Applying the firmware image If the Initiator requested it, the Distributor can initiate the Apply Firmware on Target Node procedure on all Target nodes that successfully received and verified the firmware image. The Apply Firmware on Target Node procedure takes no parameters, and to avoid ambiguity, it should be performed before a new transfer is initiated. The Apply Firmware on Target Node procedure consists of the following steps:

1. The Distributor's Firmware Update Client model instructs all Target nodes that have verified the firmware image to apply it. The Target nodes' Firmware Update Server models respond with a status message before calling their application's `apply` callback.
2. The Target node's application performs any preparations needed before applying the transfer, such as storing a snapshot of the Composition Data or clearing its configuration.
3. The Target node's application swaps the current firmware with the new image and updates its firmware image list with the new firmware ID.
4. The Distributor's Firmware Update Client model requests the full list of firmware images from each Target node, and scans through the list to make sure that the new firmware ID has replaced the old.

> **Note**
>
> During the metadata check in the distribution procedure, the Target node may have reported that it will become unprovisioned after the firmware image is applied. In this case, the Distributor's Firmware Update Client model will send a request for the full firmware image list, and expect no response.

Cancelling the distribution The firmware distribution can be cancelled at any time by the Initiator. In this case, the Distributor starts the cancelling procedure by sending a cancelling message to all Target nodes. The Distributor waits for the response from all Target nodes. Once all Target nodes have replied, or the request has timed out, the distribution procedure is cancelled. After this the distribution procedure can be started again from the `Firmware distribution` section.

API reference This section lists the types common to the Device Firmware Update mesh models.

Firmware Distribution models

Bluetooth Mesh Device Firmware Update

Bluetooth Mesh Device Firmware Update (DFU) metadata

Message The Bluetooth Mesh message provides set of structures, macros and functions used for preparing message buffers, managing message and acknowledged message contexts.

API reference
Message

Segmentation and reassembly (SAR) Segmentation and reassembly (SAR) provides a way of handling larger upper transport layer messages in a mesh network, with a purpose of enhancing the Bluetooth Mesh throughput. The segmentation and reassembly mechanism is used by the lower transport layer.

The lower transport layer defines how the upper transport layer PDUs are segmented and reassembled into multiple Lower Transport PDUs, and sends them to the lower transport layer on a peer device. If the Upper Transport PDU fits, it is sent in a single Lower Transport PDU. For longer packets, which do not fit into a single Lower Transport PDU, the lower transport layer performs segmentation, splitting the Upper Transport PDU into multiple segments.

The lower transport layer on the receiving device reassembles the segments into a single Upper Transport PDU before passing it up the stack. Delivery of a segmented message is acknowledged by the lower transport layer of the receiving node, while an unsegmented message delivery is not acknowledged. However, an Upper Transport PDU that fits into one Lower Transport PDU can also be sent as a single-segment segmented message when acknowledgment by the lower transport layer is required. Set the `send rel` flag (see `bt_mesh_msg_ctx`) to use the reliable message transmission and acknowledge single-segment segmented messages.

The transport layer is able to transport up to 32 segments with its SAR mechanism, with a maximum message (PDU) size of 384 octets. To configure message size for the Bluetooth Mesh stack, use the following Kconfig options:

- `CONFIG_BT_MESH_RX_SEG_MAX` to set the maximum number of segments in an incoming message.
- `CONFIG_BT_MESH_TX_SEG_MAX` to set the maximum number of segments in an outgoing message.

The Kconfig options `CONFIG_BT_MESH_TX_SEG_MSG_COUNT` and `CONFIG_BT_MESH_RX_SEG_MSG_COUNT` define how many outgoing and incoming segmented messages can be processed simultaneously. When more than one segmented message is sent to the same destination, the messages are queued and sent one at a time.

Incoming and outgoing segmented messages share the same pool for allocation of their segments. This pool size is configured through the `CONFIG_BT_MESH_SEG_BUFS` Kconfig option. Both incoming and outgoing messages allocate segments at the start of the transaction. The outgoing segmented message releases its segments one by one as soon as they are acknowledged by the receiver, while the incoming message releases the segments first after the message is fully received. Keep this in mind when defining the size of the buffers.

SAR does not impose extra overhead on the access layer payload per segment.

Segmentation and reassembly (SAR) Configuration models With Bluetooth Mesh Protocol Specification version 1.1, it became possible to configure SAR behavior, such as intervals, timers and retransmission counters, over a mesh network using SAR Configuration models:

- *SAR Configuration Client*
- *SAR Configuration Server*

The following SAR behavior applies regardless of the presence of a SAR Configuration Server on a node.

Transmission of segments is separated by a segment transmission interval (see the *SAR Segment Interval Step* state). Other configurable time intervals and delays available for the segmentation and reassembly are:

- Interval between unicast retransmissions (see the states *SAR Unicast Retransmissions Interval Step* and *SAR Unicast Retransmissions Interval Increment*).

- Interval between multicast retransmissions (see the *SAR Multicast Retransmissions Interval Step* state).
- Segment reception interval (see the *SAR Receiver Segment Interval Step* state).
- Acknowledgment delay increment (see the *SAR Acknowledgment Delay Increment* state).

When the last segment marked as unacknowledged is transmitted, the lower transport layer starts a retransmissions timer. The initial value of the SAR Unicast Retransmissions timer depends on the value of the TTL field of the message. If the TTL field value is greater than `0`, the initial value for the timer is set according to the following formula:

$$unicast\ retransmissions\ interval\ step + unicast\ retransmissions\ interval\ increment \times (TTL - 1)$$

If the TTL field value is `0`, the initial value of the timer is set to the unicast retransmissions interval step.

The initial value of the SAR Multicast Retransmissions timer is set to the multicast retransmissions interval.

When the lower transport layer receives a message segment, it starts a SAR Discard timer. The discard timer tells how long the lower transport layer waits before discarding the segmented message the segment belongs to. The initial value of the SAR Discard timer is the discard timeout value indicated by the *SAR Discard Timeout* state.

SAR Acknowledgment timer holds the time before a Segment Acknowledgment message is sent for a received segment. The initial value of the SAR Acknowledgment timer is calculated using the following formula:

$$min(SegN + 0.5, acknowledgment\ delay\ increment) \times segment\ reception\ interval$$

The `SegN` field value identifies the total number of segments the Upper Transport PDU is segmented into.

Four counters are related to SAR behavior:

- Two unicast retransmissions counts (see *SAR Unicast Retransmissions Count* state and *SAR Unicast Retransmissions Without Progress Count* state)
- Multicast retransmissions count (see *SAR Multicast Retransmissions Count* state)
- Acknowledgment retransmissions count (see *SAR Acknowledgment Retransmissions Count* state)

If the number of segments in the transmission is higher than the value of the *SAR Segments Threshold* state, Segment Acknowledgment messages are retransmitted using the value of the *SAR Acknowledgment Retransmissions Count* state.

SAR states There are two states defined related to segmentation and reassembly:

- SAR Transmitter state
- SAR Receiver state

The SAR Transmitter state is a composite state that controls the number and timing of transmissions of segmented messages. It includes the following states:

- SAR Segment Interval Step
- SAR Unicast Retransmissions Count
- SAR Unicast Retransmissions Without Progress Count
- SAR Unicast Retransmissions Interval Step
- SAR Unicast Retransmissions Interval Increment
- SAR Multicast Retransmissions Count

- SAR Multicast Retransmissions Interval Step

The SAR Receiver state is a composite state that controls the number and timing of Segment Acknowledgment transmissions and the discarding of reassembly of a segmented message. It includes the following states:

- SAR Segments Threshold
- SAR Discard Timeout
- SAR Acknowledgment Delay Increment
- SAR Acknowledgment Retransmissions Count
- SAR Receiver Segment Interval Step

SAR Segment Interval Step SAR Segment Interval Step state holds a value that controls the interval between transmissions of segments of a segmented message. The interval is measured in milliseconds.

Use the `CONFIG_BT_MESH_SAR_TX_SEG_INT_STEP` Kconfig option to set the default value. Segment transmission interval is then calculated using the following formula:

$$(\texttt{CONFIG_BT_MESH_SAR_TX_SEG_INT_STEP} + 1) \times 10 \text{ ms}$$

SAR Unicast Retransmissions Count SAR Unicast Retransmissions Count holds a value that defines the maximum number of retransmissions of a segmented message to a unicast destination. Use the `CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_COUNT` Kconfig option to set the default value for this state.

SAR Unicast Retransmissions Without Progress Count This state holds a value that defines the maximum number of retransmissions of a segmented message to a unicast address that will be sent if no acknowledgment was received during the timeout, or if an acknowledgment with already confirmed segments was received. Use the Kconfig option `CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_WITHOUT_PROG_COUNT` to set the maximum number of retransmissions.

SAR Unicast Retransmissions Interval Step The value of this state controls the interval step used for delaying the retransmissions of unacknowledged segments of a segmented message to a unicast address. The interval step is measured in milliseconds.

Use the `CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_INT_STEP` Kconfig option to set the default value. This value is then used to calculate the interval step using the following formula:

$$(\texttt{CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_INT_STEP} + 1) \times 25 \text{ ms}$$

SAR Unicast Retransmissions Interval Increment SAR Unicast Retransmissions Interval Increment holds a value that controls the interval increment used for delaying the retransmissions of unacknowledged segments of a segmented message to a unicast address. The increment is measured in milliseconds.

Use the Kconfig option `CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_INT_INC` to set the default value. The Kconfig option value is used to calculate the increment using the following formula:

$$(\texttt{CONFIG_BT_MESH_SAR_TX_UNICAST_RETRANS_INT_INC} + 1) \times 25 \text{ ms}$$

SAR Multicast Retransmissions Count The state holds a value that controls the total number of retransmissions of a segmented message to a multicast address. Use the Kconfig option `CONFIG_BT_MESH_SAR_TX_MULTICAST_RETRANS_COUNT` to set the total number of retransmissions.

SAR Multicast Retransmissions Interval Step This state holds a value that controls the interval between retransmissions of all segments in a segmented message to a multicast address. The interval is measured in milliseconds.

Use the Kconfig option `CONFIG_BT_MESH_SAR_TX_MULTICAST_RETRANS_INT` to set the default value that is used to calculate the interval using the following formula:

$$(\texttt{CONFIG_BT_MESH_SAR_TX_MULTICAST_RETRANS_INT} + 1) \times 25 \text{ ms}$$

SAR Discard Timeout The value of this state defines the time in seconds that the lower transport layer waits after receiving segments of a segmented message before discarding that segmented message. Use the Kconfig option `CONFIG_BT_MESH_SAR_RX_DISCARD_TIMEOUT` to set the default value. The discard timeout will be calculated using the following formula:

$$(\texttt{CONFIG_BT_MESH_SAR_RX_DISCARD_TIMEOUT} + 1) \times 5 \text{ seconds}$$

SAR Acknowledgment Delay Increment This state holds a value that controls the delay increment of an interval used for delaying the transmission of an acknowledgment message after receiving a new segment. The increment is measured in segments.

Use the Kconfig option `CONFIG_BT_MESH_SAR_RX_ACK_DELAY_INC` to set the default value. The increment value is calculated to be `CONFIG_BT_MESH_SAR_RX_ACK_DELAY_INC` $+ 1.5$.

SAR Segments Threshold SAR Segments Threshold state holds a value that defines a threshold in number of segments of a segmented message for acknowledgment retransmissions. Use the Kconfig option `CONFIG_BT_MESH_SAR_RX_SEG_THRESHOLD` to set the threshold.

When the number of segments of a segmented message is above this threshold, the stack will additionally retransmit every acknowledgment message the number of times given by the value of `CONFIG_BT_MESH_SAR_RX_ACK_RETRANS_COUNT`.

SAR Acknowledgment Retransmissions Count The SAR Acknowledgment Retransmissions Count state controls the number of retransmissions of Segment Acknowledgment messages sent by the lower transport layer. It gives the total number of retranmissions of an acknowledgment message that the stack will additionally send when the size of segments in a segmented message is above the `CONFIG_BT_MESH_SAR_RX_SEG_THRESHOLD` value.

Use the Kconfig option `CONFIG_BT_MESH_SAR_RX_ACK_RETRANS_COUNT` to set the default value for this state. The maximum number of transmissions of a Segment Acknowledgment message is `CONFIG_BT_MESH_SAR_RX_ACK_RETRANS_COUNT` $+ 1$.

SAR Receiver Segment Interval Step The SAR Receiver Segment Interval Step defines the segments reception interval step used for delaying the transmission of an acknowledgment message after receiving a new segment. The interval is measured in milliseconds.

Use the Kconfig option `CONFIG_BT_MESH_SAR_RX_SEG_INT_STEP` to set the default value and calculate the interval using the following formula:

$$(\texttt{CONFIG_BT_MESH_SAR_RX_SEG_INT_STEP} + 1) \times 10 \text{ ms}$$

Provisioning Provisioning is the process of adding devices to a mesh network. It requires two devices operating in the following roles:

- The *provisioner* represents the network owner, and is responsible for adding new nodes to the mesh network.
- The *provisionee* is the device that gets added to the network through the Provisioning process. Before the provisioning process starts, the provisionee is an *unprovisioned device*.

The Provisioning module in the Zephyr Bluetooth Mesh stack supports both the Advertising and GATT Provisioning bearers for the provisionee role, as well as the Advertising Provisioning bearer for the provisioner role.

The Provisioning process All Bluetooth Mesh nodes must be provisioned before they can participate in a Bluetooth Mesh network. The Provisioning API provides all the functionality necessary for a device to become a provisioned mesh node. Provisioning is a five-step process, involving the following steps:

- Beaconing
- Invitation
- Public key exchange
- Authentication
- Provisioning data transfer

Beaconing To start the provisioning process, the unprovisioned device must first start broadcasting the Unprovisioned Beacon. This makes it visible to nearby provisioners, which can initiate the provisioning. To indicate that the device needs to be provisioned, call `bt_mesh_prov_enable()`. The device starts broadcasting the Unprovisioned Beacon with the device UUID and the `OOB information` field, as specified in the `prov` parameter passed to `bt_mesh_init()`. Additionally, a Uniform Resource Identifier (URI) may be specified, which can point the provisioner to the location of some Out Of Band information, such as the device's public key or an authentication value database. The URI is advertised in a separate beacon, with a URI hash included in the unprovisioned beacon, to tie the two together.

Uniform Resource Identifier The Uniform Resource Identifier shall follow the format specified in the Bluetooth Core Specification Supplement. The URI must start with a URI scheme, encoded as a single utf-8 data point, or the special `none` scheme, encoded as `0x01`. The available schemes are listed on the Bluetooth website.

Examples of encoded URIs:

Table 28: URI encoding examples

URI	Encoded
`http://example.com`	`\x16//example.com`
`https://www.zephyrproject.org/`	`\x17//www.zephyrproject.org/`
`just a string`	`\x01just a string`

Provisioning invitation The provisioner initiates the Provisioning process by sending a Provisioning invitation. The invitations prompts the provisionee to call attention to itself using the Health Server *Attention state*, if available.

The Unprovisioned device automatically responds to the invite by presenting a list of its capabilities, including the supported Out of Band Authentication methods and algorithms.

Public key exchange Before the provisioning process can begin, the provisioner and the unprovisioned device exchange public keys, either in-band or Out of Band (OOB).

In-band public key exchange is a part of the provisioning process and always supported by the unprovisioned device and provisioner.

If the application wants to support public key exchange via OOB, it needs to provide public and private keys to the mesh stack. The unprovisioned device will reflect this in its capabilities. The provisioner obtains the public key via any available OOB mechanism (e.g. the device may advertise a packet containing the public key or it can be encoded in a QR code printed on the

device packaging). Note that even if the unprovisioned device has specified the public key for the Out of Band exchange, the provisioner may choose to exchange the public key in-band if it can't retrieve the public key via OOB mechanism. In this case, a new key pair will be generated by the mesh stack for each Provisioning process.

To enable support of OOB public key on the unprovisioned device side, `CONFIG_BT_MESH_PROV_OOB_PUBLIC_KEY` needs to be enabled. The application must provide public and private keys before the Provisioning process is started by initializing pointers to `bt_mesh_prov.public_key_be` and `bt_mesh_prov.private_key_be`. The keys needs to be provided in big-endian bytes order.

To provide the device's public key obtained via OOB, call `bt_mesh_prov_remote_pub_key_set()` on the provisioner side.

Authentication After the initial exchange, the provisioner selects an Out of Band (OOB) Authentication method. This allows the user to confirm that the device the provisioner connected to is actually the device they intended, and not a malicious third party.

The Provisioning API supports the following authentication methods for the provisionee:

- **Static OOB:** An authentication value is assigned to the device in production, which the provisioner can query in some application specific way. For secure provisioning with the BTM_ECDH_P256_HMAC_SHA256_AES_CCM algorithm, the Static OOB value should contain more than 128 bits of entropy to provide adequate security against attacks.
- **Input OOB:** The user inputs the authentication value. The available input actions are listed in `bt_mesh_input_action_t`.
- **Output OOB:** Show the user the authentication value. The available output actions are listed in `bt_mesh_output_action_t`.

The application must provide callbacks for the supported authentication methods in `bt_mesh_prov`, as well as enabling the supported actions in `bt_mesh_prov.output_actions` and `bt_mesh_prov.input_actions`.

When an Output OOB action is selected, the authentication value should be presented to the user when the output callback is called, and remain until the `bt_mesh_prov.input_complete` or `bt_mesh_prov.complete` callback is called. If the action is `blink`, `beep` or `vibrate`, the sequence should be repeated after a delay of three seconds or more.

When an Input OOB action is selected, the user should be prompted when the application receives the `bt_mesh_prov.input` callback. The user response should be fed back to the Provisioning API through `bt_mesh_input_string()` or `bt_mesh_input_numeric()`. If no user response is recorded within 60 seconds, the Provisioning process is aborted.

If Provisionee wants to mandate OOB authentication, it is mandatory to use the BT_MESH_ECDH_P256_HMAC_SHA256_AES_CCM algorithm.

Data transfer After the device has been successfully authenticated, the provisioner transfers the Provisioning data:

- Unicast address
- A network key
- IV index
- Network flags
  - Key refresh
  - IV update

Additionally, a device key is generated for the node. All this data is stored by the mesh stack, and the provisioning `bt_mesh_prov.complete` callback gets called.

Provisioning security Depending on the choice of public key exchange mechanism and authentication method, the provisioning process can be secure or insecure.

On May 24th 2021, ANSSI disclosed a set of vulnerabilities in the Bluetooth Mesh provisioning protocol that showcased how the low entropy provided by the Blink, Vibrate, Push, Twist and Input/Output numeric OOB methods could be exploited in impersonation and MITM attacks. In response, the Bluetooth SIG has reclassified these OOB methods as insecure in the Bluetooth Mesh Profile Specification v1.0.1 erratum 16350, as AuthValue may be brute forced in real time. To ensure secure provisioning, applications should use a static OOB value and OOB public key transfer.

API reference

Provisioning

Proxy The Proxy feature allows legacy devices like phones to access the Bluetooth Mesh network through GATT. The Proxy feature is only compiled in if the `CONFIG_BT_MESH_GATT_PROXY` option is set. The Proxy feature state is controlled by the *Configuration Server*, and the initial value can be set with `bt_mesh_cfg_srv.gatt_proxy`.

Nodes with the Proxy feature enabled can advertise with Network Identity and Node Identity, which is controlled by the *Configuration Client*.

The GATT Proxy state indicates if the Proxy feature is supported.

Private Proxy A node supporting the Proxy feature and the *Private Beacon Server* model can advertise with Private Network Identity and Private Node Identity types, which is controlled by the *Private Beacon Client*. By advertising with this set of identification types, the node allows the legacy device to connect to the network over GATT while maintaining the privacy of the network.

The Private GATT Proxy state indicates whether the Private Proxy functionality is supported.

Proxy Solicitation In the case where both GATT Proxy and Private GATT Proxy states are disabled on a node, a legacy device cannot connect to it. A node supporting the *On-Demand Private Proxy Server* may however be solicited to advertise connectable advertising events without enabling the Private GATT Proxy state. To solicit the node, the legacy device can send a Solicitation PDU by calling the `bt_mesh_proxy_solicit()` function. To enable this feature, the device must to be compiled with the `CONFIG_BT_MESH_PROXY_SOLICITATION` option set.

Solicitation PDUs are non-mesh, non-connectable, undirected advertising messages containing Proxy Solicitation UUID, encrypted with the network key of the subnet that the legacy device wants to connect to. The PDU contains the source address of the legacy device and a sequence number. The sequence number is maintained by the legacy device and is incremented for every new Solicitation PDU sent.

Each node supporting the Solicitation PDU reception holds its own Solicitation Replay Protection List (SRPL). The SRPL protects the solicitation mechanism from replay attacks by storing solicitation sequence number (SSEQ) and solicitation source (SSRC) pairs of valid Solicitation PDUs processed by the node. The delay between updating the SRPL and storing the change to the persistent storage is defined by `CONFIG_BT_MESH_RPL_STORE_TIMEOUT`.

The Solicitation PDU RPL Configuration models, *Solicitation PDU RPL Configuration Client* and *Solicitation PDU RPL Configuration Server*, provide the functionality of saving and clearing SRPL entries. A node that supports the Solicitation PDU RPL Configuration Client model can clear a section of the SRPL on the target by calling the `bt_mesh_sol_pdu_rpl_clear()` function. Communication between the Solicitation PDU RPL Configuration Client and Server is encrypted using the application key, therefore, the Solicitation PDU RPL Configuration Client can be instantiated on any device in the network.

When the node receives the Solicitation PDU and successfully authenticates it, it will start advertising connectable advertisements with the Private Network Identity type. The duration of the advertisement can be configured by the On-Demand Private Proxy Client model.

API reference

Proxy

Heartbeat The Heartbeat feature provides functionality for monitoring Bluetooth Mesh nodes and determining the distance between nodes.

The Heartbeat feature is configured through the *Configuration Server* model.

Heartbeat messages Heartbeat messages are sent as transport control packets through the network, and are only encrypted with a network key. Heartbeat messages contain the original Time To Live (TTL) value used to send the message and a bitfield of the active features on the node. Through this, a receiving node can determine how many relays the message had to go through to arrive at the receiver, and what features the node supports.

Available Heartbeat feature flags:

- `BT_MESH_FEAT_RELAY`
- `BT_MESH_FEAT_PROXY`
- `BT_MESH_FEAT_FRIEND`
- `BT_MESH_FEAT_LOW_POWER`

Heartbeat publication Heartbeat publication is controlled through the Configuration models, and can be triggered in two ways:

Periodic publication
: The node publishes a new Heartbeat message at regular intervals. The publication can be configured to stop after a certain number of messages, or continue indefinitely.

Triggered publication
: The node publishes a new Heartbeat message every time a feature changes. The set of features that can trigger the publication is configurable.

The two publication types can be combined.

Heartbeat subscription A node can be configured to subscribe to Heartbeat messages from one node at the time. To receive a Heartbeat message, both the source and destination must match the configured subscription parameters.

Heartbeat subscription is always time limited, and throughout the subscription period, the node keeps track of the number of received Heartbeats as well as the minimum and maximum received hop count.

All Heartbeats received with the configured subscription parameters are passed to the `bt_mesh_hb_cb::recv` event handler.

When the Heartbeat subscription period ends, the `bt_mesh_hb_cb::sub_end` callback gets called.

API reference

Heartbeat

Runtime Configuration The runtime configuration API allows applications to change their runtime configuration directly, without going through the Configuration models.

Bluetooth Mesh nodes should generally be configured by a central network configurator device with a *Configuration Client* model. Each mesh node instantiates a *Configuration Server* model that the Configuration Client can communicate with to change the node configuration. In some cases, the mesh node can't rely on the Configuration Client to detect or determine local constraints, such as low battery power or changes in topology. For these scenarios, this API can be used to change the configuration locally.

> **Note**
>
> Runtime configuration changes before the node is provisioned will not be stored in the *persistent storage*.

API reference
Runtime Configuration

Mesh Statistic The statistic API provides runtime monitoring of Bluetooth Mesh communication.

Frame statistic The frame statistic API allows monitoring the number of received frames over different interfaces, and the number of planned and succeeded transmission and relaying attempts.

The API helps the user to estimate the efficiency of the advertiser configuration parameters and the scanning ability of the device. The number of the monitored parameters can be easily extended by customer values.

LPN timing measurement When `CONFIG_BT_MESH_LOW_POWER` is enabled, the statistic module measures LPN friendship timing parameters by timestamping protocol events:

- T1: Poll TX complete
- T2: Scanner enabled (ReceiveDelay elapsed)
- T3: Friend response received or ReceiveWindow expired

From these timestamps the module computes the measured ReceiveDelay (T2 - T1) and ReceiveWindow (T3 - T2) in microseconds. The application can use these values to evaluate the actual on-air timing against configured friendship parameters.

An application can read out and reset statistics at any time.

API reference
Statistic

Bluetooth Mesh Shell The Bluetooth Mesh shell subsystem provides a set of Bluetooth Mesh shell commands for the *Shell* module. It allows for testing and exploring the Bluetooth Mesh API through an interactive interface, without having to write an application.

The Bluetooth Mesh shell interface provides access to most Bluetooth Mesh features, including provisioning, configuration, and message sending.

Prerequisites The Bluetooth Mesh shell subsystem depends on the application to create the composition data and do the mesh initialization.

Application The Bluetooth Mesh shell subsystem is most easily used through the Bluetooth Mesh shell application under `tests/bluetooth/mesh_shell`. See *Shell* for information on how to connect and interact with the Bluetooth Mesh shell application.

Basic usage The Bluetooth Mesh shell subsystem adds a single `mesh` command, which holds a set of sub-commands. Every time the device boots up, make sure to call `mesh init` before any of the other Bluetooth Mesh shell commands can be called:

```
uart:~$ mesh init
```

This is done to ensure that all available log will be printed to the shell output.

Provisioning The mesh node must be provisioned to become part of the network. This is only necessary the first time the device boots up, as the device will remember its provisioning data between reboots.

The simplest way to provision the device is through self-provisioning. To do this the user must provision the device with the default network key and address `0x0001`, execute:

```
uart:~$ mesh prov local 0 0x0001
```

Since all mesh nodes use the same values for the default network key, this can be done on multiple devices, as long as they're assigned non-overlapping unicast addresses. Alternatively, to provision the device into an existing network, the unprovisioned beacon can be enabled with `mesh prov pb-adv on` or `mesh prov pb-gatt on`. The beacons can be picked up by an external provisioner, which can provision the node into its network.

Once the mesh node is part of a network, its transmission parameters can be controlled by the general configuration commands:

- To set the destination address, call `mesh target dst <Addr>`.
- To set the network key index, call `mesh target net <NetKeyIdx>`.
- To set the application key index, call `mesh target app <AppKeyIdx>`.

By default, the transmission parameters are set to send messages to the provisioned address and network key.

Configuration By setting the destination address to the local unicast address (`0x0001` in the `mesh prov local` command above), we can perform self-configuration through any of the *Models* commands.

A good first step is to read out the node's own composition data:

```
uart:~$ mesh models cfg get-comp
```

This prints a list of the composition data of the node, including a list of its model IDs.

Next, since the device has no application keys by default, it's a good idea to add one:

```
uart:~$ mesh models cfg appkey add 0 0
```

Message sending With an application key added (see above), the mesh node's transition parameters are all valid, and the Bluetooth Mesh shell can send raw mesh messages through the network.

For example, to send a Generic OnOff Set message, call:

```
uart:~$ mesh test net-send 82020100
```

> **Note**
>
> All multibyte fields model messages are in little endian, except the opcode.

The message will be sent to the current destination address, using the current network and application key indexes. As the destination address points to the local unicast address by default, the device will only send packets to itself. To change the destination address to the All Nodes broadcast address, call:

```
uart:~$ mesh target dst 0xffff
```

With the destination address set to `0xffff`, any other mesh nodes in the network with the configured network and application keys will receive and process the messages we send.

> **Note**
>
> To change the configuration of the device, the destination address must be set back to the local unicast address before issuing any configuration commands.

Sending raw mesh packets is a good way to test model message handler implementations during development, as it can be done without having to implement the sending model. By default, only the reception of the model messages can be tested this way, as the Bluetooth Mesh shell only includes the foundation models. To receive a packet in the mesh node, you have to add a model with a valid opcode handler list to the composition data in `subsys/bluetooth/mesh/shell.c`, and print the incoming message to the shell in the handler callback.

Parameter formats The Bluetooth Mesh shell commands are parsed with a variety of formats:

Table 29: Parameter formats

Type	Description	Example
Integers	The default format unless something else is specified. Can be either decimal or hexadecimal.	`1234`, `0xabcd01234`
Hexstrings	For raw byte arrays, like UUIDs, key values and message payloads, the parameters should be formatted as an unbroken string of hexadecimal values without any prefix.	`deadbeef01234`
Booleans	Boolean values are denoted in the API documentation as `<val(off, on)>`.	`on`, `off`, `enabled`, `disabled`, `1`, `0`

Commands The Bluetooth Mesh shell implements a large set of commands. Some of the commands accept parameters, which are mentioned in brackets after the command name. For example, `mesh lpn set <value: off, on>`. Mandatory parameters are marked with angle brackets (e.g. `<NetKeyIdx>`), and optional parameters are marked with square brackets (e.g. `[DstAddr]`).

The Bluetooth Mesh shell commands are divided into the following groups:

- *General configuration*
- *Target*
- *Low Power Node*
- *Testing*

- *Provisioning*
- *Proxy*
- *Models*
- *Configuration database*
- *Statistic*

Note

Some commands depend on specific features being enabled in the compile time configuration of the application. Not all features are enabled by default. The list of available Bluetooth mesh shell commands can be shown in the shell by calling `mesh` without any arguments.

General configuration

`mesh init`

Initialize the mesh shell. This command must be run before any other mesh command.

`mesh reset-local`

Reset the local mesh node to its initial unprovisioned state. This command will also clear the Configuration Database (CDB) if present.

Target The target commands enables the user to monitor and set the target destination address, network index and application index for the shell. These parameters are used by several commands, like provisioning, Configuration Client, etc.

`mesh target dst [DstAddr]`

Get or set the message destination address. The destination address determines where mesh packets are sent with the shell, but has no effect on modules outside the shell's control.

- `DstAddr`: If present, sets the new 16-bit mesh destination address. If omitted, the current destination address is printed.

`mesh target net [NetKeyIdx]`

Get or set the message network index. The network index determines which network key is used to encrypt mesh packets that are sent with the shell, but has no effect on modules outside the shell's control. The network key must already be added to the device, either through provisioning or by a Configuration Client.

- `NetKeyIdx`: If present, sets the new network index. If omitted, the current network index is printed.

`mesh target app [AppKeyIdx]`

Get or set the message application index. The application index determines which application key is used to encrypt mesh packets that are sent with the shell, but has no effect on modules outside the shell's control. The application key must already be added to the device by a Configuration Client, and must be bound to the current network index.

- `AppKeyIdx`: If present, sets the new application index. If omitted, the current application index is printed.

Low Power Node

`mesh lpn set <Val(off, on)>`

Enable or disable Low Power operation. Once enabled, the device will turn off its radio and start polling for friend nodes.

- `Val`: Sets whether Low Power operation is enabled.

`mesh lpn poll`

Perform a poll to the friend node, to receive any pending messages. Only available when LPN is enabled.

Testing

`mesh test net-send <HexString>`

Send a raw mesh message with the current destination address, network and application index. The message opcode must be encoded manually.

- `HexString` Raw hexadecimal representation of the message to send.

`mesh test iv-update`

Force an IV update.

`mesh test iv-update-test <Val(off, on)>`

Set the IV update test mode. In test mode, the IV update timing requirements are bypassed.

- `Val`: Enable or disable the IV update test mode.

`mesh test rpl-clear`

Clear the replay protection list, forcing the node to forget all received messages.

Warning

Clearing the replay protection list breaks the security mechanisms of the mesh node, making it susceptible to message replay attacks. This should never be performed in a real deployment.

Health Server Test

`mesh test health-srv add-fault <FaultID>`

Register a new Health Server Fault for the Linux Foundation Company ID.

- `FaultID`: ID of the fault to register (`0x0001` to `0xFFFF`)

`mesh test health-srv del-fault [FaultID]`

Remove registered Health Server faults for the Linux Foundation Company ID.

- `FaultID`: If present, the given fault ID will be deleted. If omitted, all registered faults will be cleared.

Provisioning To allow a device to broadcast connectable unprovisioned beacons, the `CONFIG_BT_MESH_PROVISIONEE` configuration option must be enabled, along with the `CONFIG_BT_MESH_PB_GATT` option.

`mesh prov pb-gatt <Val(off, on)>`

Start or stop advertising a connectable unprovisioned beacon. The connectable unprovisioned beacon allows the mesh node to be discovered by nearby GATT based provisioners, and provisioned through the GATT bearer.

- `Val`: Enable or disable provisioning with GATT

To allow a device to broadcast unprovisioned beacons, the `CONFIG_BT_MESH_PROVISIONEE` configuration option must be enabled, along with the `CONFIG_BT_MESH_PB_ADV` option.

`mesh prov pb-adv <Val(off, on)>`

Start or stop advertising the unprovisioned beacon. The unprovisioned beacon allows the mesh node to be discovered by nearby advertising-based provisioners, and provisioned through the advertising bearer.

- `Val`: Enable or disable provisioning with advertiser

To allow a device to provision devices, the `CONFIG_BT_MESH_PROVISIONER` and `CONFIG_BT_MESH_PB_ADV` configuration options must be enabled.

`mesh prov remote-adv <UUID(1-16 hex)> <NetKeyIdx> <Addr> <AttDur(s)>`

Provision a nearby device into the mesh. The mesh node starts scanning for unprovisioned beacons with the given UUID. Once found, the unprovisioned device will be added to the mesh network with the given unicast address, and given the network key indicated by `NetKeyIdx`.

- `UUID`: UUID of the unprovisioned device. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `NetKeyIdx`: Index of the network key to pass to the device.
- `Addr`: First unicast address to assign to the unprovisioned device. The device will occupy as many addresses as it has elements, and all must be available.
- `AttDur`: The duration in seconds the unprovisioned device will identify itself for, if supported. See *Attention state* for details.

To allow a device to provision devices over GATT, the `CONFIG_BT_MESH_PROVISIONER` and `CONFIG_BT_MESH_PB_GATT_CLIENT` configuration options must be enabled.

`mesh prov remote-gatt <UUID(1-16 hex)> <NetKeyIdx> <Addr> <AttDur(s)>`

Provision a nearby device into the mesh. The mesh node starts scanning for connectable advertising for PB-GATT with the given UUID. Once found, the unprovisioned device will be added to the mesh network with the given unicast address, and given the network key indicated by `NetKeyIdx`.

- `UUID`: UUID of the unprovisioned device. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.

- `NetKeyIdx`: Index of the network key to pass to the device.
- `Addr`: First unicast address to assign to the unprovisioned device. The device will occupy as many addresses as it has elements, and all must be available.
- `AttDur`: The duration in seconds the unprovisioned device will identify itself for, if supported. See *Attention state* for details.

`mesh prov uuid [UUID(1-16 hex)]`

Get or set the mesh node's UUID, used in the unprovisioned beacons.

- `UUID`: If present, new 128-bit UUID value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the current UUID will be printed. To enable this command, the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option must be enabled.

`mesh prov input-num <Number>`

Input a numeric OOB authentication value. Only valid when prompted by the shell during provisioning. The input number must match the number presented by the other participant in the provisioning.

- `Number`: Decimal authentication number.

`mesh prov input-str <String>`

Input an alphanumeric OOB authentication value. Only valid when prompted by the shell during provisioning. The input string must match the string presented by the other participant in the provisioning.

- `String`: Unquoted alphanumeric authentication string.

`mesh prov static-oob [Val(1-32 hex)]`

Set or clear the static OOB authentication value. The static OOB authentication value must be set before provisioning starts to have any effect. The static OOB value must be same on both participants in the provisioning. To enable this command, the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option must be enabled.

- `Val`: If present, indicates the new hexadecimal value of the static OOB. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the static OOB value is cleared.

`mesh prov local <NetKeyIdx> <Addr> [IVI]`

Provision the mesh node itself. If the Configuration database is enabled, the network key must be created. Otherwise, the default key value is used.

- `NetKeyIdx`: Index of the network key to provision.
- `Addr`: First unicast address to assign to the device. The device will occupy as many addresses as it has elements, and all must be available.
- `IVI`: Indicates the current network IV index. Defaults to 0 if omitted.

`mesh prov beacon-listen <Val(off, on)>`

Enable or disable printing of incoming unprovisioned beacons. Allows a provisioner device to detect nearby unprovisioned devices and provision them. To enable this command, the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option must be enabled.

- `Val`: Whether to enable the unprovisioned beacon printing.

`mesh prov remote-pub-key <PubKey>`

Provide Device public key.

- `PubKey` - Device public key in big-endian.

`mesh prov auth-method input <Action> <Size>`

From the provisioner device, instruct the unprovisioned device to use the specified Input OOB authentication action.

- `Action` - Input action. Allowed values:
  - `0` - No input action.
  - `1` - Push action set.
  - `2` - Twist action set.
  - `4` - Enter number action set.
  - `8` - Enter String action set.
- `Size` - Authentication size.

`mesh prov auth-method output <Action> <Size>`

From the provisioner device, instruct the unprovisioned device to use the specified Output OOB authentication action.

- `Action` - Output action. Allowed values:
  - `0` - No output action.
  - `1` - Blink action set.
  - `2` - Vibrate action set.
  - `4` - Display number action set.
  - `8` - Display String action set.
- `Size` - Authentication size.

`mesh prov auth-method static <Val(1-16 hex)>`

From the provisioner device, instruct the unprovisioned device to use static OOB authentication, and use the given static authentication value when provisioning.

- `Val` - Static OOB value. Providing a hex-string shorter than 32 bytes will populate the N most significant bytes of the array and zero-pad the rest.

`mesh prov auth-method none`

From the provisioner device, don't use any authentication when provisioning new devices. This is the default behavior.

Proxy The Proxy Server module is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_GATT_PROXY` configuration option.

`mesh proxy identity-enable`

Enable the Proxy Node Identity beacon, allowing Proxy devices to connect explicitly to this device. The beacon will run for 60 seconds before the node returns to normal Proxy beacons.

The Proxy Client module is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_PROXY_CLIENT` configuration option.

`mesh proxy connect <NetKeyIdx>`

Auto-Connect a nearby proxy server into the mesh.

- `NetKeyIdx`: Index of the network key to connect.

`mesh proxy disconnect <NetKeyIdx>`

Disconnect the existing proxy connection.

- `NetKeyIdx`: Index of the network key to disconnect.

`mesh proxy solicit <NetKeyIdx>`

Begin Proxy Solicitation of a subnet. Support of this feature can be enabled through the `CONFIG_BT_MESH_PROXY_SOLICITATION` configuration option.

- `NetKeyIdx`: Index of the network key to send Solicitation PDUs to.

Models

Configuration Client The Configuration Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_CFG_CLI` configuration option. This is implemented as a separate module (`mesh models cfg`) inside the `mesh models` subcommand list. This module will work on any instance of the Configuration Client model if the mentioned shell configuration options is enabled, and as long as the Configuration Client model is present in the model composition of the application. This shell module can be used for configuring itself and other nodes in the mesh network.

The Configuration Client uses general message parameters set by `mesh target dst` and `mesh target net` to target specific nodes. When the Bluetooth Mesh shell node is provisioned, given that the `CONFIG_BT_MESH_SHELL_PROV_CTX_INSTANCE` option is enabled with the shell provisioning context initialized, the Configuration Client model targets itself by default. Similarly, when another node has been provisioned by the Bluetooth Mesh shell, the Configuration Client model targets the new node. In most common use-cases, the Configuration Client is depending on the provisioning features and the Configuration database to be fully functional. The Configuration Client always sends messages using the Device key bound to the destination address, so it will only be able to configure itself and the mesh nodes it provisioned. The following steps are an example of how you can set up a device to start using the Configuration Client commands:

- Initialize the client node (`mesh init`).
- Create the CDB (`mesh cdb create`).
- Provision the local device (`mesh prov local`).
- The shell module should now target itself.
- Monitor the composition data of the local node (`mesh models cfg get-comp`).
- Configure the local node as desired with the Configuration Client commands.
- Provision other devices (`mesh prov beacon-listen`) (`mesh prov remote-adv`) (`mesh prov remote-gatt`).

- The shell module should now target the newly added node.
- Monitor the newly provisioned nodes and their addresses (`mesh cdb show`).
- Monitor the composition data of the target device (`mesh models cfg get-comp`).
- Configure the node as desired with the Configuration Client commands.

`mesh models cfg target get`

Get the target Configuration server for the Configuration Client model.

`mesh models cfg help`

Print information for the Configuration Client shell module.

`mesh models cfg reset`

Reset the target device.

`mesh models cfg timeout [Timeout(s)]`

Get and set the Config Client model timeout used during message sending.

- `Timeout`: If present, set the Config Client model timeout in seconds. If omitted, the current timeout is printed.

`mesh models cfg get-comp [Page]`

Read a composition data page. The full composition data page will be printed. If the target does not have the given page, it will return the last page before it.

- `Page`: The composition data page to request. Defaults to 0 if omitted.

`mesh models cfg beacon [Val(off, on)]`

Get or set the network beacon transmission.

- `Val`: If present, enables or disables sending of the network beacon. If omitted, the current network beacon state is printed.

`mesh models cfg ttl [TTL]`

Get or set the default TTL value.

- `TTL`: If present, sets the new default TTL value. Legal TTL values are 0x00 and 0x02-0x7f. If omitted, the current default TTL value is printed.

`mesh models cfg friend [Val(off, on)]`

Get or set the Friend feature.

- `Val`: If present, enables or disables the Friend feature. If omitted, the current Friend feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.

`mesh models cfg gatt-proxy [Val(off, on)]`

Get or set the GATT Proxy feature.

- `Val`: If present, enables or disables the GATT Proxy feature. If omitted, the current GATT Proxy feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.

`mesh models cfg relay [<Val(off, on)> [<Count> [Int(ms)]]]`

Get or set the Relay feature and its parameters.

- `Val`: If present, enables or disables the Relay feature. If omitted, the current Relay feature state is printed:
  - `0x00`: The feature is supported, but disabled.
  - `0x01`: The feature is enabled.
  - `0x02`: The feature is not supported.
- `Count`: Sets the new relay retransmit count if `val` is `on`. Ignored if `val` is `off`. Legal retransmit count is 0-7. Defaults to `2` if omitted.
- `Int`: Sets the new relay retransmit interval in milliseconds if `val` is `on`. Legal interval range is 10-320 milliseconds. Ignored if `val` is `off`. Defaults to `20` if omitted.

`mesh models cfg node-id <NetKeyIdx> [Identity]`

Get or Set of current Node Identity state of a subnet.

- `NetKeyIdx`: The network key index to Get/Set.
- `Identity`: If present, sets the identity of Node Identity state.

`mesh models cfg polltimeout-get <LPNAddr>`

Get current value of the PollTimeout timer of the LPN within a Friend node.

- `LPNAddr` Address of Low Power node.

`mesh models cfg net-transmit-param [<Count> <Int(ms)>]`

Get or set the network transmit parameters.

- `Count`: Sets the number of additional network transmits for every sent message. Legal retransmit count is 0-7.
- `Int`: Sets the new network retransmit interval in milliseconds. Legal interval range is 10-320 milliseconds.

`mesh models cfg netkey add <NetKeyIdx> [Key(1-16 hex)]`

Add a network key to the target node. Adds the key to the Configuration Database if enabled.

- `NetKeyIdx`: The network key index to add.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. Only valid if the key does not already exist in the Configuration Database. If omitted, the default key value is used.

`mesh models cfg netkey upd <NetKeyIdx> [Key(1-16 hex)]`

Update a network key to the target node.

- `NetKeyIdx`: The network key index to updated.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the default key value is used.

`mesh models cfg netkey get`

Get a list of known network key indexes.

`mesh models cfg netkey del <NetKeyIdx>`

Delete a network key from the target node.

- `NetKeyIdx`: The network key index to delete.

`mesh models cfg appkey add <NetKeyIdx> <AppKeyIdx> [Key(1-16 hex)]`

Add an application key to the target node. Adds the key to the Configuration Database if enabled.

- `NetKeyIdx`: The network key index the application key is bound to.
- `AppKeyIdx`: The application key index to add.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. Only valid if the key does not already exist in the Configuration Database. If omitted, the default key value is used.

`mesh models cfg appkey upd <NetKeyIdx> <AppKeyIdx> [Key(1-16 hex)]`

Update an application key to the target node.

- `NetKeyIdx`: The network key index the application key is bound to.
- `AppKeyIdx`: The application key index to update.
- `Key`: If present, sets the key value as a 128-bit hexadecimal value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, the default key value is used.

`mesh models cfg appkey get <NetKeyIdx>`

Get a list of known application key indexes bound to the given network key index.

- `NetKeyIdx`: Network key indexes to get a list of application key indexes from.

`mesh models cfg appkey del <NetKeyIdx> <AppKeyIdx>`

Delete an application key from the target node.

- `NetKeyIdx`: The network key index the application key is bound to.
- `AppKeyIdx`: The application key index to delete.

`mesh models cfg model app-bind <Addr> <AppKeyIdx> <MID> [CID]`

Bind an application key to a model. Models can only encrypt and decrypt messages sent with application keys they are bound to.

- `Addr`: Address of the element the model is on.
- `AppKeyIdx`: The application key to bind to the model.
- `MID`: The model ID of the model to bind the key to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model app-unbind <Addr> <AppKeyIdx> <MID> [CID]`

Unbind an application key from a model.

- `Addr`: Address of the element the model is on.
- `AppKeyIdx`: The application key to unbind from the model.
- `MID`: The model ID of the model to unbind the key from.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model app-get <ElemAddr> <MID> [CID]`

Get a list of application keys bound to a model.

- `ElemAddr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the bound keys of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model pub <Addr> <MID> [CID] [<PubAddr> <AppKeyIdx> <Cred(off, on)> <TTL> <PerRes> <PerSteps> <Count> <Int(ms)>]`

Get or set the publication parameters of a model. If all publication parameters are included, they become the new publication parameters of the model. If all publication parameters are omitted, print the current publication parameters of the model.

- `Addr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the bound keys of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

Publication parameters:

- `PubAddr`: The destination address to publish to.
- `AppKeyIdx`: The application key index to publish with.
- `Cred`: Whether to publish with Friendship credentials when acting as a Low Power Node.
- `TTL`: TTL value to publish with (`0x00` to `0x07f`).
- `PerRes`: Resolution of the publication period steps:
  - `0x00`: The Step Resolution is 100 milliseconds
  - `0x01`: The Step Resolution is 1 second
  - `0x02`: The Step Resolution is 10 seconds

  - `0x03`: The Step Resolution is 10 minutes
- `PerSteps`: Number of publication period steps, or 0 to disable periodic publication.
- `Count`: Number of retransmission for each published message (`0` to `7`).
- `Int` The interval between each retransmission, in milliseconds. Must be a multiple of 50.

`mesh models cfg model pub-va <Addr> <UUID(1-16 hex)> <AppKeyIdx> <Cred(off, on)> <TTL> <PerRes> <PerSteps> <Count> <Int(ms)> <MID> [CID]`

Set the publication parameters of a model.

- `Addr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the bound keys of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

Publication parameters:

- `UUID`: The destination virtual address to publish to. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `AppKeyIdx`: The application key index to publish with.
- `Cred`: Whether to publish with Friendship credentials when acting as a Low Power Node.
- `TTL`: TTL value to publish with (`0x00` to `0x07f`).
- `PerRes`: Resolution of the publication period steps:
  - `0x00`: The Step Resolution is 100 milliseconds
  - `0x01`: The Step Resolution is 1 second
  - `0x02`: The Step Resolution is 10 seconds
  - `0x03`: The Step Resolution is 10 minutes
- `PerSteps`: Number of publication period steps, or 0 to disable periodic publication.
- `Count`: Number of retransmission for each published message (`0` to `7`).
- `Int` The interval between each retransmission, in milliseconds. Must be a multiple of 50.

`mesh models cfg model sub-add <ElemAddr> <SubAddr> <MID> [CID]`

Subscription the model to a group address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `ElemAddr`: Address of the element the model is on.
- `SubAddr`: 16-bit group address the model should subscribe to (`0xc000` to `0xFEFF`).
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-del <ElemAddr> <SubAddr> <MID> [CID]`

Unsubscribe a model from a group address.

- `ElemAddr`: Address of the element the model is on.
- `SubAddr`: 16-bit group address the model should remove from its subscription list (`0xc000` to `0xFEFF`).
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-add-va <ElemAddr> <LabelUUID(1-16 hex)> <MID> [CID]`

Subscribe the model to a virtual address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `ElemAddr`: Address of the element the model is on.
- `LabelUUID`: 128-bit label UUID of the virtual address to subscribe to. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-del-va <ElemAddr> <LabelUUID(1-16 hex)> <MID> [CID]`

Unsubscribe a model from a virtual address.

- `ElemAddr`: Address of the element the model is on.
- `LabelUUID`: 128-bit label UUID of the virtual address to remove the subscription of. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-ow <ElemAddr> <SubAddr> <MID> [CID]`

Overwrite all model subscriptions with a single new group address.

- `ElemAddr`: Address of the element the model is on.
- `SubAddr`: 16-bit group address the model should added to the subscription list (`0xc000` to `0xFEFF`).
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-ow-va <ElemAddr> <LabelUUID(1-16 hex)> <MID> [CID]`

Overwrite all model subscriptions with a single new virtual address. Models only receive messages sent to their unicast address or a group or virtual address they subscribe to. Models may subscribe to multiple group and virtual addresses.

- `ElemAddr`: Address of the element the model is on.

- `LabelUUID`: 128-bit label UUID of the virtual address as the new Address to be added to the subscription list. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `MID`: The model ID of the model to add the subscription to.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-del-all <ElemAddr> <MID> [CID]`

Remove all group and virtual address subscriptions from of a model.

- `ElemAddr`: Address of the element the model is on.
- `MID`: The model ID of the model to Unsubscribe all.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg model sub-get <ElemAddr> <MID> [CID]`

Get a list of addresses the model subscribes to.

- `ElemAddr`: Address of the element the model is on.
- `MID`: The model ID of the model to get the subscription list of.
- `CID`: If present, determines the Company ID of the model. If omitted, the model is a Bluetooth SIG defined model.

`mesh models cfg krp <NetKeyIdx> [Phase]`

Get or set the key refresh phase of a subnet.

- `NetKeyIdx`: The identified network key used to Get/Set the current Key Refresh Phase state.
- `Phase`: New Key Refresh Phase. Valid phases are:
  - `0x00`: Normal operation; Key Refresh procedure is not active
  - `0x01`: First phase of Key Refresh procedure
  - `0x02`: Second phase of Key Refresh procedure

`mesh models cfg hb-sub [<Src> <Dst> <Per>]`

Get or set the Heartbeat subscription parameters. A node only receives Heartbeat messages matching the Heartbeat subscription parameters. Sets the Heartbeat subscription parameters if present, or prints the current Heartbeat subscription parameters if called with no parameters.

- `Src`: Unicast source address to receive Heartbeat messages from.
- `Dst`: Destination address to receive Heartbeat messages on.
- `Per`: Logarithmic representation of the Heartbeat subscription period:
  - `0`: Heartbeat subscription will be disabled.
  - `1` to `17`: The node will subscribe to Heartbeat messages for $2^{(\text{period} - 1)}$ seconds.

`mesh models cfg hb-pub [<Dst> <Count> <Per> <TTL> <Features> <NetKeyIdx>]`

Get or set the Heartbeat publication parameters. Sets the Heartbeat publication parameters if present, or prints the current Heartbeat publication parameters if called with no parameters.

- `Dst`: Destination address to publish Heartbeat messages to.
- `Count`: Logarithmic representation of the number of Heartbeat messages to publish periodically:
  - `0`: Heartbeat messages are not published periodically.
  - `1` to `17`: The node will periodically publish $2^{(count - 1)}$ Heartbeat messages.
  - `255`: Heartbeat messages will be published periodically indefinitely.
- `Per`: Logarithmic representation of the Heartbeat publication period:
  - `0`: Heartbeat messages are not published periodically.
  - `1` to `17`: The node will publish Heartbeat messages every $2^{(period - 1)}$ seconds.
- `TTL`: The TTL value to publish Heartbeat messages with (`0x00` to `0x7f`).
- `Features`: Bitfield of features that should trigger a Heartbeat publication when changed:
  - `Bit 0`: Relay feature.
  - `Bit 1`: Proxy feature.
  - `Bit 2`: Friend feature.
  - `Bit 3`: Low Power feature.
- `NetKeyIdx`: Index of the network key to publish Heartbeat messages with.

Health Client The Health Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_HEALTH_CLI` configuration option. This is implemented as a separate module (`mesh models health`) inside the `mesh models` subcommand list. This module will work on any instance of the Health Client model if the mentioned shell configuration options is enabled, and as long as one or more Health Client model(s) is present in the model composition of the application. This shell module can be used to trigger interaction between Health Clients and Servers on devices in a Mesh network.

By default, the module will choose the first Health Client instance in the model composition when using the Health Client commands. To choose a specific Health Client instance the user can utilize the commands `mesh models health instance set` and `mesh models health instance get-all`.

The Health Client may use the general messages parameters set by `mesh target dst`, `mesh target net` and `mesh target app` to target specific nodes. If the shell target destination address is set to zero, the targeted Health Client will attempt to publish messages using its configured publication parameters.

`mesh models health instance set <ElemIdx>`

Set the Health Client model instance to use.

- `ElemIdx`: Element index of Health Client model.

`mesh models health instance get-all`

Prints all available Health Client model instances on the device.

`mesh models health fault-get <CID>`

Get a list of registered faults for a Company ID.

- `CID`: Company ID to get faults for.

`mesh models health fault-clear <CID>`

Clear the list of faults for a Company ID.

- `CID`: Company ID to clear the faults for.

`mesh models health fault-clear-unack <CID>`

Clear the list of faults for a Company ID without requesting a response.

- `CID`: Company ID to clear the faults for.

`mesh models health fault-test <CID> <TestID>`

Invoke a self-test procedure, and show a list of triggered faults.

- `CID`: Company ID to perform self-tests for.
- `TestID`: Test to perform.

`mesh models health fault-test-unack <CID> <TestID>`

Invoke a self-test procedure without requesting a response.

- `CID`: Company ID to perform self-tests for.
- `TestID`: Test to perform.

`mesh models health period-get`

Get the current Health Server publish period divisor.

`mesh models health period-set <Divisor>`

Set the current Health Server publish period divisor. When a fault is detected, the Health Server will start publishing is fault status with a reduced interval. The reduced interval is determined by the Health Server publish period divisor: Fault publish period = Publish period / $2^{divisor}$.

- `Divisor`: The new Health Server publish period divisor.

`mesh models health period-set-unack <Divisor>`

Set the current Health Server publish period divisor. When a fault is detected, the Health Server will start publishing is fault status with a reduced interval. The reduced interval is determined by the Health Server publish period divisor: Fault publish period = Publish period / $2^{divisor}$.

- `Divisor`: The new Health Server publish period divisor.

`mesh models health attention-get`

Get the current Health Server attention state.

`mesh models health attention-set <Time(s)>`

Enable the Health Server attention state for some time.

- `Time`: Duration of the attention state, in seconds (`0` to `255`)

`mesh models health attention-set-unack <Time(s)>`

Enable the Health Server attention state for some time without requesting a response.

- `Time`: Duration of the attention state, in seconds (`0` to `255`)

Binary Large Object (BLOB) Transfer Client model The *BLOB Transfer Client* can be added to the mesh shell by enabling the `CONFIG_BT_MESH_BLOB_CLI` option, and disabling the `CONFIG_BT_MESH_DFU_CLI` option.

`mesh models blob cli target <Addr>`

Add a Target node for the next BLOB transfer.

- `Addr`: Unicast address of the Target node's BLOB Transfer Server model.

`mesh models blob cli caps [<Group> [<TimeoutBase>]]`

Retrieve transfer capabilities for Target nodes.

- `Group`: Optional group address to use when communicating with Target nodes. If omitted, the BLOB Transfer Client will address each Target node individually.
- `TimeoutBase`: Optional time to wait for responses from Target nodes, in 10-second increments.

`mesh models blob cli tx <Id> <Size> <BlockSizeLog> <ChunkSize> [<Group> [<Mode(push, pull)>]]`

Perform a BLOB transfer to Target nodes. The BLOB Transfer Client will send a dummy BLOB to all Target nodes, then post a message when the transfer is completed. Note that all Target nodes must first be configured to receive the transfer using the `mesh models blob srv rx` command.

- `Id`: 64-bit BLOB transfer ID.
- `Size`: Size of the BLOB in bytes.
- `BlockSizeLog`: Logarithmic representation of the BLOB's block size. The final block size will be `1 << block size log` bytes.
- `ChunkSize`: Chunk size in bytes.
- `Group`: Optional group address to use when communicating with Target nodes. If omitted or set to 0, the BLOB Transfer Client will address each Target node individually.
- `Mode`: BLOB transfer mode to use. Must be either `push` (Push BLOB Transfer Mode) or `pull` (Pull BLOB Transfer Mode). If omitted, `push` will be used by default.

`mesh models blob cli tx-cancel`

Cancel an ongoing BLOB transfer.

`mesh models blob cli tx-get [Group]`

Determine the progress of a previously running BLOB transfer. Can be used when not performing a BLOB transfer.

- `Group`: Optional group address to use when communicating with Target nodes. If omitted or set to 0, the BLOB Transfer Client will address each Target node individually.

`mesh models blob cli tx-suspend`

Suspend the ongoing BLOB transfer.

`mesh models blob cli tx-resume`

Resume the suspended BLOB transfer.

`mesh models blob cli instance-set <ElemIdx>`

Use the BLOB Transfer Client model instance on the specified element when using the other BLOB Transfer Client model commands.

- `ElemIdx`: The element on which to find the BLOB Transfer Client model instance to use.

`mesh models blob cli instance-get-all`

Get a list of all BLOB Transfer Client model instances on the node.

BLOB Transfer Server model The *BLOB Transfer Server* can be added to the mesh shell by enabling the `CONFIG_BT_MESH_BLOB_SRV` option. The BLOB Transfer Server model is capable of receiving any BLOB data, but the implementation in the mesh shell will discard the incoming data.

`mesh models blob srv rx <ID> [<TimeoutBase(10s steps)>]`

Prepare to receive a BLOB transfer.

- `ID`: 64-bit BLOB transfer ID to receive.
- `TimeoutBase`: Optional additional time to wait for client messages, in 10-second increments.

`mesh models blob srv rx-cancel`

Cancel an ongoing BLOB transfer.

`mesh models blob srv instance-set <ElemIdx>`

Use the BLOB Transfer Server model instance on the specified element when using the other BLOB Transfer Server model commands.

- `ElemIdx`: The element on which to find the BLOB Transfer Server model instance to use.

`mesh models blob srv instance-get-all`

Get a list of all BLOB Transfer Server model instances on the node.

Binary Large Object (BLOB) Transfer flash stream BLOB flash stream configuration can be added to the mesh shell by enabling the `CONFIG_BT_MESH_SHELL_BLOB_IO_FLASH` option. By default, the shell uses a dummy BLOB stream. This option allows the user to specify which area in the flash to use. See *Flash map* for information on how to obtain related parameters.

`mesh models blob flash-stream-set <AreaID> [<Offset>]`

Set the BLOB stream to a specified area.

- `AreaID`: Flash area ID to write/read the BLOB to/from.
- `Offset`: Optional offset into the flash area to place the BLOB at (in bytes).

`mesh models blob flash-stream-unset`

Set the BLOB stream back to the dummy stream.

Firmware Update Client model The Firmware Update Client model can be added to the mesh shell by enabling configuration options `CONFIG_BT_MESH_BLOB_CLI` and `CONFIG_BT_MESH_DFU_CLI`. The Firmware Update Client demonstrates the firmware update Distributor role by transferring a dummy firmware update to a set of Target nodes.

`mesh models dfu slot add <Size> <FwID> [<Metadata>]`

Add a virtual DFU image slot that can be transferred as a DFU image. The image slot will be assigned an image slot index, which is printed as a response, and can be used to reference the slot in other commands. To update the image slot, remove it using the `mesh models dfu slot del` shell command and then add it again.

- `Size`: DFU image slot size in bytes.
- `FwID`: Firmware ID, formatted as a hexstring.
- `Metadata`: Optional firmware metadata, formatted as a hexstring.

`mesh models dfu slot del <SlotIdx>`

Delete the DFU image slot at the given index.

- `SlotIdx`: Index of the slot to delete.

`mesh models dfu slot get <SlotIdx>`

Get all available information about a DFU image slot.

- `SlotIdx`: Index of the slot to get.

`mesh models dfu cli target <Addr> <ImgIdx>`

Add a Target node.

- `Addr`: Unicast address of the Target node.
- `ImgIdx`: Image index to address on the Target node.

`mesh models dfu cli target-state`

Check the DFU Target state of the device at the configured destination address.

`mesh models dfu cli target-imgs [<MaxCount>]`

Get a list of DFU images on the device at the configured destination address.

- `MaxCount`: Optional maximum number of images to return. If omitted, there's no limit on the number of returned images.

`mesh models dfu cli target-check <SlotIdx> <TargetImgIdx>`

Check whether the device at the configured destination address will accept a DFU transfer from the given DFU image slot to the Target node's DFU image at the given index, and what the effect would be.

- `SlotIdx`: Index of the local DFU image slot to check.
- `TargetImgIdx`: Index of the Target node's DFU image to check.

`mesh models dfu cli send <SlotIdx> [<Group>]`

Start a DFU transfer to all added Target nodes.

- `SlotIdx`: Index of the local DFU image slot to send.
- `Group`: Optional group address to use when communicating with the Target nodes. If omitted, the Firmware Update Client will address each Target node individually.

`mesh models dfu cli cancel [<Addr>]`

Cancel the DFU procedure at any state on a specific Target node or on all Target nodes. When a Target node address is provided, the Firmware Update Client model will try to cancel the DFU procedure on the provided Target node. Otherwise, the Firmware Update Client model will try to cancel the ongoing DFU procedure on all Target nodes.

- `Addr`: Optional unicast address of a Target node on which to cancel the DFU procedure.

`mesh models dfu cli apply`

Apply the most recent DFU transfer on all Target nodes. Can only be called after a DFU transfer is completed.

`mesh models dfu cli confirm`

Confirm that the most recent DFU transfer was successfully applied on all Target nodes. Can only be called after a DFU transfer is completed and applied.

`mesh models dfu cli suspend`

Suspend the ongoing DFU transfer.

`mesh models dfu cli resume`

Resume the suspended DFU transfer.

`mesh models dfu cli progress`

Check the progress of the current transfer.

`mesh models dfu cli instance-set <ElemIdx>`

Use the Firmware Update Client model instance on the specified element when using the other Firmware Update Client model commands.

- `ElemIdx`: The element on which to find the Firmware Update Client model instance to use.

`mesh models dfu cli instance-get-all`

Get a list of all Firmware Update Client model instances on the node.

Firmware Update Server model The Firmware Update Server model can be added to the mesh shell by enabling configuration options `CONFIG_BT_MESH_BLOB_SRV` and `CONFIG_BT_MESH_DFU_SRV`. The Firmware Update Server demonstrates the firmware update Target role by accepting any firmware update. The mesh shell Firmware Update Server will discard the incoming firmware data, but otherwise behave as a proper firmware update Target node.

`mesh models dfu srv applied`

Mark the most recent DFU transfer as applied. Can only be called after a DFU transfer is completed, and the Distributor has requested that the transfer is applied.

As the mesh shell Firmware Update Server doesn't actually apply the incoming firmware image, this command can be used to emulate an applied status, to notify the Distributor that the transfer was successful.

`mesh models dfu srv progress`

Check the progress of the current transfer.

`mesh models dfu srv rx-cancel`

Cancel incoming DFU transfer.

`mesh models dfu srv instance-set <ElemIdx>`

Use the Firmware Update Server model instance on the specified element when using the other Firmware Update Server model commands.

- `ElemIdx`: The element on which to find the Firmware Update Server model instance to use.

`mesh models dfu srv instance-get-all`

Get a list of all Firmware Update Server model instances on the node.

Firmware Distribution Server model The Firmware Distribution Server model commands can be added to the mesh shell by enabling the `CONFIG_BT_MESH_DFD_SRV` configuration option. The shell commands for this model mirror the messages sent to the server by a Firmware Distribution Client model. To use these commands, a Firmware Distribution Server must be instantiated by the application.

`mesh models dfd receivers-add <Addr>,<FwIdx>[;<Addr>,<FwIdx>]...`

Add receivers to the Firmware Distribution Server. Supply receivers as a list of comma-separated addr,fw_idx pairs, separated by semicolons, for example, `0x0001,0;0x0002,0;0x0004,1`. Do not use spaces in the receiver list. Repeated calls to this command will continue populating the receivers list until `mesh models dfd receivers-delete-all` is called.

- `Addr`: Address of the receiving node(s).
- `FwIdx`: Index of the firmware slot to send to `Addr`.

`mesh models dfd receivers-delete-all`

Delete all receivers from the server.

`mesh models dfd receivers-get <First> <Count>`

Get a list of info about firmware receivers.

- `First`: Index of the first receiver to get from the receiver list.
- `Count`: The number of receivers for which to get info.

`mesh models dfd capabilities-get`

Get the capabilities of the server.

`mesh models dfd get`

Get information about the current distribution state, phase and the transfer parameters.

`mesh models dfd start <AppKeyIdx> <SlotIdx> [<Group> [<PolicyApply> [<TTL> [<TimeoutBase> [<XferMode>]]]]]`

Start the firmware distribution.

- `AppKeyIdx`: Application index to use for sending. The common application key should be bound to the Firmware Update and BLOB Transfer models on the Distributor and Target nodes.
- `SlotIdx`: Index of the local image slot to send.
- `Group`: Optional group address to use when communicating with the Target nodes. If omitted, the Firmware Distribution Server will address each Target node individually. To keep addressing each Target node individually while changing other arguments, set this argument value to 0.
- `PolicyApply`: Optional field that corresponds to the update policy. Setting this to `true` will make the Firmware Distribution Server apply the image immediately after the transfer is completed.
- `TTL`: Optional. TTL value to use when sending. Defaults to configured default TTL.
- `TimeoutBase`: Optional additional value used to calculate timeout values in the firmware distribution process, in 10-second increments.. See *Transfer timeout* for information about how `timeout_base` is used to calculate the transfer timeout. Defaults to 0.
- `XferMode`: Optional BLOB transfer mode. 1 = Push mode (Push BLOB Transfer Mode), 2 = Pull mode (Pull BLOB Transfer Mode). Defaults to Push mode.

`mesh models dfd suspend`

Suspends the ongoing distribution.

`mesh models dfd cancel`

Cancel the ongoing distribution.

`mesh models dfd apply`

Apply the distributed firmware.

`mesh models dfd fw-get <FwID>`

Get information about the firmware image uploaded to the server.

- `FwID`: Firmware ID of the image to get.

`mesh models dfd fw-get-by-idx <Idx>`

Get information about the firmware image uploaded to the server in a specific slot.

- `Idx`: Index of the slot to get the image from.

`mesh models dfd fw-delete <FwID>`

Delete a firmware image from the server.

- `FwID`: Firmware ID of the image to delete.

`mesh models dfd fw-delete-all`

Delete all firmware images from the server.

`mesh models dfd instance-set <ElemIdx>`

Use the Firmware Distribution Server model instance on the specified element when using the other Firmware Distribution Server model commands.

- `ElemIdx`: The element on which to find the Firmware Distribution Server model instance to use.

`mesh models dfd instance-get-all`

Get a list of all Firmware Distribution Server model instances on the node.

DFU metadata The DFU metadata shell commands allow generating metadata that can be used by a Target node to check the firmware before accepting it. The commands are enabled through the `CONFIG_BT_MESH_SHELL_DFU_METADATA` configuration option.

`mesh models dfu metadata comp-clear`

Clear the stored composition data to be used for the Target node.

`mesh models dfu metadata comp-add <CID> <ProductID> <VendorID> <Crpl> <Features>`

Create a header of the Composition Data Page 0.

- `CID`: Company identifier assigned by Bluetooth SIG.
- `ProductID`: Vendor-assigned product identifier.
- `VendorID`: Vendor-assigned version identifier.
- `Crpl`: The size of the replay protection list.
- `Features`: Features supported by the node in bit field format:
  - `0`: Relay.
  - `1`: Proxy.
  - `2`: Friend.

- 3: Low Power.

`mesh models dfu metadata comp-elem-add <Loc> <NumS> <NumV> {<SigMID>|<VndCID> <VndMID>}...`

Add element description of the Target node.

- `Loc`: Element location.
- `NumS`: Number of SIG models instantiated on the element.
- `NumV`: Number of vendor models instantiated on the element.
- `SigMID`: SIG Model ID.
- `VndCID`: Vendor model company identifier.
- `VndMID`: Vendor model identifier.

`mesh models dfu metadata comp-hash-get [<Key(16 hex)>]`

Generate a hash of the stored Composition Data to be used in metadata.

- `Key`: Optional 128-bit key to be used to generate the hash. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.

`mesh models dfu metadata encode <Major> <Minor> <Rev> <BuildNum> <Size> <CoreType> <Hash> <Elems> [<UserData>]`

Encode metadata for the DFU.

- `Major`: Major version of the firmware.
- `Minor`: Minor version of the firmware.
- `Rev`: Revision number of the firmware.
- `BuildNum`: Build number.
- `Size`: Size of the signed bin file.
- `CoreType`: New firmware core type:
  - 1: Application core.
  - 2: Network core.
  - 4: Applications specific BLOB.
- `Hash`: Hash of the composition data generated using `mesh models dfu metadata comp-hash-get` command.
- `Elems`: Number of elements on the new firmware.
- `UserData`: User data supplied with the metadata.

Segmentation and Reassembly (SAR) Configuration Client The SAR Configuration client is an optional mesh model that can be enabled through the `CONFIG_BT_MESH_SAR_CFG_CLI` configuration option. The SAR Configuration Client model is used to support the functionality of configuring the behavior of the lower transport layer of a node that supports the SAR Configuration Server model.

`mesh models sar tx-get`

Send SAR Configuration Transmitter Get message.

`mesh models sar tx-set <SegIntStep> <UniRetransCnt> <UniRetransWithoutProgCnt> <UniRetransIntStep> <UniRetransIntInc> <MultiRetransCnt> <MultiRetransInt>`

Send SAR Configuration Transmitter Set message.

- `SegIntStep`: SAR Segment Interval Step state.
- `UniRetransCnt`: SAR Unicast Retransmissions Count state.
- `UniRetransWithoutProgCnt`: SAR Unicast Retransmissions Without Progress Count state.
- `UniRetransIntStep`: SAR Unicast Retransmissions Interval Step state.
- `UniRetransIntInc`: SAR Unicast Retransmissions Interval Increment state.
- `MultiRetransCnt`: SAR Multicast Retransmissions Count state.
- `MultiRetransInt`: SAR Multicast Retransmissions Interval state.

`mesh models sar rx-get`

Send SAR Configuration Receiver Get message.

`mesh models sar rx-set <SegThresh> <AckDelayInc> <DiscardTimeout> <RxSegIntStep> <AckRetransCount>`

Send SAR Configuration Receiver Set message.

- `SegThresh`: SAR Segments Threshold state.
- `AckDelayInc`: SAR Acknowledgment Delay Increment state.
- `DiscardTimeout`: SAR Discard Timeout state.
- `RxSegIntStep`: SAR Receiver Segment Interval Step state.
- `AckRetransCount`: SAR Acknowledgment Retransmissions Count state.

Private Beacon Client The Private Beacon Client model is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_PRIV_BEACON_CLI` configuration option.

`mesh models prb priv-beacon-get`

Get the target's Private Beacon state. Possible values:

- `0x00`: The node doesn't broadcast Private beacons.
- `0x01`: The node broadcasts Private beacons.

`mesh models prb priv-beacon-set <Val(off, on)> <RandInt(10s steps)>`

Set the target's Private Beacon state.

- `Val`: Control Private Beacon state.
- `RandInt`: Random refresh interval (in 10-second steps), or 0 to keep current value.

`mesh models prb priv-gatt-proxy-get`

Get the target's Private GATT Proxy state. Possible values:

- `0x00`: The Private Proxy functionality is supported, but disabled.
- `0x01`: The Private Proxy functionality is enabled.
- `0x02`: The Private Proxy functionality is not supported.

`mesh models prb priv-gatt-proxy-set <Val(off, on)>`

Set the target's Private GATT Proxy state.

- `Val`: New Private GATT Proxy value:
  - `0x00`: Disable the Private Proxy functionality.
  - `0x01`: Enable the Private Proxy functionality.

`mesh models prb priv-node-id-get <NetKeyIdx>`

Get the target's Private Node Identity state. Possible values:

- `0x00`: The node does not adverstise with the Private Node Identity.
- `0x01`: The node advertises with the Private Node Identity.
- `0x02`: The node doesn't support advertising with the Private Node Identity.
- `NetKeyIdx`: Network index to get the Private Node Identity state of.

`mesh models prb priv-node-id-set <NetKeyIdx> <State>`

Set the target's Private Node Identity state.

- `NetKeyIdx`: Network index to set the Private Node Identity state of.
- `State`: New Private Node Identity value:
  - `0x00`: Stop advertising with the Private Node Identity.
  - `0x01`: Start advertising with the Private Node Identity.

Opcodes Aggregator Client The Opcodes Aggregator client is an optional Bluetooth Mesh model that can be enabled through the `CONFIG_BT_MESH_OP_AGG_CLI` configuration option. The Opcodes Aggregator Client model is used to support the functionality of dispatching a sequence of access layer messages to nodes supporting the Opcodes Aggregator Server model.

`mesh models opagg seq-start <ElemAddr>`

Start the Opcodes Aggregator Sequence message. This command initiates the context for aggregating messages and sets the destination address for next shell commands to `elem_addr`.

- `ElemAddr`: Element address that will process the aggregated opcodes.

`mesh models opagg seq-send`

Send the Opcodes Aggregator Sequence message. This command completes the procedure, sends the aggregated sequence message to the target node and clears the context.

`mesh models opagg seq-abort`

Abort the Opcodes Aggregator Sequence message. This command clears the Opcodes Aggregator Client context.

Remote Provisioning Client The Remote Provisioning Client is an optional Bluetooth Mesh model enabled through the `CONFIG_BT_MESH_RPR_CLI` configuration option. The Remote Provisioning Client model provides support for remote provisioning of devices into a mesh network by using the Remote Provisioning Server model.

This shell module can be used to trigger interaction between Remote Provisioning Clients and Remote Provisioning Servers on devices in a mesh network.

`mesh models rpr scan <Timeout(s)> [<UUID(1-16 hex)>]`

Start scanning for unprovisioned devices.

- `Timeout`: Scan timeout in seconds. Must be at least 1 second.
- `UUID`: Device UUID to scan for. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, all devices will be reported.

`mesh models rpr scan-ext <Timeout(s)> <UUID(1-16 hex)> [<ADType> ... ]`

Start the extended scanning for unprovisioned devices.

- `Timeout`: Scan timeout in seconds. Valid values from `BT_MESH_RPR_EXT_SCAN_TIME_MIN` to `BT_MESH_RPR_EXT_SCAN_TIME_MAX`.
- `UUID`: Device UUID to start extended scanning for. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `ADType`: List of AD types to include in the scan report. Must contain 1 to `CONFIG_BT_MESH_RPR_AD_TYPES_MAX` entries.

`mesh models rpr scan-srv [<ADType> ... ]`

Start the extended scanning for the Remote Provisioning Server.

- `ADType`: List of AD types to include in the scan report. Must contain 1 to `CONFIG_BT_MESH_RPR_AD_TYPES_MAX` entries.

`mesh models rpr scan-caps`

Get the scanning capabilities of the Remote Provisioning Server.

`mesh models rpr scan-get`

Get the current scanning state of the Remote Provisioning Server.

`mesh models rpr scan-stop`

Stop any ongoing scanning on the Remote Provisioning Server.

`mesh models rpr link-get`

Get the current link status of the Remote Provisioning Server.

`mesh models rpr link-close`

Close any open links on the Remote Provisioning Server.

`mesh models rpr provision-remote <UUID(1-16 hex)> <NetKeyIdx> <Addr>`

Provision a mesh node using the PB-Remote provisioning bearer.

- `UUID`: UUID of the unprovisioned node. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `NetKeyIdx`: Network Key Index to give to the unprovisioned node.
- `Addr`: Address to assign to remote device. If `addr` is 0, the lowest available address will be chosen.

`mesh models rpr reprovision-remote <Addr> [<CompChanged(false, true)>]`

Reprovision a mesh node using the PB-Remote provisioning bearer.

- `Addr`: Address to assign to remote device. If `addr` is 0, the lowest available address will be chosen.
- `CompChanged`: The Target node has indicated that its Composition Data has changed. Defaults to false.

`mesh models rpr instance-set <ElemIdx>`

Use the Remote Provisioning Client model instance on the specified element when using the other Remote Provisioning Client model commands.

- `ElemIdx`: The element on which to find the Remote Provisioning Client model instance to use.

`mesh models rpr instance-get-all`

Get a list of all Remote Provisioning Client model instances on the node.

Large Composition Data Client The Large Composition Data Client is an optional Bluetooth Mesh model enabled through the `CONFIG_BT_MESH_LARGE_COMP_DATA_CLI` configuration option. The Large Composition Data Client model is used to support the functionality of reading pages of Composition Data that do not fit in a Config Composition Data Status message, and reading the metadata of the model instances.

`mesh models lcd large-comp-data-get <Page> <Offset>`

Send the Large Composition Data Get message to query a portion of the Composition Data state of a node.

- `Page`: Page number of the Composition Data.
- `Offset`: Offset within the page.

`mesh models lcd models-metadata-get <Page> <Offset>`

Send the Models Metadata Get message to query a portion of a page of the Models Metadata state.

- `Page`: Page number of the Models Metadata.
- `Offset`: Offset within the page.

Bridge Configuration Client The Bridge Configuration Client model is an optional Bluetooth Mesh model that can be enabled through the `CONFIG_BT_MESH_BRG_CFG_CLI` configuration option. The model provides functionality for configuring the subnet bridge functionality of a mesh node.

`mesh models brg get`

Get the current Subnet Bridge state.

`mesh models brg set <State(disable, enable)>`

Set the Subnet Bridge state.

- `State`: Disable or enable the Subnet Bridge functionality.

`mesh models brg table-size-get`

Get the current size of the Bridging Table.

`mesh models brg table-add <Directions> <NetIdx1> <NetIdx2> <Addr1> <Addr2>`

Add an entry to the Bridging Table.

- `Directions`: Allowed directions for the bridged traffic. Valid values are:
  - `0x01`: Bridging is allowed only for messages with `Addr1` as the source address and `Addr2` as the destination address.
  - `0x02`: Bridging is allowed in both directions.
- `NetIdx1`: NetKey index of the first subnet.
- `NetIdx2`: NetKey index of the second subnet.
- `Addr1`: Address of the node in the first subnet.
- `Addr2`: Address of the node in the second subnet.

`mesh models brg table-remove <NetIdx1> <NetIdx2> <Addr1> <Addr2>`

Remove an entry from the Bridging Table.

- `NetIdx1`: NetKey index of the first subnet.
- `NetIdx2`: NetKey index of the second subnet.
- `Addr1`: Address of the node in the first subnet.
- `Addr2`: Address of the node in the second subnet.

`mesh models brg subnets-get <Filter> <NetIdx> <StartIdx>`

Get a filtered set of NetKey index pairs extracted from the Bridging Table.

- `Filter`: Filter to be applied when reporting pairs of NetKey indexes extracted from the Bridging Table. Allowed values:
  - `0x00`: Report all pairs.
  - `0x01`: Report pairs in which the NetKey index of the first subnet matches `NetIdx`.
  - `0x02`: Report pairs in which the NetKey index of the second subnet matches `NetIdx`.
  - `0x03`: Report pairs in which one of the NetKey indexes matches `NetIdx`.
- `NetIdx`: NetKey index of any of the subnets.
- `StartIdx`: Start offset in units of pairs of NetKey indexes to read.

`mesh models brg table-get <NetIdx1> <NetIdx2> <StartIdx>`

Get a list of addresses and allowed traffic directions of the Bridging Table entries.

- `NetIdx1`: NetKey index of the first subnet.
- `NetIdx2`: NetKey index of the second subnet.
- `StartIdx`: Start offset to read in units of Bridging Table state entries.

Configuration database The Configuration database is an optional mesh subsystem that can be enabled through the `CONFIG_BT_MESH_CDB` configuration option. The Configuration database is only available on provisioner devices, and allows them to store all information about the mesh network. To avoid conflicts, there should only be one mesh node in the network with the Configuration database enabled. This node is the Configurator, and is responsible for adding new nodes to the network and configuring them.

`mesh cdb create [NetKey(1-16 hex)]`

Create a Configuration database.

- `NetKey`: Optional network key value of the primary network key (NetKeyIndex=0). Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. Defaults to the default key value if omitted.

`mesh cdb clear`

Clear all data from the Configuration database.

`mesh cdb show`

Show all data in the Configuration database.

`mesh cdb node-add <UUID(1-16 hex)> <Addr> <ElemCnt> <NetKeyIdx> [DevKey(1-16 hex)]`

Manually add a mesh node to the configuration database. Note that devices provisioned with `mesh provision` and `mesh provision-adv` will be added automatically if the Configuration Database is enabled and created.

- `UUID`: 128-bit hexadecimal UUID of the node. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest.
- `Addr`: Unicast address of the node, or 0 to automatically choose the lowest available address.
- `ElemCnt`: Number of elements on the node.
- `NetKeyIdx`: The network key the node was provisioned with.
- `DevKey`: Optional 128-bit device key value for the device. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, a random value will be generated.

`mesh cdb node-del <Addr>`

Delete a mesh node from the Configuration database. If possible, the node should be reset with `mesh reset` before it is deleted from the Configuration database, to avoid unexpected behavior and uncontrolled access to the network.

- `Addr` Address of the node to delete.

`mesh cdb subnet-add <NetKeyIdx> [<NetKey(1-16 hex)>]`

Add a network key to the Configuration database. The network key can later be passed to mesh nodes in the network. Note that adding a key to the Configuration database does not automatically add it to the local node's list of known network keys.

- `NetKeyIdx`: Key index of the network key to add.

- NetKey: Optional 128-bit network key value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, a random value will be generated.

`mesh cdb subnet-del <NetKeyIdx>`

Delete a network key from the Configuration database.

- NetKeyIdx: Key index of the network key to delete.

`mesh cdb app-key-add <NetKeyIdx> <AppKeyIdx> [<AppKey(1-16 hex)>]`

Add an application key to the Configuration database. The application key can later be passed to mesh nodes in the network. Note that adding a key to the Configuration database does not automatically add it to the local node's list of known application keys.

- NetKeyIdx: Network key index the application key is bound to.
- AppKeyIdx: Key index of the application key to add.
- AppKey: Optional 128-bit application key value. Providing a hex-string shorter than 16 bytes will populate the N most significant bytes of the array and zero-pad the rest. If omitted, a random value will be generated.

`mesh cdb app-key-del <AppKeyIdx>`

Delete an application key from the Configuration database.

- AppKeyIdx: Key index of the application key to delete.

On-Demand Private GATT Proxy Client The On-Demand Private GATT Proxy Client model is an optional mesh subsystem that can be enabled through the CONFIG_BT_MESH_OD_PRIV_PROXY_CLI configuration option.

`mesh models od_priv_proxy od-priv-gatt-proxy [Dur(s)]`

Set the On-Demand Private GATT Proxy state on active target, or fetch the value of this state from it.

- Dur: If given, set the state of On-Demand Private GATT Proxy to this value in seconds. Fetch this value otherwise.

Solicitation PDU RPL Client The Solicitation PDU RPL Client model is an optional mesh subsystem that can be enabled through the CONFIG_BT_MESH_SOL_PDU_RPL_CLI configuration option.

`mesh models sol_pdu_rpl sol-pdu-rpl-clear <RngStart> <Ackd> [RngLen]`

Clear active target's solicitation replay protection list (SRPL) in given range of solicitation source (SSRC) addresses.

- RngStart: Start address of the SSRC range.
- Ackd: This argument decides on whether an acknowledged or unacknowledged message will be sent.
- RngLen: Range length for the SSRC addresses to be cleared from the solicitiation RPL list. This parameter is optional; if absent, only a single SSRC address will be cleared.

Statistic The statistic is an optional mesh module that can be enabled through the CONFIG_BT_MESH_STATISTIC configuration option.

`mesh stat adv_get`

Get the frame statistic. The command prints numbers of received frames, as well as numbers of planned and succeeded transmission attempts.

`mesh stat adv_clear`

Clear frame statistics collected before.

`mesh stat lpn_get`

Get the measured LPN friendship timing parameters. The command prints ReceiveDelay and ReceiveWindow values derived from timestamps taken during poll cycles. Requires `CONFIG_BT_MESH_LOW_POWER`.

`mesh stat lpn_clear`

Clear LPN timing statistics collected before. Requires `CONFIG_BT_MESH_LOW_POWER`.

Subnet Bridge With Bluetooth Mesh Protocol Specification version 1.1, the Bluetooth Mesh Subnet Bridge feature was introduced. This feature allows mesh networks to use subnets for area isolation, but to also allow communication between specific devices in different, adjacent subnets without compromising security.

The Bluetooth Mesh Subnet Bridge feature makes it possible for selected nodes in the network to act as Subnet Bridges, allowing controlled communication by relaying messages between nodes in adjacent subnets.

The Subnet Bridge feature includes two models:

- *Bridge Configuration Server*
- *Bridge Configuration Client*

The Bridge Configuration Server model is mandatory for supporting the Subnet Bridge feature. The Bridge Configuration Client model is optional and allows nodes to configure the Subnet Bridge on other nodes. These models define the necessary states, messages, and behaviors for configuring and managing the Subnet Bridge feature.

The configuration and management of the Subnet Bridge feature are handled using the Bridge Configuration Server and Client models. A node implementing the Bridge Configuration Client model can act as a *Configuration Manager* for the Subnet Bridge feature.

Concepts To get a better understanding of the Subnet Bridge feature and its capabilities, an overview of a few concepts is needed.

Subnet A subnet is a group of nodes within a mesh network that share a common network key, enabling them to communicate securely at the network layer. Each subnet operates independently, with nodes exchanging messages exclusively within that group. A node may belong to multiple subnets at the same time.

Subnet Bridge node A Subnet Bridge node is a node in a Bluetooth Mesh network that belongs to multiple subnets and has a Subnet Bridge functionality enabled. Subnet bridging can be performed only by such a node. The Subnet Bridge node connects the subnets, and allows communication between them by relaying messages across the subnet groups.

The Subnet Bridge node has a primary subnet, based on the primary NetKey, which handles the IV Update procedure and propagates updates to other subnets. The secondary subnets, on which messages are relayed, are referred to as *bridged subnets*.

Bridging Table The Bridging Table contains the entries for the subnets that are bridged by the node, and is managed by the Bridge Configuration Server model.

The maximum number of entries in the bridging table is defined by the `CONFIG_BT_MESH_BRG_TABLE_ITEMS_MAX` option, which defaults to the minimum value of 16, with a maximum possible size of 255.

Enabling or disabling the Subnet Bridge feature The Bridge Configuration Client (or Configuration Manager) can enable or disable the Subnet Bridge feature on a node by sending a **Subnet Bridge Set** message to the Bridge Configuration Server model on the target node, using the `bt_mesh_brg_cfg_cli_set()` function.

Adding or removing subnets The Bridge Configuration Client can add or remove an entry from the Bridging Table by sending a **Bridging Table Add** or **Bridging Table Remove** message to the Bridge Configuration Server model on the target node, calling the `bt_mesh_brg_cfg_cli_table_add()` or `bt_mesh_brg_cfg_cli_table_remove()` functions.

Subnet Bridge states The Subnet Bridge has the following states:

- *Subnet Bridge*: This state indicates whether the Subnet Bridge feature is enabled or disabled on the node. The Bridge Configuration Client can retrieve this information by sending a **Subnet Bridge Get** message to the Bridge Configuration Server using the `bt_mesh_brg_cfg_cli_get()` function.
- *Bridging Table*: This state holds the bridging table. The Client can request a list of entries from a Bridging Table by sending a **Bridging Table Get** message to the target node using the `bt_mesh_brg_cfg_cli_table_get()` function.

  The Client can get a list of subnets currently bridged by a Subnet Bridge by sending a **Bridged Subnets Get** message to the target Server by calling the `bt_mesh_brg_cfg_cli_subnets_get()` function.
- *Bridging Table Size*: This state reports the maximum number of entries the Bridging Table can store. The Client can retrieve this information by sending a **Bridging Table Size Get** message using the `bt_mesh_brg_cfg_cli_table_size_get()` function. This is a read-only state.

Subnet bridging and replay protection The Subnet Bridge feature enables message relaying between subnets and requires effective replay protection to ensure network security. Key considerations to take into account are described below.

Relay buffer considerations When a message is relayed between subnets by a Subnet Bridge, it is allocated from the relay buffer pool. The number of relay buffers are configurable using the `CONFIG_BT_MESH_RELAY_BUF_COUNT` Kconfig option.

When `CONFIG_BT_MESH_ADV_EXT` is enabled, messages will be transmitted using the relay advertising sets. The number of advertising sets are configurable using the `CONFIG_BT_MESH_RELAY_ADV_SETS` Kconfig option.

Both the relay buffer pool and advertising sets can be used even if the relay feature `CONFIG_BT_MESH_RELAY` is disabled.

Replay protection and Bridging Table A Subnet Bridge node must implement replay protection for all Access and Transport Control messages sent to bridged subnets.

The Replay Protection List (RPL) works with the Bridging Table to ensure security:

- The Subnet Bridge stores the latest IVISeq for each source address authorized to send messages to bridged subnets.

- Messages with an IVISeq less than or equal to the stored value are discarded, while valid messages update the stored IVISeq before being relayed.

To ensure proper operation, it is important that the RPL and Bridging Table are synchronized, as every bridged message must pass through the replay protection mechanism before being relayed.

> **Note**
>
> The RPL size should scale with the Bridging Table. As the number of bridged subnets grows, more source addresses and IVISeq values must be tracked, requiring a larger RPL to maintain effective replay protection.

Subnet Bridge and Directed Forwarding Bluetooth Mesh Directed Forwarding (MDF) enables efficient routing between nodes across subnets by optimizing relay paths. While MDF can enhance Subnet Bridging by handling path discovery and forwarding, the current implementation does not support this feature.

API reference This section contains types and defines common to the Bridge Configuration models.

Bridge Configuration common header

Configuration Database (CDB)

- *Overview*
  - *Key features*
- *Configuration options*
- *Data Structures*
  - *Node structure*
  - *Subnet structure*
  - *Application Key structure*
  - *Main CDB structure*
- *Usage scenarios*
  - *Provisioner initialization*
  - *Provisioning and node allocation*
  - *Subnet and key management*
  - *Subnet lookup and removal*
  - *Setting up application keys*
  - *Application keys lookup and removal*
  - *Node configuration*
  - *Iterating over nodes*
  - *Node lookup and removal*
  - *IV Index management*

  - *Working with device keys*
  - *Clearing the CDB*
- *Best practices*
- *Common pitfalls*
- *API reference*
- *Related documentation*

Overview The Configuration Database (CDB) is the subsystem for Bluetooth® Mesh provisioner devices. It stores and manages information about the mesh network, including provisioned nodes, network keys, and application keys. The CDB enables a provisioner to track the network topology, assign addresses, and configure nodes systematically.

> **Note**
>
> The CDB implementation has proprietary design and does not follow the Bluetooth SIG specification.

The CDB is enabled with the `CONFIG_BT_MESH_CDB` configuration option and is typically used only on the provisioner device in the network. There should be only one device in the mesh network with CDB enabled to avoid conflicts.

Key features

- **Persistent Storage**: All CDB data is stored persistently when `CONFIG_BT_SETTINGS` is enabled, allowing the provisioner to recover network state across reboots.
- **Address Management**: Automatic allocation and tracking of unicast addresses for provisioned nodes.
- **Key Management**: Storage and synchronization of network keys and application keys.
- **Node Tracking**: Maintains information about each provisioned node including UUID, address, element count, device key, and configuration status.
- **IV Index Management**: Tracks the network's IV Index and IV Update state.

Configuration options The following Kconfig options control the CDB capacity:

- `CONFIG_BT_MESH_CDB_NODE_COUNT` - Maximum number of nodes the CDB can handle.
- `CONFIG_BT_MESH_CDB_SUBNET_COUNT` - Maximum number of subnets the CDB can handle.
- `CONFIG_BT_MESH_CDB_APP_KEY_COUNT` - Maximum number of application keys the CDB can handle.

Additional options:

- `CONFIG_BT_MESH_CDB_KEY_SYNC` - Enables automatic synchronization between mesh keys on the provisioner node and the CDB keys. If option is enabled then operations with keys performed through the *Configuration Server* are automatically mirrored to the CDB.

Data Structures The CDB was designed to handle data collected in several data structures. The structures represent nodes, subnets, application keys, and the overall database state.

Node structure

```
struct bt_mesh_cdb_node {
        /** Node UUID */
        uint8_t  uuid[16];
        /** Primary element address */
        uint16_t addr;
        /** Network key index used during provisioning */
        uint16_t net_idx;
        /** Number of elements */
        uint8_t  num_elem;
        /** Device key */
        struct bt_mesh_key dev_key;
        /** Node flags */
        ATOMIC_DEFINE(flags, BT_MESH_CDB_NODE_FLAG_COUNT);
};
```

Node flags:

- `BT_MESH_CDB_NODE_CONFIGURED` - Set when the node has been configured with keys and bindings.

Subnet structure

```
struct bt_mesh_cdb_subnet {
        /** Network key index */
        uint16_t net_idx;
        /** Key Refresh phase */
        uint8_t kr_phase;
        /** Old and new keys for Key Refresh */
        struct {
                struct bt_mesh_key net_key;
        } keys[2];
};
```

Application Key structure

```
struct bt_mesh_cdb_app_key {
        /** Bound network key index */
        uint16_t net_idx;
        /** Application key index */
        uint16_t app_idx;
        /** Old and new keys for Key Refresh */
        struct {
                struct bt_mesh_key app_key;
        } keys[2];
};
```

Main CDB structure

```
struct bt_mesh_cdb {
        /** Network IV Index */
        uint32_t iv_index;
        /** Lowest available unicast address for provisioning */
        uint16_t lowest_avail_addr;
        /** Flags for CDB state (private) */
        ATOMIC_DEFINE(flags, BT_MESH_CDB_FLAG_COUNT);
        /** Nodes in the mesh network */
```

(continues on next page)

(continued from previous page)

```
        struct bt_mesh_cdb_node nodes[NODE_COUNT];
        /** Subnets in the mesh network */
        struct bt_mesh_cdb_subnet subnets[SUBNET_COUNT];
        /** Application keys in the mesh network */
        struct bt_mesh_cdb_app_key app_keys[APP_KEY_COUNT];
};

/** The global CDB instance is accessible through this variable. */
extern struct bt_mesh_cdb bt_mesh_cdb;
```

Usage scenarios The following patterns demonstrate common CDB usage scenarios:

- *Provisioner initialization* - Creating and initializing the CDB
- *Provisioning and node allocation* - Managing node provisioning and addresses
- *Subnet and key management* - Managing network keys and subnets
- *Subnet lookup and removal* - Finding and removing subnets
- *Setting up application keys* - Adding application keys to the CDB
- *Application keys lookup and removal* - Finding and removing application keys
- *Node configuration* - Configuring provisioned nodes
- *Iterating over nodes* - Processing nodes with callbacks
- *Node lookup and removal* - Finding and removing nodes
- *IV Index management* - Managing IV Index updates
- *Working with device keys* - Importing/exporting device keys
- *Clearing the CDB* - Resetting the database

> **Note**
>
> The provided below source code of scenarios should be treated as conceptual patterns rather than exact implementations. They require adaptation to fit into an application context and testing. Please read the CDB API description for correct usage.

Provisioner initialization On the embedded provisioner node, the CDB must be initialized before the mesh stack is provisioned and the application can use it for key storage, node tracking, and configuration. This step establishes the primary network key, the initial subnet, and the starting IV Index and address space, so that all later CDB operations work on a well-defined network context and can be persisted across reboots.

A typical provisioner initializes the CDB during startup:

```
#include <zephyr/bluetooth/mesh.h>

static int provisioner_init(void)
{
    uint8_t net_key[16];
    int err;

    /* Initialize Bluetooth Mesh */
    err = bt_mesh_init(&prov, &comp);
    if (err) {
```

(continues on next page)

(continued from previous page)

```
        return err;
    }

    /* Load settings if persistent storage is enabled */
    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        settings_load();
    }

    /* Generate or use predefined network key */
    bt_rand(net_key, 16);

    /* Create CDB with primary network key */
    err = bt_mesh_cdb_create(net_key);
    if (err == -EALREADY) {
        printk("Using stored CDB\n");
    } else if (err) {
        printk("Failed to create CDB (err %d)\n", err);
        return err;
    } else {
        printk("Created new CDB\n");
    }

    return 0;
}
```

Key points:

- Call the `bt_mesh_cdb_create()` function with the primary network key.
- Returns error code if CDB already exists (loaded from persistent storage).
- The function automatically creates a subnet with `NetIdx = 0`.
- Sets the IV Index to `0` and lowest available address to `1`.

Provisioning and node allocation When you add new devices to the mesh, their addresses, device keys, and basic metadata must be recorded consistently in the provisioner's database. Automatic node allocation during PB-ADV/PB-GATT provisioning keeps CDB and the actual network in sync. When provisioning a new device, the CDB automatically allocates the node during the provisioning process.

However, you can also manually allocate nodes or check allocation. Manual node allocation is useful for advanced scenarios such as importing nodes from an external source, pre-allocating address ranges, or recovering a network from known information.

```
/* Provisioning callback - node is automatically added to CDB */
static void node_added(uint16_t idx, uint8_t uuid[16], uint16_t addr,
                       uint8_t num_elem)
{
    printk("Node added: addr=0x%04x, elements=%d\n", addr, num_elem);
    /* The CDB node is created automatically by the provisioning subsystem */
}

static const struct bt_mesh_prov prov = {
    .uuid = dev_uuid,
    .node_added = node_added,
    /* ... other callbacks ... */
};
```

(continues on next page)

(continued from previous page)

```
/* Manual node allocation (if needed) */
static struct bt_mesh_cdb_node *allocate_node(const uint8_t uuid[16],
                                              uint8_t num_elem)
{
    struct bt_mesh_cdb_node *node;
    uint16_t addr;

    /* Get free address or specify one (0 = auto-allocate) */
    addr = bt_mesh_cdb_free_addr_get(num_elem);
    if (addr == BT_MESH_ADDR_UNASSIGNED) {
        printk("No free addresses available\n");
        return NULL;
    }

    /* Allocate node in CDB */
    node = bt_mesh_cdb_node_alloc(uuid, addr, num_elem, net_idx);
    if (node == NULL) {
        printk("Failed to allocate node\n");
        return NULL;
    }

    return node;
}
```

Key points:

- During normal PB-ADV/PB-GATT provisioning, nodes are automatically added to the CDB.
- The `bt_mesh_cdb_node_alloc()` function creates a CDB entry with the specified parameters.
- Pass `BT_MESH_ADDR_UNASSIGNED` as the address to let the CDB auto-assign the lowest available address.
- The `bt_mesh_cdb_free_addr_get()` function finds a free address range for a given element count.
- Address allocation checks for conflicts and ensures the unicast address validity.

Subnet and key management Large or segmented mesh networks often use multiple subnets to separate traffic domains, enable secure migration, or support the Key Refresh procedure. Managing subnets and their network keys in the CDB allows the provisioner to create new subnets, rotate keys, and derive the correct flags and IV Index when provisioning and configuring nodes.

Manage network keys and subnets in the CDB:

```
/* Add a new subnet */
static int add_subnet(uint16_t net_idx)
{
    struct bt_mesh_cdb_subnet *sub;
    uint8_t net_key[16];
    int err;

    /* Allocate subnet in CDB */
    sub = bt_mesh_cdb_subnet_alloc(net_idx);
    if (sub == NULL) {
        printk("Failed to allocate subnet\n");
        return -ENOMEM;
```

(continues on next page)

(continued from previous page)

```
    }

    /* Generate network key */
    bt_rand(net_key, 16);

    /* Import key value */
    err = bt_mesh_cdb_subnet_key_import(sub, 0, net_key);
    if (err) {
        bt_mesh_cdb_subnet_del(sub, false);
        return err;
    }

    /* Store subnet */
    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        bt_mesh_cdb_subnet_store(sub);
    }

    return 0;
}

/* Get subnet for provisioning */
static int get_subnet_flags(uint16_t net_idx)
{
    struct bt_mesh_cdb_subnet *sub;
    uint8_t flags;

    sub = bt_mesh_cdb_subnet_get(net_idx);
    if (sub == NULL) {
        return -ENOENT;
    }

    flags = bt_mesh_cdb_subnet_flags(sub);
    printk("Subnet flags: KR=%d IVU=%d\n",
           !!(flags & BT_MESH_NET_FLAG_KR),
           !!(flags & BT_MESH_NET_FLAG_IVU));

    return 0;
}
```

Key points:

- The `bt_mesh_cdb_subnet_alloc()` function allocates a subnet by NetIdx.
- The `bt_mesh_cdb_subnet_key_import()` function sets the actual key value.
- The `bt_mesh_cdb_subnet_flags()` function returns flags needed for provisioning data.
- The flags include Key Refresh and IV Update state from the CDB.

Subnet lookup and removal At some point you may need to inspect an existing subnet (for diagnostics or tooling) or remove it entirely (for example when decommissioning a segment, rotating to a new subnet, or cleaning up after experimentation). The CDB provides lookup and deletion helpers so that the provisioner can keep its view of active subnets consistent with what nodes actually use.

Find and manage subnets in the CDB:

```
/* Get subnet by NetIdx */
static void lookup_subnet(uint16_t net_idx)
{
    struct bt_mesh_cdb_subnet *sub;

    sub = bt_mesh_cdb_subnet_get(net_idx);
    if (sub == NULL) {
        printk("Subnet 0x%03x not found\n", net_idx);
        return;
    }

    printk("Subnet 0x%03x: kr_phase=%d\n", net_idx, sub->kr_phase);
}

/* Remove a subnet from the CDB */
static void remove_subnet(uint16_t net_idx)
{
    struct bt_mesh_cdb_subnet *sub;

    sub = bt_mesh_cdb_subnet_get(net_idx);
    if (sub == NULL) {
        printk("Subnet not found\n");
        return;
    }

    /* First, remove the subnet from all nodes in the network */
    /* ... send NetKey Delete to all nodes using Config Client ... */

    /* Delete subnet from CDB */
    bt_mesh_cdb_subnet_del(sub, true);

    printk("Subnet 0x%03x removed\n", net_idx);
}

/* Export subnet key for external use */
static int export_subnet_key(uint16_t net_idx, int key_idx)
{
    struct bt_mesh_cdb_subnet *sub;
    uint8_t net_key[16];
    int err;

    sub = bt_mesh_cdb_subnet_get(net_idx);
    if (sub == NULL) {
        return -ENOENT;
    }

    err = bt_mesh_cdb_subnet_key_export(sub, key_idx, net_key);
    if (err) {
        printk("Failed to export subnet key (err %d)\n", err);
        return err;
    }

    printk("NetKey[%d] for subnet 0x%03x: ", key_idx, net_idx);
    for (int i = 0; i < 16; i++) {
        printk("%02x", net_key[i]);
    }
    printk("\n");
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Key points:

- The `bt_mesh_cdb_subnet_get()` function retrieves a subnet by its NetKeyIndex.
- Pass `true` to the `bt_mesh_cdb_subnet_del()` function to clear persistent storage.
- Always remove the subnet from all nodes before deleting it from the CDB.
- Use the `bt_mesh_cdb_subnet_key_export()` function to retrieve key material securely.
- The `key_idx` parameter (`0` or `1`) selects between old and new keys during the Key Refresh procedure.

Setting up application keys Application keys define which application traffic can be encrypted and where it can be bound. After the CDB is initialized, the provisioner must create one or more application keys so that it can bind models on nodes and enable real application-level communication (for example, lighting or sensor data).

After creating the CDB, add application keys for node configuration:

```
static void setup_cdb_keys(void)
{
    struct bt_mesh_cdb_app_key *key;
    uint8_t app_key[16];
    int err;

    /* Allocate application key in CDB */
    key = bt_mesh_cdb_app_key_alloc(net_idx, app_idx);
    if (key == NULL) {
        printk("Failed to allocate app-key\n");
        return;
    }

    /* Generate random key value */
    bt_rand(app_key, 16);

    /* Import the key into CDB */
    err = bt_mesh_cdb_app_key_import(key, 0, app_key);
    if (err) {
        printk("Failed to import appkey (err %d)\n", err);
        return;
    }

    /* Store to persistent storage */
    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        bt_mesh_cdb_app_key_store(key);
    }
}
```

Key points:

- The `bt_mesh_cdb_app_key_alloc()` function allocates a slot in the CDB.
- The `bt_mesh_cdb_app_key_import()` function sets the actual key value.
- The second parameter to import (`0`) is the key index for the Key Refresh procedure (`0` = current key).

- Always store keys after creation when using persistent storage.

Application keys lookup and removal Over time, you may need to inspect which network key an application key is bound to, rotate or revoke application keys, or clean up keys that are no longer used. Proper lookup and removal through the CDB helps avoid dangling bindings and ensures that nodes and the provisioner share the same view of usable application keys.

Find and manage application keys in the CDB:

```
/* Get application key by AppIdx */
static void lookup_app_key(uint16_t app_idx)
{
    struct bt_mesh_cdb_app_key *key;

    key = bt_mesh_cdb_app_key_get(app_idx);
    if (key == NULL) {
        printk("AppKey 0x%03x not found\n", app_idx);
        return;
    }

    printk("AppKey 0x%03x: bound to NetIdx 0x%03x\n",
           app_idx, key->net_idx);
}

/* Remove an application key from the CDB */
static void remove_app_key(uint16_t app_idx)
{
    struct bt_mesh_cdb_app_key *key;

    key = bt_mesh_cdb_app_key_get(app_idx);
    if (key == NULL) {
        printk("AppKey not found\n");
        return;
    }

    /* First, remove the app key from all nodes in the network */
    /* ... send AppKey Delete to all nodes using Config Client ... */

    /* Delete app key from CDB */
    bt_mesh_cdb_app_key_del(key, true);

    printk("AppKey 0x%03x removed\n", app_idx);
}

/* Export application key for external use */
static int export_app_key(uint16_t app_idx, int key_idx)
{
    struct bt_mesh_cdb_app_key *key;
    uint8_t app_key[16];
    int err;

    key = bt_mesh_cdb_app_key_get(app_idx);
    if (key == NULL) {
        return -ENOENT;
    }

    err = bt_mesh_cdb_app_key_export(key, key_idx, app_key);
    if (err) {
```

(continues on next page)

(continued from previous page)

```
        printk("Failed to export app key (err %d)\n", err);
        return err;
    }

    printk("AppKey[%d] 0x%03x: ", key_idx, app_idx);
    for (int i = 0; i < 16; i++) {
        printk("%02x", app_key[i]);
    }
    printk("\n");

    return 0;
}

/* Update application key binding */
static int update_app_key_binding(uint16_t app_idx, uint16_t new_net_idx)
{
    struct bt_mesh_cdb_app_key *key;

    key = bt_mesh_cdb_app_key_get(app_idx);
    if (key == NULL) {
        return -ENOENT;
    }

    /* Verify the new subnet exists */
    if (bt_mesh_cdb_subnet_get(new_net_idx) == NULL) {
        printk("Target subnet 0x%03x not found\n", new_net_idx);
        return -ENOENT;
    }

    key->net_idx = new_net_idx;

    /* Store updated binding */
    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        bt_mesh_cdb_app_key_store(key);
    }

    printk("AppKey 0x%03x rebound to NetIdx 0x%03x\n",
           app_idx, new_net_idx);

    return 0;
}
```

Key points:

- The `bt_mesh_cdb_app_key_get()` function retrieves an application key by its AppKeyIndex.
- Pass `true` to the `bt_mesh_cdb_app_key_del()` function to clear persistent storage.
- Always remove the app key from all nodes before deleting it from the CDB.
- Use the `bt_mesh_cdb_app_key_export()` function to retrieve key material securely.
- The `key_idx` parameter (`0` or `1`) selects between old and new keys during the Key Refresh procedure.
- Application keys are bound to a network key through the `net_idx` field.

Node configuration Provisioning only adds a node to the network; it does not make it participate in your application. After provisioning, you must configure nodes—add network/app keys,

bind models, and set subscriptions/publications. The CDB stores the keys and configuration status so the provisioner can resume configuration after a reboot and avoid re-configuring already finished nodes.

After provisioning, configure the node by adding keys and binding models:

```
static void configure_node(struct bt_mesh_cdb_node *node)
{
    NET_BUF_SIMPLE_DEFINE(buf, BT_MESH_RX_SDU_MAX);
    struct bt_mesh_cdb_app_key *key;
    uint8_t app_key[16];
    uint8_t status;
    int err;

    printk("Configuring node 0x%04x\n", node->addr);

    /* Get application key from CDB */
    key = bt_mesh_cdb_app_key_get(app_idx);
    if (key == NULL) {
        printk("App-key not found\n");
        return;
    }

    /* Export key value from CDB */
    err = bt_mesh_cdb_app_key_export(key, 0, app_key);
    if (err) {
        printk("Failed to export key (err %d)\n", err);
        return;
    }

    /* Add application key to the node */
    err = bt_mesh_cfg_cli_app_key_add(net_idx, node->addr,
                                      net_idx, app_idx,
                                      app_key, &status);
    if (err || status) {
        printk("Failed to add app-key (err %d, status %d)\n",
               err, status);
        return;
    }

    /* Get composition data and bind models */
    err = bt_mesh_cfg_cli_comp_data_get(net_idx, node->addr, 0,
                                        &status, &buf);
    if (err || status) {
        printk("Failed to get composition data\n");
        return;
    }

    /* Parse composition and bind models to app key */
    /* ... model binding code ... */

    /* Mark node as configured */
    atomic_set_bit(node->flags, BT_MESH_CDB_NODE_CONFIGURED);

    /* Persist configuration status */
    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        bt_mesh_cdb_node_store(node);
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Key points:

- Use the `bt_mesh_cdb_app_key_get()` and the `bt_mesh_cdb_app_key_export()` functions to retrieve keys.
- The Configuration Client model is used to configure remote nodes.
- Set the `BT_MESH_CDB_NODE_CONFIGURED` flag after successful configuration.
- Always call the `bt_mesh_cdb_node_store()` function to persist the configured state.

Iterating over nodes Many management tasks involve acting on all (or a filtered subset of) nodes: checking which ones still need configuration, generating statistics, or performing bulk operations. The CDB iterator lets the provisioner traverse all known nodes in a safe, abstract way without relying on internal storage details.

The CDB provides an iterator to process all allocated nodes:

```
/* Check for unconfigured nodes */
static uint8_t check_unconfigured(struct bt_mesh_cdb_node *node,
                                  void *user_data)
{
    if (!atomic_test_bit(node->flags, BT_MESH_CDB_NODE_CONFIGURED)) {
        printk("Node 0x%04x needs configuration\n", node->addr);
        configure_node(node);
    }

    return BT_MESH_CDB_ITER_CONTINUE;
}

static void process_nodes(void)
{
    bt_mesh_cdb_node_foreach(check_unconfigured, NULL);
}

/* Example: Count nodes on a specific subnet */
static uint8_t count_subnet_nodes(struct bt_mesh_cdb_node *node,
                                  void *user_data)
{
    uint16_t *net_idx = user_data;
    static int count = 0;

    if (node->net_idx == *net_idx) {
        count++;
    }

    return BT_MESH_CDB_ITER_CONTINUE;
}
```

Key points:

- The `bt_mesh_cdb_node_foreach()` function calls the callback for each allocated node.
- Return `BT_MESH_CDB_ITER_CONTINUE` to continue iteration.
- Return `BT_MESH_CDB_ITER_STOP` to stop iteration early.
- The callback is only called for nodes with `addr != BT_MESH_ADDR_UNASSIGNED`.

Node lookup and removal Sometimes a single node must be inspected (for example, when debugging) or removed from the network (for example, when it is physically decommissioned or replaced). Using CDB node lookup and deletion keeps the logical view of the mesh consistent with the actual devices and prevents address reuse conflicts.

Find and manage nodes in the CDB:

```
/* Get node by address */
static void lookup_node(uint16_t addr)
{
    struct bt_mesh_cdb_node *node;

    node = bt_mesh_cdb_node_get(addr);
    if (node == NULL) {
        printk("Node 0x%04x not found\n", addr);
        return;
    }

    printk("Node 0x%04x: %d elements, net_idx=0x%03x\n",
           node->addr, node->num_elem, node->net_idx);
}

/* Remove a node from the network */
static void remove_node(uint16_t addr)
{
    struct bt_mesh_cdb_node *node;

    node = bt_mesh_cdb_node_get(addr);
    if (node == NULL) {
        printk("Node not found\n");
        return;
    }

    /* First, send Node Reset to the device (recommended) */
    /* ... send reset command using Config Client ... */

    /* Delete node from CDB */
    bt_mesh_cdb_node_del(node, true);

    printk("Node 0x%04x removed\n", addr);
}
```

Key points:

- The `bt_mesh_cdb_node_get()` function searches by element address (works for any element address).
- Always send a `Node Reset` command before removing a node to avoid orphaned devices.
- Pass `true` to the `bt_mesh_cdb_node_del()` function to clear persistent storage.
- Deleting a node might update the `lowest_avail_addr` if appropriate (please read CDB API for details).

IV Index management The IV Index is a core part of Bluetooth Mesh security and replay protection. When IV Update procedures occur, the provisioner must update its stored IV Index so that future provisioning and configuration use the correct value and address space. Storing this in the CDB keeps the network state coherent across reboots and between provisioning actions.

Update the network's IV Index:

```
/* Update IV Index when IV Update procedure occurs */
static void handle_iv_update(uint32_t iv_index, bool iv_update)
{
    bt_mesh_cdb_iv_update(iv_index, iv_update);

    printk("IV Index updated: %u (IV Update: %s)\n",
           iv_index, iv_update ? "in progress" : "normal");
}

/* The IV Index is automatically used during provisioning */
static void provision_with_current_iv(void)
{
    /* The provisioner automatically uses bt_mesh_cdb.iv_index
     * and bt_mesh_cdb_subnet_flags() when sending provisioning data
     */
}
```

Key points:

- The `bt_mesh_cdb_iv_update()` function updates both the IV Index and IV Update flag.
- The CDB automatically resets `lowest_avail_addr` during IV Index updates.
- The provisioner subsystem automatically uses CDB's IV Index for new nodes.

Working with device keys Device keys are sensitive information and often handled through secure key storage (for example, PSA crypto). The CDB's key import/export APIs abstract how keys are stored, allowing the application to use device keys when needed (for configuration or external tooling) without breaking the security model or relying on direct pointers into key storage.

Import and export device keys using the key management API:

```
/* Import a known device key */
static int import_dev_key(struct bt_mesh_cdb_node *node,
                          const uint8_t dev_key[16])
{
    int err;

    err = bt_mesh_cdb_node_key_import(node, dev_key);
    if (err) {
        printk("Failed to import device key (err %d)\n", err);
        return err;
    }

    /* Store updated node */
    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        bt_mesh_cdb_node_store(node);
    }

    return 0;
}

/* Export device key for external use */
static int export_dev_key(struct bt_mesh_cdb_node *node)
{
    uint8_t dev_key[16];
    int err;
```

(continues on next page)

(continued from previous page)

```
    err = bt_mesh_cdb_node_key_export(node, dev_key);
    if (err) {
        printk("Failed to export device key (err %d)\n", err);
        return err;
    }

    /* Use the exported key */
    printk("Device key: ");
    for (int i = 0; i < 16; i++) {
        printk("%02x", dev_key[i]);
    }
    printk("\n");

    return 0;
}
```

Key points:

- Always use import/export functions when working with keys (required for PSA crypto).
- Keys are stored securely and are not be directly accessible through pointers.
- The provisioner automatically generates and imports device keys during provisioning.

Clearing the CDB For factory reset, testing, or starting a completely new network, you may need to wipe all provisioning and configuration data. Clearing the CDB resets the logical network state so you can safely recreate it from scratch, while avoiding inconsistencies with stale keys or node entries.

Reset the entire configuration database:

```
static void reset_network(void)
{
    printk("Clearing CDB...\n");

    /* Clear all CDB data */
    bt_mesh_cdb_clear();

    printk("CDB cleared\n");
}
```

Key points:

- The `bt_mesh_cdb_clear()` function removes all nodes, subnets, and app keys.
- Clears persistent storage if the `CONFIG_BT_SETTINGS` Kconfig option is enabled.
- Marks the CDB as invalid (must call the `bt_mesh_cdb_create()` function to use again).

Best practices

- **Error handling** - Check return values from all CDB functions, especially allocation functions which can fail when capacity is reached.
- **Address management** - Let the CDB manage address allocation by passing `0` to the `bt_mesh_cdb_node_alloc()` function unless you have specific addressing requirements.
- **Configuration tracking** - Always set the `BT_MESH_CDB_NODE_CONFIGURED` flag and store the node after successful configuration to avoid reconfiguring nodes unnecessarily.
- **Key security** - Never log or expose keys in production code. Use import/export functions to handle key material securely.

- **Iteration safety** - Do not modify the CDB (add/remove nodes) during `bt_mesh_cdb_node_foreach()` iteration. Collect addresses first, then modify.

Common pitfalls

CDB synchronization among multiple provisioners
If there are multiple provisioners in the network, ensure they synchronize their CDBs to avoid conflicts. Use application-level mechanisms to share CDB among multiple provisioners. There is no built-in mechanism for CDB synchronization.

Forgetting to store
CDB modifications are not automatically persisted. Always call the appropriate `_store()` function after making changes when using persistent storage.

Incorrect key index
When importing/exporting keys, the key index parameter (`0` or `1`) refers to the position in the Key Refresh key array, not the NetIdx or AppIdx.

Address conflicts
If manually specifying addresses, ensure they do not conflict with existing nodes. Use the `bt_mesh_cdb_free_addr_get()` function to find available addresses.

Over-capacity
CDB has fixed capacity defined by Kconfig. Exceeding capacity causes allocation failures. Monitor CDB usage and increase capacity if needed.

Accessing invalid nodes
Always check if returned pointers from `_get()` or `_alloc()` functions are `NULL` before dereferencing.

API reference For complete API documentation, refer to:

- Header file: `include/zephyr/bluetooth/mesh/cdb.h`
- Implementation: `subsys/bluetooth/mesh/cdb.c`

Related documentation

- *Bluetooth Mesh Profile*
- *Provisioning*
- Mesh Shell commands: *Bluetooth Mesh Shell*

Core Host and drivers

Logical Link Control and Adaptation Protocol (L2CAP) L2CAP layer enables connection-oriented channels which can be enabled with the configuration option: `CONFIG_BT_L2CAP_DYNAMIC_CHANNEL`. These channels support segmentation and reassembly transparently, they also support credit based flow control making it suitable for data streams.

Channels instances are represented by the `bt_l2cap_chan` struct which contains the callbacks in the `bt_l2cap_chan_ops` struct to inform when the channel has been connected, disconnected or when the encryption has changed. In addition to that it also contains the `recv` callback which is called whenever an incoming data has been received. Data received this way can be marked as processed by returning 0 or using `bt_l2cap_chan_recv_complete()` API if processing is asynchronous.

Note

The recv callback is called directly from RX Thread thus it is not recommended to block for long periods of time.

For sending data the bt_l2cap_chan_send() API can be used noting that it may block if no credits are available, and resuming as soon as more credits are available.

Servers can be registered using bt_l2cap_server_register() API passing the bt_l2cap_server struct which informs what psm it should listen to, the required security level sec_level, and the callback accept which is called to authorize incoming connection requests and allocate channel instances. The allocated objects must be of type bt_l2cap_le_chan, with the channel reference returned through the accept callback pointing to the object's chan member, as shown in the example below.

```
#define PSM 0x29 /* example PSM */

static int l2cap_recv(struct bt_l2cap_chan *chan, struct net_buf *buf);
static void l2cap_connected(struct bt_l2cap_chan *chan);
static void l2cap_disconnected(struct bt_l2cap_chan *chan);

/* ops used for every channel instance */
static struct bt_l2cap_chan_ops l2cap_ops = {
        .recv       = l2cap_recv,
        .connected  = l2cap_connected,
        .disconnected = l2cap_disconnected,
};

/* Fixed array of channel instances, one per possible connection */
static struct bt_l2cap_le_chan l2cap_chans[CONFIG_BT_MAX_CONN];

static int accept_cb(struct bt_conn *conn, struct bt_l2cap_server *server,
                     struct bt_l2cap_chan **chan)
{
        uint8_t conn_index = bt_conn_index(conn);

        /* initialize the chosen entry */
        (void)memset(&l2cap_chans[conn_index], 0, sizeof(l2cap_chans[conn_index]));
        l2cap_chans[conn_index].chan.ops = &l2cap_ops;

        *chan = &l2cap_chans[conn_index].chan;

        printk("L2CAP channel accepted, assigned chan[%d]\n", conn_index);

        return 0; /* accept */
}

static struct bt_l2cap_server server = {
        .psm = PSM,
        .sec_level = BT_SECURITY_L1,
        .accept = accept_cb,
};
```

A complete working example demonstrating L2CAP dynamic channels is available in the acceptor (server) sample: bluetooth_l2cap_coc_acceptor

Fixed Channels The user can also define fixed channels using the `BT_L2CAP_FIXED_CHANNEL_DEFINE` macro. Fixed channels are initialized upon connection, and do not support segmentation. Note that even though the `accept` callback passes the channel reference as a `bt_l2cap_chan`, the allocated object must be of type `bt_l2cap_le_chan`, with the reference pointing to its `chan` member. An example of how to define a fixed channel is shown below.

```
static struct bt_l2cap_le_chan fixed_chan[CONFIG_BT_MAX_CONN];

/* Callbacks are assumed to be defined prior. */
static struct bt_l2cap_chan_ops ops = {
    .recv = recv_cb,
    .sent = sent_cb,
    .connected = connected_cb,
    .disconnected = disconnected_cb,
};

static int l2cap_fixed_accept(struct bt_conn *conn, struct bt_l2cap_chan **chan)
{
    uint8_t conn_index = bt_conn_index(conn);

    fixed_chan[conn_index] = (struct bt_l2cap_le_chan){
        .chan.ops = &ops,
    };

    *chan = &fixed_chan[conn_index].chan;

    return 0;
}

BT_L2CAP_FIXED_CHANNEL_DEFINE(fixed_channel) = {
    .cid = 0x0010,
    .accept = l2cap_fixed_accept,
};
```

Client Channels Client channels can be initiated with use of `bt_l2cap_chan_connect()` API and can be disconnected with the `bt_l2cap_chan_disconnect()` API. Note that the latter can also disconnect channel instances created by servers.

See the initiator (client) sample for a complete example: bluetooth_l2cap_coc_initiator

API Reference

L2CAP

Related code samples

- L2CAP Connection Oriented Channels (Acceptor)Accept incoming L2CAP Connection Oriented Channel connections.
- L2CAP Connection Oriented Channels (Initiator)Initiate L2CAP Connection Oriented Channel connections and send data.

Connection Management The Zephyr Bluetooth stack uses an abstraction called `bt_conn` to represent connections to other devices. The internals of this struct are not exposed to the application, but a limited amount of information (such as the remote address) can be acquired using the `bt_conn_get_info()` API. Connection objects are reference counted, and the application is

expected to use the `bt_conn_ref()` API whenever storing a connection pointer for a longer period of time, since this ensures that the object remains valid (even if the connection would get disconnected). Similarly the `bt_conn_unref()` API is to be used when releasing a reference to a connection.

A common mistake is to forget unreleasing a reference to a connection object created by functions `bt_conn_le_create()` and `bt_conn_le_create_synced()`. To protect against this, use the `CONFIG_BT_CONN_CHECK_NULL_BEFORE_CREATE` Kconfig option, which forces these functions return an error if the connection pointer passed to them is not NULL. This helps to spot such issues and avoid sporadic bugs caused by not releasing the connection object.

An application may track connections by registering a `bt_conn_cb` struct using the `bt_conn_cb_register()` or `BT_CONN_CB_DEFINE` APIs. This struct lets the application define callbacks for connection & disconnection events, as well as other events related to a connection such as a change in the security level or the connection parameters. When acting as a central the application will also get hold of the connection object through the return value of the `bt_conn_le_create()` API.

API Reference
Connection management

Related code samples

- Basic Audio Profile (BAP) Broadcast Audio AssistantUse BAP Broadcast Assistant functionality.
- Basic Audio Profile (BAP) Broadcast Audio SinkUse BAP Broadcast Sink functionality.
- Basic Audio Profile (BAP) Unicast Audio ClientUse BAP Unicast Client functionality.
- Common Audio Profile (CAP) HandoverConnect to a CAP acceptor and perform CAP handover procedures
- Common Audio Profile (CAP) InitiatorConnect to CAP Acceptors and setup unicast audio streaming or broadcast audio streams.
- Peripheral Accept ListAdvertise and accept connections only from devices on an accept list.
- Peripheral SC-onlyEnable "Secure Connections Only" mode for a Bluetooth LE peripheral.
- Telephone and Media Audio Profile (TMAP) CentralImplement the TMAP Call Gateway (CG) and Unicast Media Sender (UMS) roles.

Data Buffers

API Reference
Data buffers

HCI Drivers

API Reference
Bluetooth HCI

HCI Packet Helpers Helpers for framing HCI command packets and parsing command responses in `net_buf_simple` buffers and packet bytes, independent of the Bluetooth Host and of

the HCI driver interface, together with a lockstep helper for HCI drivers that exchange HCI commands with the controller over their own transport, for example for vendor-specific controller initialization.

Both helpers are part of every build with `CONFIG_BT` enabled. The lockstep helper's header lives with the HCI driver API, under `include/zephyr/drivers/bluetooth/`.

These are not general application APIs: the intended users are HCI drivers and Bluetooth stack internals. Applications that need to send HCI commands alongside a running Host use the higher-level `bt_hci_cmd_alloc()`, `bt_hci_cmd_send()` and `bt_hci_cmd_send_sync()` APIs instead, which cooperate with the Host's command flow control.

API Reference
Bluetooth HCI packet helpers

Bluetooth HCI lockstep command helper

Bluetooth H:4 UART driver vendor extension interface

HCI RAW channel

Overview HCI RAW channel API is intended to expose HCI interface to the remote entity. The local Bluetooth controller gets owned by the remote entity and host Bluetooth stack is not used. RAW API provides direct access to packets which are sent and received by the Bluetooth HCI driver.

API Reference
HCI RAW channel

Related code samples

- HCI 3-wire (H:5)Expose a Bluetooth controller to another device or CPU over H5:HCI transport.
- HCI IPCExpose a Bluetooth controller to another device or CPU using the IPC subsystem.
- HCI SPIExpose a Bluetooth controller to another device or CPU over SPI.
- HCI UARTExpose a Bluetooth controller to another device or CPU over UART.
- HCI UART asyncExpose a Bluetooth controller to another device or CPU over asynchronous UART.
- HCI USBTurn a Zephyr board into a USB Bluetooth dongle (compatible with all operating systems).
- Legacy HCI USBTurn a Zephyr board into a USB Bluetooth dongle (compatible with all operating systems).

Cryptography

API Reference
Cryptography

Other

Bluetooth Controller

API Reference
Bluetooth Controller

Universal Unique Identifiers (UUIDs)

API Reference
UUIDs

Tools

This page lists and describes tools that can be used to assist during Bluetooth stack or application development in order to help, simplify and speed up the development process.

- *Mobile applications*
- *Using BlueZ with Zephyr*
- *Running on QEMU or native_sim*
  - *Using the Host System Bluetooth Controller*
  - *Using a Zephyr-based Bluetooth Controller*
  - *HCI Tracing*
- *Running on a Virtual Controller and native_sim*
  - *Android Emulator*
- *Using Zephyr-based Controllers with BlueZ*

Mobile applications It is often useful to make use of existing mobile applications to interact with hardware running Zephyr, to test functionality without having to write any additional code or requiring extra hardware.

The recommended mobile applications for interacting with Zephyr are:

- Android:
  - nRF Connect for Android
  - nRF Mesh for Android
  - LightBlue for Android
- iOS:
  - nRF Connect for iOS
  - nRF Mesh for iOS
  - LightBlue for iOS

Using BlueZ with Zephyr The Linux Bluetooth Protocol Stack, BlueZ, comes with a very useful set of tools that can be used to debug and interact with Zephyr's Bluetooth Host and Controller. In order to benefit from these tools you will need to make sure that you are running a recent version of the Linux Kernel and BlueZ:

- Linux Kernel 4.10+
- BlueZ 4.45+

Additionally, some of the BlueZ tools might not be bundled by default by your Linux distribution. If you need to build BlueZ from scratch to update to a recent version or to obtain all of its tools you can follow the steps below:

```
git clone git://git.kernel.org/pub/scm/bluetooth/bluez.git
cd bluez
./bootstrap-configure --disable-android --disable-midi
make
```

You can then find `btattach`, `btmgt` and `btproxy` in the `tools/` folder and `btmon` in the `monitor/` folder.

You'll need to enable BlueZ's experimental features so you can access its most recent Bluetooth functionality. Do this by editing the file `/lib/systemd/system/bluetooth.service` and making sure to include the `-E` option in the daemon's execution start line:

```
ExecStart=/usr/libexec/bluetooth/bluetoothd -E
```

Finally, reload and restart the daemon:

```
sudo systemctl daemon-reload
sudo systemctl restart bluetooth
```

Running on QEMU or native_sim It's possible to run Bluetooth applications using either the *QEMU emulator* or native_sim.

In either case, a Bluetooth controller needs to be exported from the host OS (Linux) to the emulator. For this purpose you will need some tools described in the *Using BlueZ with Zephyr* section.

Using the Host System Bluetooth Controller The host OS's Bluetooth controller is connected in the following manner:

- To the second QEMU serial line using a UNIX socket. This socket gets used with the help of the QEMU option `-serial unix:/tmp/bt-server-bredr`. This option gets passed to QEMU through `QEMU_EXTRA_FLAGS` automatically whenever an application has enabled Bluetooth support.
- To native_sim's BT User Channel driver through the use of a command-line option passed to the native_sim executable: `--bt-dev=hci0`

On the host side, BlueZ allows you to export its Bluetooth controller through a so-called user channel for QEMU and native_sim to use.

> **Note**
>
> You only need to run `btproxy` when using QEMU. native_sim handles the UNIX socket proxying automatically

If you are using QEMU, in order to make the Controller available you will need one additional step using `btproxy`:

1. Make sure that the Bluetooth controller is down
2. Use the btproxy tool to open the listening UNIX socket, type:

   ```
   sudo tools/btproxy -u -i 0
   Listening on /tmp/bt-server-bredr
   ```

   You might need to replace `-i 0` with the index of the Controller you wish to proxy.

If you see `Received unknown host packet type 0x00` when running QEMU, then add `-z` to the `btproxy` command line to ignore any null bytes transmitted at startup.

Once the hardware is connected and ready to use, you can then proceed to building and running a sample:

- Choose one of the Bluetooth sample applications located in `samples/bluetooth`.
- To run a Bluetooth application in QEMU, type:

```
west build -b qemu_x86 samples/bluetooth/<sample>
west build -t run
```

  Running QEMU now results in a connection with the second serial line to the `bt-server-bredr` UNIX socket, letting the application access the Bluetooth controller.

- To run a Bluetooth application in native_sim, first build it:

```
west build -b native_sim samples/bluetooth/<sample>
```

  And then run it with:

```
$ sudo ./build/zephyr/zephyr.exe --bt-dev=hci0
```

Using a Zephyr-based Bluetooth Controller Depending on which hardware you have available, you can choose between two transports when building a single-mode, Zephyr-based Bluetooth Controller:

- UART: Use the bluetooth_hci_uart sample and follow the instructions in bluetooth-hci-uart-qemu-posix.
- USB: Use the bluetooth_hci_usb sample and then treat it as a Host System Bluetooth Controller (see previous section)

HCI Tracing When running the Host on a computer connected to an external Controller, it is very useful to be able to see the full log of exchanges between the two, in the format of a *Host Controller Interface* log. In order to see those logs, you can use the built-in `btmon` tool from BlueZ:

```
$ btmon
```

The output looks like this:

```
= New Index: 00:00:00:00:00:00 (Primary,Virtual,Control)
↪0.274200
= Open Index: 00:00:00:00:00:00
↪0.274500
< HCI Command: Reset (0x03|0x0003) plen 0                          #1
↪0.274600
> HCI Event: Command Complete (0x0e) plen 4                        #2
↪0.274700
      Reset (0x03|0x0003) ncmd 1
      Status: Success (0x00)
< HCI Command: Read Local Supported Features (0x04|0x0003) plen 0  #3
↪0.274800
> HCI Event: Command Complete (0x0e) plen 12                       #4
↪0.274900
      Read Local Supported Features (0x04|0x0003) ncmd 1
      Status: Success (0x00)
      Features: 0x00 0x00 0x00 0x00 0x60 0x00 0x00 0x00
         BR/EDR Not Supported
         LE Supported (Controller)
```

Embedded HCI tracing When running both Host and Controller in actual Integrated Circuits, you will only see normal log messages on the console by default, without any way of accessing the HCI traffic between the Host and the Controller. However, there is a special Bluetooth logging mode that converts the console to use a binary protocol that interleaves both normal log messages as well as the HCI traffic.

Set the following Kconfig options to enable this protocol before building your application:

```
CONFIG_BT_DEBUG_MONITOR_UART=y
CONFIG_UART_CONSOLE=n
```

- Setting `CONFIG_BT_DEBUG_MONITOR_UART` activates the formatting
- Clearing `CONFIG_UART_CONSOLE` makes the UART unavailable for the system console. E.g. for `printk` and the `boot banner`

Optionally, on boards whose monitor UART driver supports the interrupt API, setting `CONFIG_BT_DEBUG_MONITOR_UART_INTERRUPT_DRIVEN` queues complete monitor records and transmits them from the UART interrupt handler instead of blocking while each byte is transmitted. Records that do not fit in the buffer are dropped and reported in the next monitor record. The buffer size can be adjusted with `CONFIG_BT_DEBUG_MONITOR_UART_BUFFER_SIZE`.

To decode the binary protocol that will now be sent to the console UART you need to use the btmon tool from *BlueZ*:

```
$ btmon --tty <console TTY> --tty-speed 115200
```

If UART is not available (or you still want non-binary logs), you can set `CONFIG_BT_DEBUG_MONITOR_RTT` instead, which will use Segger RTT. For example, if trying to connect to a nRF52840DK with S/N 683578642:

```
$ btmon --jlink nRF52840_xxAA,683578642
```

Running on a Virtual Controller and native_sim An alternative to a Bluetooth physical controller is the use of a virtual controller. This controller can be connected over an HCI TCP server. This TCP server must support the HCI H4 protocol. In comparison to the physical controller variant, the virtual controller allows to test a Zephyr application running on the native boards without a physical Bluetooth controller.

The main use case for a virtual controller is to do Bluetooth connectivity tests without the need of Bluetooth hardware. This allows to automate Bluetooth integration tests with external applications such as a Bluetooth gateway or a mobile application.

To demonstrate this functionality an example is given to interact with a virtual controller. For this purpose, the experimental python module Bumble from Google is used as it allows to create a TCP Bluetooth virtual controller and connect with the Zephyr Bluetooth host. To install bumble follow the Bumble Getting Started Guide.

> **Note**
>
> If your Zephyr application requires the use of the HCI LE Set extended commands, install the branch `controller-extended-advertising` from Bumble.

Android Emulator You can test the virtual controller by connecting a Bluetooth Zephyr application to the Android Emulator.

To connect your application to the Android Emulator follow the next steps:

1. Build your Zephyr application and disable the HCI ACL flow control (i.e. `CONFIG_BT_HCI_ACL_FLOW_CONTROL=n`) as the virtual controller from android does not support it at the moment.

2. Install Android Emulator version >= 33.1.4.0. The easiest way to do this is by installing the latest Android Studio Preview version.
3. Create a new Android Virtual Device (AVD) with the Android Device Manager. The AVD should use at least SDK API 34.
4. Run the Android Emulator via terminal as follows:

   ```
   emulator avd YOUR_AVD -packet-streamer-endpoint default
   ```

5. Create a Bluetooth bridge between the Zephyr application and the virtual controller from Android Emulator with the Bumble utility `hci-bridge`.

   ```
   bumble-hci-bridge tcp-server:_:1234 android-netsim
   ```

   This command will create a TCP server bridge on the local host IP address `127.0.0.1` and port number `1234`.

6. Run the Zephyr application and connect to the TCP server created in the last step.

   ```
   ./zephyr.exe --bt-dev=127.0.0.1:1234
   ```

After following these steps the Zephyr application will be available to the Android Emulator over the virtual Bluetooth controller that was bridged with Bumble. You can verify that the Zephyr application can communicate over Bluetooth by opening the Bluetooth settings in your AVD and scanning for your Zephyr application device. To test this you can build the Bluetooth peripheral samples such as ble_peripheral_hr or ble_peripheral_dis.

Using Zephyr-based Controllers with BlueZ If you want to test a Zephyr-powered Bluetooth Controller using BlueZ's Bluetooth Host, you will need a few tools described in the *Using BlueZ with Zephyr* section. Once you have installed the tools you can then use them to interact with your Zephyr-based controller:

```
sudo tools/btmgmt --index 0
[hci0]# auto-power
[hci0]# find -l
```

You might need to replace `--index 0` with the index of the Controller you wish to manage. Additional information about `btmgmt` can be found in its manual pages.

Shell

The Bluetooth Shell is an application based on the *Shell* module. It offer a collection of commands made to easily interact with the Bluetooth stack.

For specific Bluetooth functionality see also the following shell documentation

Bluetooth: Basic Audio Profile Shell This document describes how to run Basic Audio Profile functionality which includes:

- Capabilities and Endpoint discovery
- Audio Stream Endpoint procedures

Commands

```
bap --help
Subcommands:
   init                    : [ase_sink_count, ase_source_count]
   select_broadcast        : <stream>
   create_broadcast        : [preset <preset_name>] [enc <broadcast_code>]
   start_broadcast         :
   stop_broadcast          :
```

(continues on next page)

(continued from previous page)

```
  delete_broadcast       :
  create_broadcast_sink  : 0x<broadcast_id>
  create_sink_by_name    : <broadcast_name>
  sync_broadcast         : 0x<bis_index> [[[0x<bis_index>] 0x<bis_index>] ...]
                          [bcode <broadcast code> || bcode_str <broadcast code
                          as string>]
  stop_broadcast_sink    : Stops broadcast sink
  term_broadcast_sink    :
  discover               : [dir: sink, source]
  config                 : <direction: sink, source> <index> [loc <loc_bits>]
                          [preset <preset_name>]
  stream_qos             : interval [framing] [latency] [pd] [sdu] [phy] [rtn]
  qos                    : Send QoS configure for Unicast Group
  enable                 : [context]
  connect                : Connect the CIS of the stream
  stop
  list
  print_ase_info         : Print ASE info for default connection
  metadata               : [context]
  start
  disable
  release
  select_unicast         : <stream>
  preset                 : <sink, source, broadcast> [preset]
                          [config
                                [freq <frequency>]
                                [dur <duration>]
                                [chan_alloc <location>]
                                [frame_len <frame length>]
                                [frame_blks <frame blocks>]]
                          [meta
                                [pref_ctx <context>]
                                [stream_ctx <context>]
                                [program_info <program info>]
                                [lang <ISO 639-3 lang>]
                                [ccid_list <ccids>]
                                [parental_rating <rating>]
                                [program_info_uri <URI>]
                                [audio_active_state <state>]
                                [bcast_flag]
                                [extended <meta>]
                                [vendor <meta>]]
  send                   : Send to Audio Stream [data]
  stats                  : Sets or gets the statistics reporting interval in # of
                          packets (set 0 to disable)
  set_location           : <direction: sink, source> <location bitmask>
  set_context            : <direction: sink, source><context bitmask> <type:
                          supported, available>
```

Table 30: State Machine Transitions

Command	Depends	Allowed States	Next States
init	none	any	none
discover	init	any	any
config	discover	idle/codec-configured/qos-configured	codec-configured
qos	config	codec-configured/qos-configured	qos-configured
enable	qos	qos-configured	enabling
connect	qos/enable	qos-configured/enabling	qos-configured/enabling
[start]	enable/connect	enabling	streaming
disable	enable	enabling/streaming	disabling
[stop]	disable	disabling	qos-configure/idle
release	config	any	releasing/codec-configure/idle
list	none	any	none
select_unicast	none	any	none
send	enable	streaming	none

Example Central Connect and establish a sink stream:

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect <address>
uart:~$ gatt exchange-mtu
uart:~$ bap discover sink
uart:~$ bap config sink 0
uart:~$ bap qos
uart:~$ bap enable
uart:~$ bap connect
```

Connect and establish a source stream:

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect <address>
uart:~$ gatt exchange-mtu
uart:~$ bap discover source
uart:~$ bap config source 0
uart:~$ bap qos
uart:~$ bap enable
uart:~$ bap connect
uart:~$ bap start
```

Disconnect and release:

```
uart:~$ bap disable
uart:~$ bap release
```

Example Peripheral Listen:

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt advertise on
```

Server initiated disable and release:

```
uart:~$ bap disable
uart:~$ bap release
```

Example Broadcast Source Create and establish a broadcast source stream:

```
uart:~$ bap init
uart:~$ bap create_broadcast
uart:~$ bap start_broadcast
```

Stop and release a broadcast source stream:

```
uart:~$ bap stop_broadcast
uart:~$ bap delete_broadcast
```

Example Broadcast Sink Scan for and establish a broadcast sink stream. The command `bap create_broadcast_sink` will either use existing periodic advertising sync (if exist) or start scanning and sync to the periodic advertising with the provided broadcast ID before syncing to the BIG.

```
uart:~$ bap init
uart:~$ bap create_broadcast_sink 0xEF6716
No PA sync available, starting scanning for broadcast_id
Found broadcaster with ID 0xEF6716 and addr R:03:47:95:75:C0:08 and sid 0x00
Attempting to PA sync to the broadcaster
PA synced to broadcast with broadcast ID 0xEF6716
Attempting to sync to the BIG
Received BASE from sink 0x20019080:
Presentation delay: 40000
Subgroup count: 1
Subgroup 0x20024182:
   Codec Format: 0x06
   Company ID  : 0x0000
   Vendor ID   : 0x0000
   codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
      Codec specific configuration:
      Sampling frequency: 16000 Hz (3)
      Frame duration: 10000 us (1)
      Channel allocation:
               Front left (0x00000001)
               Front right (0x00000002)
      Octets per codec frame: 40
      Codec specific metadata:
      Streaming audio contexts:
         Unspecified (0x0001)
      BIS index: 0x01
         codec cfg id 0x06 cid 0x0000 vid 0x0000 count 6
         Codec specific configuration:
            Channel allocation:
               Front left (0x00000001)
         Codec specific metadata:
            None
      BIS index: 0x02
         codec cfg id 0x06 cid 0x0000 vid 0x0000 count 6
         Codec specific configuration:
            Channel allocation:
```

(continues on next page)

(continued from previous page)

```
                Front right (0x00000002)
          Codec specific metadata:
             None
Possible indexes: 0x01 0x02
Sink 0x20019110 is ready to sync without encryption
uart:~$ bap sync_broadcast 0x01
```

Scan for and establish a broadcast sink stream by broadcast name The command `bap create_sink_by_name` will start scanning and sync to the periodic advertising with the provided broadcast name before syncing to the BIG.

```
uart:~$ bap init
uart:~$ bap create_sink_by_name "Test Broadcast"
Starting scanning for broadcast_name
Found matched broadcast name 'Test Broadcast' with address R:03:47:95:75:C0:08
Found broadcaster with ID 0xEF6716 and addr R:03:47:95:75:C0:08 and sid 0x00
Attempting to PA sync to the broadcaster
PA synced to broadcast with broadcast ID 0xEF6716
Attempting to create the sink
Received BASE from sink 0x20019080:
Presentation delay: 40000
Subgroup count: 1
Subgroup 0x20024182:
   Codec Format: 0x06
   Company ID  : 0x0000
   Vendor ID   : 0x0000
   codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
      Codec specific configuration:
      Sampling frequency: 16000 Hz (3)
      Frame duration: 10000 us (1)
      Channel allocation:
               Front left (0x00000001)
               Front right (0x00000002)
      Octets per codec frame: 40
      Codec specific metadata:
      Streaming audio contexts:
         Unspecified (0x0001)
      BIS index: 0x01
         codec cfg id 0x06 cid 0x0000 vid 0x0000 count 6
         Codec specific configuration:
            Channel allocation:
               Front left (0x00000001)
         Codec specific metadata:
            None
      BIS index: 0x02
         codec cfg id 0x06 cid 0x0000 vid 0x0000 count 6
         Codec specific configuration:
            Channel allocation:
               Front right (0x00000002)
         Codec specific metadata:
            None
Possible indexes: 0x01 0x02
Sink 0x20019110 is ready to sync without encryption
uart:~$ bap sync_broadcast 0x01
```

Syncing to encrypted broadcast If the broadcast is encrypted, the broadcast code can be entered with the `bap sync_broadcast` command as such:

```
Sink 0x20019110 is ready to sync with encryption
uart:~$ bap sync_broadcast 0x01 bcode 0102030405060708090a0b0c0d0e0f
```

The broadcast code can be 1-16 values, either as a string or a hexadecimal value.

```
Sink 0x20019110 is ready to sync with encryption
uart:~$ bap sync_broadcast 0x01 bcode_str thisismycode
```

Stop and release a broadcast sink stream:

```
uart:~$ bap stop_broadcast_sink
uart:~$ bap term_broadcast_sink
```

Init The `init` command register local PAC records which are necessary to be able to configure stream and properly manage capabilities in use.

Table 31: State Machine Transitions

Depends	Allowed States	Next States
none	any	none

```
uart:~$ bap init
```

Discover PAC(s) and ASE(s) Once connected the `discover` command discover PAC records and ASE characteristics representing remote endpoints.

Table 32: State Machine Transitions

Depends	Allowed States	Next States
init	any	any

> **Note**
>
> Use command `gatt exchange-mtu` to make sure the MTU is configured properly.

```
uart:~$ gatt exchange-mtu
Exchange pending
Exchange successful
uart:~$ bap discover [type: sink, source]
uart:~$ bap discover sink
conn 0x8256f80: dir Sink (0x01)
codec cap id 0x06 cid 0x0000 vid 0x0000
  Codec specific capabilities:
    Supported sampling frequencies:
      8000 Hz (0x0001)
      11025 Hz (0x0002)
      16000 Hz (0x0004)
      22050 Hz (0x0008)
      24000 Hz (0x0010)
      32000 Hz (0x0020)
```

(continues on next page)

(continued from previous page)

```
      44100 Hz (0x0040)
      48000 Hz (0x0080)
      88200 Hz (0x0100)
      96000 Hz (0x0200)
      176400 Hz (0x0400)
      192000 Hz (0x0800)
      384000 Hz (0x1000)
    Supported frame durations:
      7.5 ms (0x01)
      10 ms (0x02)
    Supported channel counts:
      1 channel (0x01)
      2 channels (0x02)
    Supported octets per codec frame counts:
      Min: 30
      Max: 155
    Supported max codec frames per SDU: 4
  Codec capabilities metadata:
    Preferred audio contexts:
      Unspecified        (0x0001)
      Conversational     (0x0002)
      Media              (0x0004)
conn 0x8256f80: dir Sink (0x01)
codec cap id 0x06 cid 0x0000 vid 0x0000
  Codec specific capabilities:
    Supported sampling frequencies:
      16000 Hz (0x0004)
      32000 Hz (0x0020)
      48000 Hz (0x0080)
    Supported frame durations:
      7.5 ms (0x01)
      10 ms (0x02)
    Supported channel counts:
      1 channel (0x01)
      2 channels (0x02)
    Supported octets per codec frame counts:
      Min: 30
      Max: 120
    Supported max codec frames per SDU: 4
  Codec capabilities metadata:
    Preferred audio contexts:
      Game               (0x0008)
Supported audio contexts:
  Sink:
    Unspecified        (0x0001)
    Conversational     (0x0002)
    Media              (0x0004)
    Game               (0x0008)
    Instructional      (0x0010)
    Voice assistant    (0x0020)
    Live               (0x0040)
    Sound effects      (0x0080)
  Source:
    Unspecified        (0x0001)
    Conversational     (0x0002)
    Media              (0x0004)
```

(continues on next page)

(continued from previous page)

```
    Game                (0x0008)
    Instructional       (0x0010)
    Voice assistant     (0x0020)
    Live                (0x0040)
    Sound effects       (0x0080)
Sink location:
  Front left                (0x00000001)
  Front right               (0x00000002)
Available audio contexts:
  Sink:
    Unspecified         (0x0001)
    Conversational      (0x0002)
    Media               (0x0004)
    Game                (0x0008)
    Instructional       (0x0010)
    Voice assistant     (0x0020)
    Live                (0x0040)
    Sound effects       (0x0080)
  Source:
    Unspecified         (0x0001)
    Conversational      (0x0002)
    Media               (0x0004)
    Game                (0x0008)
    Instructional       (0x0010)
    Voice assistant     (0x0020)
    Live                (0x0040)
    Sound effects       (0x0080)
Conn: 0x8256f80, Sink #0: ep 0x8288e1c
Conn: 0x8256f80, Sink #1: ep 0x8289000
Discover complete: err 0
```

Select preset The `preset` command can be used to either print the default preset configuration or set a different one, it is worth noting that it doesn't change any stream previously configured.

```
uart:~$ bap preset
preset - <sink, source, broadcast> [preset]
         [config
               [freq <frequency>]
               [dur <duration>]
               [chan_alloc <location>]
               [frame_len <frame length>]
               [frame_blks <frame blocks>]]
         [meta
               [pref_ctx <context>]
               [stream_ctx <context>]
               [program_info <program info>]
               [lang <ISO 639-3 lang>]
               [ccid_list <ccids>]
               [parental_rating <rating>]
               [program_info_uri <URI>]
               [audio_active_state <state>]
               [bcast_flag]
               [extended <meta>]
               [vendor <meta>]]
uart:~$ bap preset sink
```

(continues on next page)

(continued from previous page)

```
16_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
   Codec specific configuration:
      Sampling frequency: 16000 Hz (3)
      Frame duration: 10000 us (1)
      Channel allocation:
                  Front left (0x00000001)
                  Front right (0x00000002)
      Octets per codec frame: 40
   Codec specific metadata:
      Streaming audio contexts:
         Game (0x0008)
QoS: interval 10000 framing 0x00 phy 0x02 sdu 40 rtn 2 latency 10 pd 40000

uart:~$ bap preset sink 32_2_1
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
   Codec specific configuration:
      Sampling frequency: 32000 Hz (6)
      Frame duration: 10000 us (1)
      Channel allocation:
                  Front left (0x00000001)
                  Front right (0x00000002)
      Octets per codec frame: 80
   Codec specific metadata:
      Streaming audio contexts:
         Game (0x0008)
   QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000
```

Configure preset The `bap preset` command can also be used to configure the preset used for the subsequent commands. It is possible to add or set (or reset) any value. To reset the preset, the command can simply be run without the `config` or `meta` parameter. The parameters are using the assigned numbers values.

```
uart:~$ bap preset sink 32_2_1
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
data #0: type 0x01 value_len 1
00000000: 06                                              |.                 |
data #1: type 0x02 value_len 1
00000000: 01                                              |.                 |
data #2: type 0x03 value_len 4
00000000: 03 00 00 00                                     |....              |
data #3: type 0x04 value_len 2
00000000: 50 00                                           |P.                |
meta #0: type 0x02 value_len 2
00000000: 08 00                                           |..                |
QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000

uart:~$ bap preset sink 32_2_1 config freq 10
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
data #0: type 0x01 value_len 1
00000000: 0a                                              |.                 |
data #1: type 0x02 value_len 1
```

(continues on next page)

(continued from previous page)

```
00000000: 01                                              |.                 |
data #2: type 0x03 value_len 4
00000000: 03 00 00 00                                     |....              |
data #3: type 0x04 value_len 2
00000000: 50 00                                           |P.                |
meta #0: type 0x02 value_len 2
00000000: 08 00                                           |..                |
QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000

uart:~$ bap preset sink 32_2_1 config freq 10 meta lang "eng" stream_ctx 4
32_2_1
codec cfg id 0x06 cid 0x0000 vid 0x0000 count 16
data #0: type 0x01 value_len 1
00000000: 0a                                              |.                 |
data #1: type 0x02 value_len 1
00000000: 01                                              |.                 |
data #2: type 0x03 value_len 4
00000000: 03 00 00 00                                     |....              |
data #3: type 0x04 value_len 2
00000000: 50 00                                           |P.                |
meta #0: type 0x02 value_len 2
00000000: 04 00                                           |..                |
meta #1: type 0x04 value_len 3
00000000: 65 6e 67                                        |eng               |
QoS: interval 10000 framing 0x00 phy 0x02 sdu 80 rtn 2 latency 10 pd 40000
```

Configure Codec The `config` command attempts to configure a stream for the given direction using a preset codec configuration which can either be passed directly or in case it is omitted the default preset is used.

Table 33: State Machine Transitions

Depends	Allowed States	Next States
discover	idle/codec-configured/qos-configured	codec-configured

```
uart:~$ bap config <direction: sink, source> <index> [loc <loc_bits>] [preset
↪<preset_name>]
uart:~$ bap config sink 0
Setting location to 0x00000000
ASE config: preset 16_2_1
stream 0x2000df70 config operation rsp_code 0 reason 0
```

Configure Stream QoS The `stream_qos` Sets a new stream QoS.

```
uart:~$ bap stream_qos <interval> [framing] [latency] [pd] [sdu] [phy] [rtn]
uart:~$ bap stream_qos 10
```

Configure QoS The `qos` command attempts to configure the stream QoS using the preset configuration, each individual QoS parameter can be set with use optional parameters.

Table 34: State Machine Transitions

Depends	Allowed States	Next States
config	qos-configured/codec-configured	qos-configured

```
uart:~$ bap qos
```

Enable The `enable` command attempts to enable the stream previously configured.

Table 35: State Machine Transitions

Depends	Allowed States	Next States
qos	qos-configured	enabling

```
uart:~$ bap enable [context]
uart:~$ bap enable Media
```

Connect The `connect` command attempts to connect the stream previously configured. Sink streams will have to be started by the unicast server, and source streams will have to be started by the unicast client.

Table 36: State Machine Transitions

Depends	Allowed States	Next States
qos/enable	qos-configured/enabling	qos-configured/enabling

```
uart:~$ bap connect
```

Start The `start` command is only necessary when starting a source stream.

Table 37: State Machine Transitions

Depends	Allowed States	Next States
enable/connect	enabling	streaming

```
uart:~$ bap start
```

Disable The `disable` command attempts to disable the stream previously enabled, if the remote peer accepts then the ISO disconnection procedure is also initiated.

Table 38: State Machine Transitions

Depends	Allowed States	Next States
enable	enabling/streaming	disabling

```
uart:~$ bap disable
```

Stop The `stop` command is only necessary when acting as a sink as it indicates to the source the stack is ready to stop receiving data.

Table 39: State Machine Transitions

Depends	Allowed States	Next States
disable	disabling	qos-configure/idle

```
uart:~$ bap stop
```

Release The `release` command releases the current stream and its configuration.

Table 40: State Machine Transitions

Depends	Allowed States	Next States
config	any	releasing/codec-configure/idle

```
uart:~$ bap release
```

List The `list` command list the available streams.

Table 41: State Machine Transitions

Depends	Allowed States	Next States
none	any	none

```
uart:~$ bap list
*0: ase 0x01 dir 0x01 state 0x01
```

Select Unicast The `select_unicast` command set a unicast stream as default.

Table 42: State Machine Transitions

Depends	Allowed States	Next States
none	any	none

```
uart:~$ bap select <ase>
uart:~$ bap select 0x01
Default stream: 1
```

To select a broadcast stream:

```
uart:~$ bap select 0x01 broadcast
Default stream: 1 (broadcast)
```

Send The send command sends data over BAP Stream.

Table 43: State Machine Transitions

Depends	Allowed States	Next States
enable	streaming	none

```
uart:~$ bap send [count]
uart:~$ bap send
Audio sending...
```

Bluetooth: Basic Audio Profile: Broadcast Assistant Shell This document describes how to run the BAP Broadcast Assistant functionality. Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

The Broadcast Assistant is responsible for offloading scan for a resource restricted device, such that scanning does not drain the battery. The Broadcast Assistant shall support scanning for periodic advertisements and may optionally support the periodic advertisements synchronization transfer (PAST) protocol.

The Broadcast Assistant will typically be phones or laptops. The Broadcast Assistant scans for periodic advertisements and transfer information to the server.

It is necessary to have `CONFIG_BT_BAP_BROADCAST_ASSISTANT_LOG_LEVEL_DBG` enabled for using the Broadcast Assistant interactively.

When the Bluetooth stack has been initialized (`bt init`), and a device has been connected, the Broadcast Assistant can discover BASS on the connected device calling `bap_broadcast_assistant discover`, which will start a discovery for the BASS UUIDs and store the handles, and subscribe to all notifications.

```
uart:~$ bap_broadcast_assistant --help
bap_broadcast_assistant - Bluetooth BAP broadcast assistant client shell
                        commands
Subcommands:
discover          : Discover BASS on the server
scan_start        : Start scanning for broadcasters
scan_stop         : Stop scanning for BISs
add_src           : Add a source <address: P:XX:XX:XX:XX:XX:XX or
                    R:XX:XX:XX:XX:XX:XX> <adv_sid> <sync_pa> <broadcast_id>
                    [<sync_bis>] [<pa_interval>] [<metadata>]
add_broadcast_id  : Add a source by broadcast ID <broadcast_id> <sync_pa>
                    [<sync_bis>] [<metadata>]
add_broadcast_name: Add a source by broadcast name <broadcast_name> <sync_pa>
                    [<sync_bis>] [<metadata>]
add_pa_sync       : Add a PA sync as a source <sync_pa> <broadcast_id>
                    [bis_index [bis_index [bix_index [...]]]]>
mod_src           : Set sync <src_id> <sync_pa> [<pa_interval> | "unknown"]␣
↪[<sync_bis>]
                    [<metadata>]
broadcast_code    : Send a string-based broadcast code of up to 16 bytes
                    <src_id> <broadcast code>
rem_src           : Remove a source <src_id>
read_state        : Remove a source <index>
```

Example usage

Setup

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect P:xx:xx:xx:xx:xx:xx
```

When connected

Note

> The Broadcast Assistant will not actually start scanning for periodic advertisements, as that feature is still, at the time of writing, not implemented.

Start scanning for periodic advertisements for a server:

```
uart:~$ bap_broadcast_assistant discover
BASS discover done with 1 recv states
uart:~$ bap_broadcast_assistant scan_start true
BASS scan start successful
Found broadcaster with ID 0x05BD38 and addr R:1E:4D:0A:AA:6E:49 and sid 0x00
```

Adding a source to the receive state with add_src:

```
uart:~$ bap_broadcast_assistant add_src P:11:22:33:44:55:66 5 1 1
BASS recv state: src_id 0, addr P:11:22:33:44:55:66, sid 5, sync_state 1,␣
↪encrypt_state 00000000000000000000000000000000
    [0]: BIS sync 0, metadata_len 0
```

Adding a source to the receive state with add_broadcast_id (recommended):

```
uart:~$ bap_broadcast_assistant add_broadcast_id 0x05BD38 true
[DEVICE]: R:1E:4D:0A:AA:6E:49, AD evt type 5, RSSI -28 Broadcast Audio Source C:0␣
↪S:0 D:0 SR:0 E:1 Prim: LE 1M, Secn: LE 2M, Interval: 0x03c0 (1200000 us), SID:␣
↪0x0
Found BAP broadcast source with address R:1E:4D:0A:AA:6E:49 and ID 0x05BD38
BASS recv state: src_id 0, addr R:1E:4D:0A:AA:6E:49, sid 0, sync_state 0,␣
↪encrypt_state 0
     [0]: BIS sync 0x0000, metadata_len 0
BASS add source successful
BASS recv state: src_id 0, addr R:1E:4D:0A:AA:6E:49, sid 0, sync_state 2,␣
↪encrypt_state 0
     [0]: BIS sync 0x0000, metadata_len 0
BASS recv state: src_id 0, addr R:1E:4D:0A:AA:6E:49, sid 0, sync_state 2,␣
↪encrypt_state 0
     [0]: BIS sync 0x0000, metadata_len 4
              Metadata length 2, type 2, data: 0100
```

Modifying a receive state:

```
uart:~$ bap_broadcast_assistant mod_src 0 true 0x03c0 0x02
BASS modify source successful
BASS recv state: src_id 0, addr R:1E:4D:0A:AA:6E:49, sid 0, sync_state 2,␣
↪encrypt_state 0
     [0]: BIS sync 0x0001, metadata_len 4
              Metadata length 2, type 2, data: 0100
```

Supplying a broadcast code:

```
uart:~$ bap_broadcast_assistant broadcast_code 0 secretCode
Sending broadcast code:
00000000: 73 65 63 72 65 74 43 6f 64 65 00 00 00 00 00 00 |secretCo de....|
uart:~$ BASS broadcast code successful
```

Bluetooth: Basic Audio Profile: Scan Delegator Shell This document describes how to run the Scan Delegator functionality, Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

The Scan Delegator may optionally support the periodic advertisements synchronization transfer (PAST) protocol.

The Scan Delegator server typically resides on devices that have inputs or outputs.

It is necessary to have `CONFIG_BT_BAP_SCAN_DELEGATOR_LOG_LEVEL_DBG` enabled for using the Scan Delegator interactively.

The Scan Delegator can currently only set the sync state of a receive state, but does not actually support syncing with periodic advertisements yet.

```
bap_scan_delegator --help
bap_scan_delegator - Bluetooth BAP Scan Delegator shell commands
Subcommands:
  init                : Initialize the service and register callbacks
  set_past_pref       : Set PAST preference <true || false>
  sync_pa             : Sync to PA <src_id>
  term_pa             : Terminate PA sync <src_id>
  add_src             : Add a PA as source <addr> <sid> <broadcast_id>
                        <enc_state> [bis_sync [metadata]]
  add_src_by_pa_sync  : Add a PA as source <broadcast_id> <enc_state> [bis_sync
                        [metadata]]
  mod_src             : Modify source <src_id> <broadcast_id> <enc_state>
                        [bis_sync [metadata]]
  rem_src             : Remove source <src_id>
  synced              : Set server scan state <src_id> <bis_syncs>
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ bap_scan_delegator init
uart:~$ bt advertise on
Advertising started
```

Adding a source

```
uart:~$ bap_scan_delegator add_src P:11:22:33:44:55:66 0 1234 0
Receive state with ID 0 updated
```

Adding a source from a PA sync

```
uart:~$ bt scan on
Found broadcaster with ID 0x681A22 and addr R:2C:44:05:82:EB:82 and sid 0x00␣
↪(looking for 0x1000000)
uart:~$ bt scan off
uart:~$ bt per-adv-sync-create R:2C:44:05:82:EB:82 0
PA 0x2003e9b0 synced
uart:~$ bap_scan_delegator add_src_by_pa_sync 0x681A22 0
Receive state with ID 0 updated
```

When connected Set sync state for a source:

```
uart:~$ bap_scan_delegator synced 0 1 3 0
```

Bluetooth: Common Audio Profile Shell This document describes how to run the Common Audio Profile functionality.

CAP Acceptor The Acceptor will typically be a resource-constrained device, such as a headset, earbud or hearing aid. The Acceptor can initialize a Coordinated Set Identification Service instance, if it is in a pair with one or more other CAP Acceptors.

Using the CAP Acceptor When the Bluetooth stack has been initialized (`bt init`), the Acceptor can be registered by calling `cap_acceptor init`, which will register the CAS and CSIS services, as well as register callbacks.

```
cap_acceptor --help
cap_acceptor - Bluetooth CAP acceptor shell commands
Subcommands:
  init          :Initialize the service and register callbacks [size <int>]
                 [rank <int>] [not-lockable] [sirk <data>]
  lock          :Lock the set
  release       :Release the set [force]
  sirk          :Set the currently used SIRK <sirk>
  get_info      :Get CSIS info
  sirk_rsp      :Set the response used in SIRK requests <accept, accept_enc,
                 reject, oob>
```

Besides initializing the CAS and the CSIS, there are also commands to lock and release the CSIS instance, as well as printing and modifying access to the SIRK of the CSIS.

Setting a new SIRK This command can modify the currently used SIRK. To get the new RSI to advertise on air, `bt adv-data` or `bt advertise` must be called again to set the new advertising data. If `CONFIG_BT_CSIP_SET_MEMBER_SIRK_NOTIFIABLE` is enabled, this will also notify connected clients.

```
uart:~$ cap_acceptor sirk 00112233445566778899aabbccddeeff
SIRK updated
```

Getting the current SIRK This command can get the currently used SIRK.

```
uart:~$ cap_acceptor get_info
Info for 0x2003b0c8
        SIRK
00000000: 20 37 0a 00 95 c4 04 20  00 00 00 00 f1 79 09 00 | 7.....  .....y..|
        Set size: 2
        Rank: 1
        Lockable: true
        Locked: false
```

CAP Initiator The Initiator will typically be a resource-rich device, such as a phone or PC. The Initiator can discover CAP Acceptors's CAS and optional CSIS services. The CSIS service can be read to provide information about other CAP Acceptors in the same Coordinated Set. The Initiator can execute stream control procedures on sets of devices, either ad-hoc or Coordinated, and thus provides an easy way to setup multiple streams on multiple devices at once.

Using the CAP Initiator When the Bluetooth stack has been initialized (`bt init`), the Initiator can discover CAS and the optionally included CSIS instance by calling (`cap_initiator discover`). The CAP initiator also supports broadcast audio as a source.

```
uart:~$ cap_initiator --help
cap_initiator - Bluetooth CAP initiator shell commands
Subcommands:
```

(continues on next page)

(continued from previous page)

```
  discover          : Discover CAS
  unicast_start     : Unicast Start [csip] [sinks <cnt> (default 1)] [sources
                     <cnt> (default 1)] [conns (<cnt> | all) (default 1)]
  unicast_list      : Unicast list streams
  unicast_update    : Unicast Update <all | stream [stream [stream...]]>
  unicast_stop      : Unicast stop streams [stream [stream [stream...]]] (all by␣
↪default)
  unicast_cancel    : Unicast cancel current procedure
  ac_1              : Unicast audio configuration 1
  ac_2              : Unicast audio configuration 2
  ac_3              : Unicast audio configuration 3
  ac_4              : Unicast audio configuration 4
  ac_5              : Unicast audio configuration 5
  ac_6_i            : Unicast audio configuration 6(i)
  ac_6_ii           : Unicast audio configuration 6(ii)
  ac_7_i            : Unicast audio configuration 7(i)
  ac_7_ii           : Unicast audio configuration 7(ii)
  ac_8_i            : Unicast audio configuration 8(i)
  ac_8_ii           : Unicast audio configuration 8(ii)
  ac_9_i            : Unicast audio configuration 9(i)
  ac_9_ii           : Unicast audio configuration 9(ii)
  ac_10             : Unicast audio configuration 10
  ac_11_i           : Unicast audio configuration 11(i)
  ac_11_ii          : Unicast audio configuration 11(ii)
  broadcast_start   :
  broadcast_update  : <meta>
  broadcast_stop    :
  broadcast_delete  :
  ac_12             : Broadcast audio configuration 12
  ac_13             : Broadcast audio configuration 13
  ac_14             : Broadcast audio configuration 14
```

Before being able to perform any stream operation, the device must also perform the `bap discover` operation to discover the ASEs and PAC records. The `bap init` command also needs to be called.

When connected Discovering CAS and CSIS on a device:

```
uart:~$ cap_initiator discover
discovery completed with CSIS
```

Discovering ASEs and PAC records on a device:

```
uart:~$ bap discover
conn 0x81cc260: #0: codec 0x81d5b28 dir 0x01
codec 0x06 cid 0x0000 vid 0x0000 count 5
data #0: type 0x01 len 2
00000000: f5                                              |.                 |
data #1: type 0x02 len 1
data #2: type 0x03 len 1
data #3: type 0x04 len 4
00000000: 1e 00 f0                                        |...               |
data #4: type 0x05 len 1
meta #0: type 0x01 len 2
00000000: 06                                              |.                 |
dir 1 loc 1
```

(continues on next page)

(continued from previous page)

```
snk ctx 6 src ctx 6
Conn: 0x81cc260, Sink #0: ep 0x81e4248
Conn: 0x81cc260, Sink #1: ep 0x81e46a8
conn 0x81cc260: #0: codec 0x81d5f00 dir 0x02
codec 0x06 cid 0x0000 vid 0x0000 count 5
data #0: type 0x01 len 2
00000000: f5                                              |.                |
data #1: type 0x02 len 1
data #2: type 0x03 len 1
data #3: type 0x04 len 4
00000000: 1e 00 f0                                        |...              |
data #4: type 0x05 len 1
meta #0: type 0x01 len 2
00000000: 06                                              |.                |
dir 2 loc 1
snk ctx 6 src ctx 6
Conn: 0x81cc260, Source #0: ep 0x81e5c88
Conn: 0x81cc260, Source #1: ep 0x81e60e8
Discover complete: err 0
```

Both of the above commands should be done for each device that you want to use in the set. To use multiple devices, simply connect to more and then use `bt select` the device to execute the commands on.

Once all devices have been connected and the respective discovery commands have been called, the `cap_initiator unicast_start` command can be used to put one or more streams into the streaming state.

```
uart:~$ cap_initiator unicast_start sinks 1 sources 0 conns all
Setting up 1 sinks and 0 sources on each (2) conn
Starting 1 streams
Unicast start completed
```

To stop all the streams that has been started, the `cap_initiator unicast_stop` command can be used.

```
uart:~$ cap_initiator unicast_stop all
Unicast stop completed
```

When doing broadcast To start a broadcast as the CAP initiator there are a few steps to be done:

1. Create and configure an extended advertising set with periodic advertising
2. Create and configure a broadcast source
3. Setup extended and periodic advertising data

The following commands will setup a CAP broadcast source using the 16_2_1 preset (defined by BAP):

```
bt init
bap init
bt adv-create nconn-nscan ext-adv
bt per-adv-param
bap preset broadcast 16_2_1
cap_initiator ac_12
bt adv-data dev-name discov
```

(continues on next page)

(continued from previous page)

```
bt per-adv-data
cap_initiator broadcast_start
bt adv-start
bt per-adv on
```

The broadcast source is created by the `cap_initiator ac_12`, `cap_initiator ac_13`, and `cap_initiator ac_14` commands, configuring the broadcast source for the defined audio configurations from BAP. The broadcast source can then be stopped with `cap_initiator broadcast_stop` or deleted with `cap_initiator broadcast_delete`.

The metadata of the broadcast source can be updated at any time, including when it is already streaming. To update the metadata the `cap_initiator broadcast_update` command can be used. The command takes an array of data, and the only requirement (besides having valid data) is that the streaming context shall be set. For example to set the streaming context to media, the command can be used as

```
cap_initiator broadcast_update 03020400
CAP Broadcast source updated with new metadata. Update the advertised base via `bt␣
↪per-adv-data`
bt per-adv-data
```

The `bt per-adv-data` command should be used afterwards to update the data is the advertised BASE. The data must be little-endian, so in the above example the metadata `03020400` is setting the metadata entry with `03` as the length, `02` as the type (streaming context) and `0400` as the value `BT_AUDIO_CONTEXT_TYPE_MEDIA` (which has the numeric value of 0x).

CAP Commander The Commander will typically be a either co-located with a CAP Initiator or be on a separate resource-rich mobile device, such as a phone or smartwatch. The Commander can discover CAP Acceptors's CAS and optional CSIS services. The CSIS service can be read to provide information about other CAP Acceptors in the same Coordinated Set. The Commander can provide information about broadcast sources to CAP Acceptors or coordinate capture and rendering information such as mute or volume states.

Using the CAP Commander When the Bluetooth stack has been initialized (`bt init`), the Commander can discover CAS and the optionally included CSIS instance by calling (`cap_commander discover`).

```
cap_commander --help
cap_commander - Bluetooth CAP commander shell commands
Subcommands:
  discover                  :Discover CAS
  cancel                    :CAP commander cancel current procedure
  change_volume             :Change volume on all connections <volume>
  change_volume_mute        :Change volume mute state on all connections <mute>
  change_volume_offset      :Change volume offset per connection <volume_offset
                             [volume_offset [...]]>
  change_microphone_mute    :Change microphone mute state on all connections <mute>
  change_microphone_gain    :Change microphone gain per connection <gain
                             [gain [...]]>
  broadcast_reception_start : Start broadcast reception with source
                              <address: P:XX:XX:XX:XX:XX:XX or R:XX:XX:XX:XX:XX:XX>
                              <adv_sid> <broadcast_id>
                              [<pa_interval>] [<sync_bis>] [<metadata>]
  broadcast_reception_stop  : Stop broadcast reception <src_id [...]>
  distribute_broadcast_code : Distribute broadcast code <src_id [...]>
↪<broadcast_code>
```

Before being able to perform any stream operation, the device must also perform the `bap discover` operation to discover the ASEs and PAC records. The `bap init` command also needs to be called.

When connected

Discovering CAS and CSIS on a device

```
uart:~$ cap_commander discover
discovery completed with CSIS
```

Setting the volume on all connected devices

```
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$ cap_commander change_volume 15
uart:~$ cap_commander change_volume 15
Setting volume to 15 on 2 connections
VCP volume 15, mute 0
VCP vol_set done
VCP volume 15, mute 0
VCP flags 0x01
VCP vol_set done
Volume change completed
```

Setting the volume offset on one or more devices The offsets are set by connection index, so connection index 0 gets the first offset, and index 1 gets the second offset, etc.:

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ cap_commander change_volume_offset 10
Setting volume offset on 1 connections
VOCS inst 0x200140a4 offset 10
Offset set for inst 0x200140a4
Volume offset change completed
uart:~$
uart:~$ cap_commander change_volume_offset 10 15
Setting volume offset on 2 connections
Offset set for inst 0x200140a4
VOCS inst 0x20014188 offset 15
Offset set for inst 0x20014188
Volume offset change completed
```

Setting the volume mute on all connected devices

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ vcp_vol_ctlr discover
VCP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ cap_commander change_volume_mute 1
Setting volume mute to 1 on 2 connections
VCP volume 100, mute 1
VCP mute done
VCP volume 100, mute 1
VCP mute done
Volume mute change completed
uart:~$ cap_commander change_volume_mute 0
Setting volume mute to 0 on 2 connections
VCP volume 100, mute 0
VCP unmute done
VCP volume 100, mute 0
VCP unmute done
Volume mute change completed
```

Setting the microphone mute on all connected devices

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 VOCS and 1 AICS
uart:~$
uart:~$ cap_commander change_microphone_mute 1
Setting microphone mute to 1 on 2 connections
MICP microphone 100, mute 1
MICP mute done
MICP microphone 100, mute 1
MICP mute done
Microphone mute change completed
uart:~$ cap_commander change_microphone_mute 0
Setting microphone mute to 0 on 2 connections
MICP microphone 100, mute 0
MICP unmute done
MICP microphone 100, mute 0
```

(continues on next page)

(continued from previous page)

```
MICP unmute done
Microphone mute change completed
```

Setting the microphone gain on one or more devices The gains are set by connection index, so connection index 0 gets the first offset, and index 1 gets the second offset, etc.:

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 AICS
uart:~$
uart:~$ bt connect <device B>
Connected: <device B>
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ micp_mic_ctlr discover
MICP discover done with 1 AICS
uart:~$
uart:~$ cap_commander change_microphone_gain 10
Setting microphone gain on 1 connections
AICS inst 0x200140a4 state gain 10, mute 0, mode 0
Gain set for inst 0x200140a4
Microphone gain change completed
uart:~$
uart:~$ cap_commander change_microphone_gain 10 15
Setting microphone gain on 2 connections
Gain set for inst 0x200140a4
AICS inst 0x20014188 state gain 15, mute 0, mode 0
Gain set for inst 0x20014188
Microphone gain change completed
```

Starting and stopping broadcast reception

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ bap_init
uart:~$ cap_commander discover
discovery completed with CSIS
uart:~$ bap_broadcast_assistant discover
BASS discover done with 1 recv states
uart:~$ cap_commander broadcast_reception_start <device B> 0 4
Starting broadcast reception on 1 connection(s)
Broadcast reception start completed
uart:~$ cap_commander broadcast_reception_stop 0
Stopping broadcast reception on 1 connection(s)
Broadcast reception stop completed
```

Distributing the broadcast code

```
uart:~$ bt connect <device A>
Connected: <device A>
uart:~$ bap_init
uart:~$ cap_commander discover
```

(continues on next page)

(continued from previous page)

```
discovery completed with CSIS
uart:~$ bap_broadcast_assistant discover
BASS discover done with 1 recv states
uart:~$ cap_commander broadcast_reception_start <device B> 0 4
Starting broadcast reception on 1 connection(s)
Broadcast reception start completed
uart:~$ cap_commander distribute_broadcast_code 0 "BroadcastCode"
Distribute broadcast code completed
```

CAP Handover The handover procedures allow the user to switch between unicast and broadcast streams. Since broadcast streams are always unidirectional, the procedures will only work for streams with audio direction from the Initiator to the Acceptor (sink streams).

Using the CAP Handover procedures When the Bluetooth stack has been initialized (`bt init`), one or more remote CAP acceptor devices have been connected, and audio streams have been set up, the handover procedures can be used to switch between unicast and broadcast. Before any of the handover procedures can be used, the `bap discover`, `cap_initiator discover` and `bap_broadcast_assistant discover` commands must have been issued and completed.

```
cap_handover --help
cap_handover - Bluetooth CAP handover shell commands
Subcommands:
  unicast_to_broadcast  : Handover current unicast group to broadcast (unicast
                        group will be deleted) [enc <broadcast_code>] [preset
↪<preset_name>]
  broadcast_to_unicast  : Handover current broadcast source to unicast
                        (broadcast source will be deleted)
                        [conns <count>|all] [preset <preset_name>]
```

Handover unicast to broadcast This command hands over one or more unicast streams from unicast to broadcast.

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect <addr>

# Discover necessary services
uart:~$ bap discover
uart:~$ cap_initiator discover
uart:~$ bap_broadcast_assistant discover

# Setup unicast audio e.g. using the ac_1
uart:~$ cap_initiator ac_1

# Create a non-connectable and non-scannable extended advertising set for broadcast
uart:~$ bt adv-create nconn-nscan ext-adv
uart:~$ bt per-adv-param

# Perform the handover and update the advertising data to contain the broadcast ID
uart:~$ cap_handover unicast_to_broadcast
uart:~$ bt adv-data dev-name discov
uart:~$ bt per-adv-data

# Enable periodic advertising (extended advertising is enabled as part of handover)
uart:~$ bt per-adv on
```

Handover broadcast to unicast This command hands over one or more unicast streams from broadcast to unicast.

```
uart:~$ bt init
uart:~$ bap init
uart:~$ bt connect <addr>

# Discover necessary services
uart:~$ bap discover
uart:~$ cap_initiator discover
uart:~$ bap_broadcast_assistant discover

# Create a non-connectable and non-scannable extended advertising set for broadcast
uart:~$ bt adv-create nconn-nscan ext-adv
uart:~$ bt per-adv-param

# Setup broadcast audio e.g. using the ac_12
uart:~$ cap_initiator ac_12
uart:~$ cap_initiator broadcast_start

# Set advertising data and enable advertising
uart:~$ bt adv-data dev-name discov
uart:~$ bt per-adv-data
uart:~$ bt per-adv on
uart:~$ bt adv-start

# Wait for broadcast sink to self-scan, or use broadcast reception to instruct␣
↪sink to sync

# Perform the handover
uart:~$ cap_handover broadcast_to_unicast

# Terminate the advertiser (optional)
uart:~$ bt adv-stop
uart:~$ bt per-adv off
uart:~$ bt adv-delete
```

Bluetooth: Call Control Profile Shell

Call Control Server The Call Control Server is a role that typically resides on devices that can make calls, including calls from apps such as Skype, e.g. (smart)phones and PCs, which are typically GAP Central devices.

Using the Call Control Server The Server can be controlled locally, or by a remote device (when in a call). For example a remote device may initiate a call to the server, or the Server may initiate a call to remote device, without a client.

For all commands that take an optional `index`, if the index is not supplied then it defaults to `0` which is the GTBS bearer.

```
ccp_call_control_server --help
ccp_call_control_server - Bluetooth CCP Call Control Server shell commands
Subcommands:
  init                    : Initialize CCP Call Control Server
  set_bearer_name         : Set bearer name [index] <name>
  get_bearer_name         : Get bearer name [index]
```

(continues on next page)

(continued from previous page)

```
  get_bearer_uci          : Get bearer UCI [index]
  set_bearer_tech         : Set bearer technology [index] <technology>
  get_bearer_tech         : Get bearer technology [index]
  set_bearer_uri_schemes  : Set bearer URI schemes supported list [index] <URI␣
↪schemes>
                           (e.g. "tel,skype")
  get_bearer_uri_schemes  : Get bearer URI schemes supported list [index]
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ ccp_call_control_server init
Registered GTBS bearer
Registered bearer[1]
uart:~$ bt connect P:xx:xx:xx:xx:xx:xx
```

Setting and getting the bearer name

```
uart:~$ ccp_call_control_server get_bearer_name
Bearer[0] name: Generic TBS
uart:~$ ccp_call_control_server set_bearer_name "New name"
Bearer[0] name: New name
uart:~$ ccp_call_control_server get_bearer_name
Bearer[0] name: New name
uart:~$ ccp_call_control_server get_bearer_name 1
Bearer[1] name: Telephone Bearer #1
uart:~$ ccp_call_control_server set_bearer_name 1 "New TBS name"
Bearer[1] name: New TBS name
uart:~$ ccp_call_control_server get_bearer_name 1
Bearer[1] name: New TBS name
```

Getting the bearer UCI

```
uart:~$ ccp_call_control_server get_bearer_uci
Bearer[0] UCI: un999
uart:~$ ccp_call_control_server get_bearer_uci 1
Bearer[1] UCI: skype
```

Setting and getting the bearer technology

```
uart:~$ ccp_call_control_server get_bearer_tech
Bearer[0] technology: 3G (0x01)
uart:~$ ccp_call_control_server set_bearer_tech 0x02
Bearer[0] new technology: 4G (0x02)
```

Setting and getting the bearer URI supported schemes list

```
uart:~$ ccp_call_control_server get_bearer_uri_schemes
Bearer[0] URI schemes supported list: tel,skype
uart:~$ ccp_call_control_server set_bearer_uri_schemes "tel,teamspeak"
Bearer[0] new URI schemes supported list: tel,teamspeak
```

Call Control Client The Call Control Client is a role that typically resides on resource constrained devices such as earbuds or headsets.

Using the Call Control Client The Client can control a remote CCP server device. For example a remote device may have an incoming call that can be accepted by the Client.

```
uart:~$ ccp_call_control_client --help
ccp_call_control_client - Bluetooth CCP Call Control Client shell commands
Subcommands:
  discover  : Discover GTBS and TBS on remote device
```

Example Usage when connected

```
uart:~$ ccp_call_control_client discover
Discovery completed with GTBS and 1 TBS bearers
```

```
uart:~$ ccp_call_control_client read_bearer_name
Bearer 0x20046254 name: Generic TBS
uart:~$ ccp_call_control_client read_bearer_name 1
Bearer 0x20046256 name: Telephone Bearer #1
uart:~$ ccp_call_control_client read_bearer_uci
Bearer 0x20046254 UCI: un999
uart:~$ ccp_call_control_client read_bearer_uci 1
Bearer 0x20046256 UCI: skype
```

Bluetooth: Coordinated Set Identification Profile Shell This document describes how to run the coordinated set identification functionality, both as a client and as a server. Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

Set Coordinator (Client) The client will typically be a resource-rich device, such as a smartphone or a laptop. The client is able to lock and release members of a coordinated set. While the coordinated set is locked, no other clients may lock the set.

To lock a set, the client must connect to each of the set members it wants to lock. This implementation will always try to connect to all the members of the set, and at the same time. Thus if the set size is 3, then `BT_MAX_CONN` shall be at least 3.

If the locks on set members shall persists through disconnects, it is necessary to bond with the set members. If you need to bond with multiple set members, make sure that `BT_MAX_PAIRED` is correctly configured.

Using the Set Coordinator When the Bluetooth stack has been initialized (`bt init`), and a set member device has been connected, the call control client can be initialized by calling `csip_set_coordinator init`, which will start a discovery for the TBS uuids and store the handles, and optionally subscribe to all notifications (default is to subscribe to all).

Once the client has connected and discovered the handles, then it can read the set information, which is needed to identify other set members. The client can then scan for and connect to the remaining set members, and once all the members has been connected to, it can lock and release the set.

It is necessary to enable `CONFIG_BT_CSIP_SET_COORDINATOR_LOG_LEVEL_DBG` to properly use the set coordinator.

```
csip_set_coordinator --help
csip_set_coordinator - Bluetooth CSIP_SET_COORDINATOR shell commands
```

(continues on next page)

(continued from previous page)

```
Subcommands:
   init              :Initialize CSIP_SET_COORDINATOR
   discover          :Run discover for CSIS on peer device [member_index]
   discover_members  :Scan for set members <set_pointer>
   lock_set          :Lock set
   release_set       :Release set
   lock              :Lock specific member [member_index]
   release           :Release specific member [member_index]
   lock_get          :Get the lock value of the specific member and instance
                     [member_index [inst_idx]]
```

Example usage

Setup

```
uart:~$ init
uart:~$ bt connect P:xx:xx:xx:xx:xx:xx
```

When connected Discovering sets on a device:

```
uart:~$ csip_set_coordinator init
<dbg> bt_csip_set_coordinator.primary_discover_func: [ATTRIBUTE] handle 0x0048
<dbg> bt_csip_set_coordinator.primary_discover_func: Discover complete, found 1␣
↪instances
<dbg> bt_csip_set_coordinator.discover_func: Setup complete for 1 / 1
Found 1 sets on device
uart:~$ csip_set_coordinator discover_sets
<dbg> bt_csip_set_coordinator.SIRK
36 04 9a dc 66 3a a1 a1 |6...f:..
1d 9a 2f 41 01 73 3e 01 |../A.s>.
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_size_cb:␣
↪2
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_lock_cb:␣
↪1
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_rank_cb: 1
Set size 2 (pointer: 0x566fdfe8)
```

Discover set members, based on the set pointer above:

```
uart:~$ csip_set_coordinator discover_members 0x566fdfe8
<dbg> bt_csip_set_coordinator.csip_found: Found CSIS advertiser with address␣
↪P:34:02:86:03:86:c0
<dbg> bt_csip_set_coordinator.is_set_member: hash: 0x33ccb1, prand 0x5bfe6a
<dbg> bt_csip_set_coordinator.is_discovered: P:34:02:86:03:86:c0
<dbg> bt_csip_set_coordinator.is_discovered: P:34:13:e8:b3:7f:9e
<dbg> bt_csip_set_coordinator.csip_found: Found member (2 / 2)
Discovered 2/2 set members
```

Lock set members:

```
uart:~$ csip_set_coordinator lock_set
<dbg> bt_csip_set_coordinator.bt_csip_set_coordinator_lock_set: Connecting to␣
↪P:34:02:86:03:86:c0
<dbg> bt_csip_set_coordinator.csip_set_coordinator_connected: Connected to␣
↪P:34:02:86:03:86:c0
```

(continues on next page)

(continued from previous page)

```
<dbg> bt_csip_set_coordinator.discover_func: Setup complete for 1 / 1
<dbg> bt_csip_set_coordinator.csip_set_coordinator_lock_set_init_cb:
<dbg> bt_csip_set_coordinator.SIRK
36 04 9a dc 66 3a a1 a1 |6...f:..
1d 9a 2f 41 01 73 3e 01 |../A.s>.
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_size_cb:␣
↪2
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_set_lock_cb:␣
↪1
<dbg> bt_csip_set_coordinator.csip_set_coordinator_discover_sets_read_rank_cb: 2
<dbg> bt_csip_set_coordinator.csip_set_coordinator_write_lowest_rank: Locking␣
↪member with rank 1
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was locked
<dbg> bt_csip_set_coordinator.csip_set_coordinator_write_lowest_rank: Locking␣
↪member with rank 2
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was locked
Set locked
```

Release set members:

```
uart:~$ csip_set_coordinator release_set
<dbg> bt_csip_set_coordinator.csip_set_coordinator_release_highest_rank: Releasing␣
↪member with rank 2
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was released
<dbg> bt_csip_set_coordinator.csip_set_coordinator_release_highest_rank: Releasing␣
↪member with rank 1
<dbg> bt_csip_set_coordinator.notify_func: Instance 0 lock was released
Set released
```

Coordinated Set Member (Server) The server on devices that are part of a set, consisting of at least two devices, e.g. a pair of earbuds.

Using the Set Member

```
csip_set_member --help
csip_set_member - Bluetooth CSIP set member shell commands
   Subcommands:
   register           : Initialize the service and register callbacks [size
                        <int>] [rank <int>] [not-lockable] [sirk <data>]
   lock               : Lock the set
   release            : Release the set [force]
   sirk               : Set the currently used SIRK <sirk>
   set_size_and_rank  : Set the currently used size and rank <size> <rank>
   get_info           : Get service info
   sirk_rsp           : Set the response used in SIRK requests <accept,
                        accept_enc, reject, oob>
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ csip_set_member register
```

Setting a new SIRK This command can modify the currently used SIRK. To get the new RSI to advertise on air, `bt adv-data` or `bt advertise` must be called again to set the new advertising data. If `CONFIG_BT_CSIP_SET_MEMBER_SIRK_NOTIFIABLE` is enabled, this will also notify connected clients.

```
uart:~$ csip_set_member sirk 00112233445566778899aabbccddeeff
SIRK updated
```

Setting a new set size and rank This command can modify the set size and rank of a service instance. This shall be done for all device in the set at the same time, and all devices shall have the same set size. The rank will be ignored if the set is not lockable, else the rank shall be <= the set size, and shall be unique for this device in the set.

```
uart:~$ csip_set_member set_size_and_rank 1 1
Set size and rank updated to 1 and 1
```

Getting the current info This command can get the currently used set info.

```
uart:~$ csip_set_member get_info
Info for 0x2003b0c8
        SIRK
00000000: 20 37 0a 00 95 c4 04 20  00 00 00 00 f1 79 09 00 | 7.....  .....y..|
        Set size: 2
        Rank: 1
        Lockable: true
        Locked: false
```

Bluetooth: Gaming Audio Profile Shell This document describes how to run the Gaming Audio Profile shell functionality. Unlike most other low-layer profiles, GMAP is a profile that exists and has a service (GMAS) on all devices. Thus both the initiator and acceptor (or central and peripheral) should do a discovery of the remote device's GMAS to see what GMAP roles and features they support.

Using the GMAP Shell When the Bluetooth stack has been initialized (`bt init`), the GMAS can be registered by by calling `gmap init`. It is also strongly suggested to enable BAP via `bap init`.

```
uart:~$ gmap --help
gmap - Bluetooth GMAP shell commands
Subcommands:
  init      : [none]
  set_role  : [ugt | ugg | bgr | bgs]
  discover  : [none]
  ac_1      : Unicast audio configuration 1
  ac_2      : Unicast audio configuration 2
  ac_3      : Unicast audio configuration 3
  ac_4      : Unicast audio configuration 4
  ac_5      : Unicast audio configuration 5
  ac_6_i    : Unicast audio configuration 6(i)
  ac_6_ii   : Unicast audio configuration 6(ii)
  ac_7_ii   : Unicast audio configuration 7(ii)
  ac_8_i    : Unicast audio configuration 8(i)
  ac_8_ii   : Unicast audio configuration 8(ii)
  ac_11_i   : Unicast audio configuration 11(i)
  ac_11_ii  : Unicast audio configuration 11(ii)
  ac_12     : Broadcast audio configuration 12
```

(continues on next page)

(continued from previous page)

```
  ac_13     : Broadcast audio configuration 13
  ac_14     : Broadcast audio configuration 14
```

The `set_role` command can be used to change the role at runtime, assuming that the device supports the role (the GMAP roles depend on some BAP configurations).

Example Central with GMAP UGT role Connect and establish Gaming Audio streams using Audio Configuration (AC) 3 (some logging has been omitted for clarity):

```
uart:~$ bt init
uart:~$ bap init
uart:~$ gmap init
uart:~$ bt connect <address>
uart:~$ gatt exchange-mtu
uart:~$ bap discover
Discover complete: err 0
uart:~$ cap_initiator discover
discovery completed with CSIS
uart:~$ gmap discover
gmap discovered for conn 0x2001c7d8:
     role 0x0f
     ugg_feat 0x07
     ugt_feat 0x6f
     bgs_feat 0x01
     bgr_feat 0x03
uart:~$ bap preset sink 32_2_gr
uart:~$ bap preset source 32_2_gs
uart:~$ gmap ac_3
Starting 2 streams for AC_3
stream 0x20020060 config operation rsp_code 0 reason 0
stream 0x200204d0 config operation rsp_code 0 reason 0
stream 0x200204d0 qos operation rsp_code 0 reason 0
stream 0x20020060 qos operation rsp_code 0 reason 0
Stream 0x20020060 enabled
stream 0x200204d0 enable operation rsp_code 0 reason 0
Stream 0x200204d0 enabled
stream 0x20020060 enable operation rsp_code 0 reason 0
Stream 0x20020060 started
stream 0x200204d0 start operation rsp_code 0 reason 0
Stream 0x200204d0 started
Unicast start completed
uart:~$ bap start_sine
Started transmitting on default_stream 0x20020060
[0]: stream 0x20020060 : TX LC3: 80 (seq_num 24800)
```

Bluetooth: Media Control Profile Shell This document describes how to run the media control functionality, using the shell, both as a client and as a server.

The media control server consists of to parts. There is a media player (mpl) that contains the logic to handle media, and there is a media control service (mcs) that serves as a GATT-based interface to the player. The media control client consists of one part, the GATT based client (mcc).

The media control server may include an object transfer service (ots) and the media control client may include an object transfer client (otc). When these are included, a richer set of functionality is available.

The media control server and client both implement the Generic Media Control Service (only), and do not use any underlying Media Control Services.

Note that in the examples below, in many cases the debug output has been removed and long outputs may have been shortened to make the examples shorter and clearer.

Also note that this documentation does not list all shell commands, it just shows examples of some of them. The set of commands is explorable from the mcc shell and the mpl shell, by typing `mcc` or `mpl` and pressing TAB. A help text for each command can be found by doing `mcc <command> help` or `mpl <command> help`.

Overview A media player has a *name* and an *icon* that allows identification of the player for the user.

The content of the media player is structured into tracks and groups. A media player has a number of groups. A group contains tracks and other groups. (In this implementation, a group only contains tracks, not other groups.) Tracks can be divided into segments.

An active player will have a *current track*. This is the track that is playing now (if the player is playing). The current track has a *title*, a *duration* (given in hundredths of a second) and a *position* - the current position of the player within the track.

There is also a *current group* (the group of the current track), a *parent group* (the parent group of the current group) and a *next track*.

The media player is in a *state*, which will be one of playing, paused, seeking or inactive. When playing, playback happens at a given *playback speed*, and the tracks are played according to the *playing order*, which is one of the *playing orders supported*. Track changes are signalled as notifications of the *track changed* characteristic. When seeking (fast forward or fast rewind), the track position is moved according to the *seeking speed*.

The *opcodes supported* tells which operations are supported by the player by writing to the *media control point*. There is also a *search control point* that allows to search for groups and tracks according to various criteria, with the result returned in the *search results*.

Finally, the *content control ID* is used to associate the media player with an audio stream.

Media Control Client (MCP) The media control client is used to control, and to get information from, a media control server. Control is done by writing to one of the two control points, or by writing to other writable characteristics. Getting information is done by reading characteristics, or by configuring the server to send notifications.

Using the media control client Before use, the media control client must be initialized by the command `mcc init`.

To achieve a connection to the peer, the `bt` commands must be used - `bt init` followed by `bt advertise on` (or `bt connect` if the server is advertising).

When the media control client is connected to a media control server, the client can discover the server's Generic Media Control Service, by giving the command `mcc discover_mcs`. This will store the handles of the service, and (optionally, but default) subscribe to all notifications.

After discovery, the media control client can read and write characteristics, including the media control point and the search control point.

Example usage

Setup

```
uart:~$ bt init
Bluetooth initialized

uart:~$ mcc init
MCC init complete

uart:~$ bt advertise on
Advertising started
Connected: R:F6:58:DC:27:F3:57
```

When connected Service discovery (GMCS and included OTS):

```
uart:~$ mcc discover_mcs
<dbg> bt_mcc.bt_mcc_discover_mcs: start discovery of MCS primary service
<dbg> bt_mcc.discover_primary_func: [ATTRIBUTE] handle 0x00ae
<dbg> bt_mcc.discover_primary_func: Primary discovery complete
<dbg> bt_mcc.discover_primary_func: UUID: 2800
<dbg> bt_mcc.discover_primary_func: UUID: 8fd7
<dbg> bt_mcc.discover_primary_func: Start discovery of MCS characteristics
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b0
<dbg> bt_mcc.discover_mcs_char_func: Player name, UUID: 8fa0
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b2
<dbg> bt_mcc.discover_mcs_char_func: Icon Object, UUID: 8fa1
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b4
<dbg> bt_mcc.discover_mcs_char_func: Icon URI, UUID: 8fa2
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00b6
<dbg> bt_mcc.discover_mcs_char_func: Track Changed, UUID: 8fa3
<dbg> bt_mcc.discover_mcs_char_func: Subscribing - handle: 0x00b6
[...]
<dbg> bt_mcc.discover_mcs_char_func: [ATTRIBUTE] handle 0x00ea
<dbg> bt_mcc.discover_mcs_char_func: Content Control ID, UUID: 8fb5
<dbg> bt_mcc.discover_mcs_char_func: Setup complete for MCS
<dbg> bt_mcc.discover_mcs_char_func: Start discovery of included services
<dbg> bt_mcc.discover_include_func: [ATTRIBUTE] handle 0x00af
<dbg> bt_mcc.discover_include_func: Include UUID 1825
<dbg> bt_mcc.discover_include_func: Discover include complete for MCS: OTS
<dbg> bt_mcc.discover_include_func: Start discovery of OTS characteristics
<dbg> bt_mcc.discover_otc_char_func: [ATTRIBUTE] handle 0x009c
<dbg> bt_mcc.discover_otc_char_func: OTS Features
[...]
<dbg> bt_mcc.discover_otc_char_func: [ATTRIBUTE] handle 0x00ac
<dbg> bt_mcc.discover_otc_char_func: Object Size
Discovery complete
<dbg> bt_otc.bt_otc_register: 0
<dbg> bt_otc.bt_otc_register: L2CAP psm 0x  25 sec_level 1 registered
<dbg> bt_mcc.discover_otc_char_func: Setup complete for OTS 1 / 1
uart:~$
```

Reading characteristics - the player name and the track duration as examples:

```
uart:~$ mcc read_player_name
Player name: My media player
4d 79 20 6d 65 64 69 61  20 70 6c 61 79 65 72    |My media  player

uart:~$ mcc read_track_duration
Track duration: 6300
```

Note that the value of some characteristics may be truncated due to being too long to fit in the ATT packet. Increasing the ATT MTU may help:

```
uart:~$ mcc read_track_title
Track title: Interlude #1 (Song for

uart:~$ gatt exchange-mtu
Exchange pending
Exchange successful

uart:~$ mcc read_track_title
Track title: Interlude #1 (Song for Alison)
```

Writing characteristics - track position as an example:

The track position is where the player "is" in the current track. Read the track position, change it by writing to it, confirm by reading it again.

```
uart:~$ mcc read_track_position
Track Position: 0

uart:~$ mcc set_track_position 500
Track Position: 500

uart:~$ mcc read_track_position
Track Position: 500
```

Controlling the player via the control point:

Writing to the control point allows the client to request the server to do operations like play, pause, fast forward, change track, change group and so on. Some operations (e.g. goto track) take an argument. Currently, raw opcode values are used as input to the control point shell command. These opcode values can be found in the mpl.h header file.

Send the play command (opcode "1"), the command to go to track (opcode "52") number three, and the pause command (opcode "2"):

```
uart:~$ mcc set_cp 1
Media State: 1
Operation: 1, result: 1
Operation: 1, param: 0

uart:~$ mcc set_cp 52 3
Track changed
Track title: Interlude #3 (Levanto Seventy)
Track duration: 7800
Track Position: 0
Current Track Object ID: 0x000000000104
Next Track Object ID: 0x000000000105
Operation: 52, result: 1
Operation: 52, param: 3

uart:~$ mcc set_cp 2
Media State: 2
Operation: 2, result: 1
Operation: 2, param: 0
```

Using the included object transfer client When object transfer is supported by both the client and the server, a larger set of characteristics is available. These include object IDs for the various

track and group objects. These IDs can be used to select and download the corresponding objects from the server's object transfer service.

Read the object ID of the current group object:

```
uart:~$ mcc read_current_group_obj_id
Current Group Object ID: 0x000000000107
```

Select the object with that ID:

```
uart:~$ mcc ots_select 0x107
Selecting object succeeded
```

Read the object's metadata:

```
uart:~$ mcc ots_read_metadata
Reading object metadata succeeded
<inf> bt_mcc: Object's meta data:
<inf> bt_mcc:        Current size    :35
<inf> bt_otc: --- Displaying 1 metadata records ---
<inf> bt_otc: Object ID: 0x000000000107
<inf> bt_otc: Object name: Joe Pass - Guitar Inte
<inf> bt_otc: Object Current Size: 35
<inf> bt_otc: Object Allocate Size: 35
<inf> bt_otc: Type: Group Obj Type
<inf> bt_otc: Properties:0x4
<inf> bt_otc:  - read permitted
```

Read the object itself:

The object received is a group object. It consists of a series of records consisting of a type (track or group) and an object ID.

```
uart:~$ mcc ots_read_current_group_object
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000102
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000103
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000104
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000105
<dbg> bt_mcc.on_group_content: Object type: 0, object  ID: 0x000000000106
```

Search The search control point takes as its input a sequence of search control items, each consisting of length, type (e.g. track name or artist name) and parameter (the track name or artist name to search for). If the result is successful, the search results are stored in an object in the object transfer service. The ID of the search results ID object can be read from the search results object ID characteristic. The search result object can then be downloaded as for the current group object above. (Note that the search results object ID is empty until a search has been done.)

This implementation has a working implementation of the search functionality interface and the server-side search control point parameter parsing. But the **actual searching is faked**, the same results are returned no matter what is searched for.

There are two commands for search, one (`mcc set_scp_raw`) allows to input the search control point parameter (the sequence of search control items) as a string. The other (`mcc set_scp_ioptest`) does preset IOP test searches and takes the round number of the IOP search control point test as a parameter.

Before the search, the search results object ID is empty

```
uart:~$ mcc read_search_results_obj_id
Search Results Object ID: 0x000000000000
<dbg> bt_mcc.mcc_read_search_results_obj_id_cb: Zero-length Search Results Object␣
↪ID
```

Run the search corresponding to the fourth round of the IOP test:

The search control point parameter generated by this command and parameter has one search control item. The length field (first octet) is 16 (0x10). (The length of the length field itself is not included.) The type field (second octet) is 0x04 (search for a group name). The parameter (the group name to search for) is "TSPX_Group_Name".

```
uart:~$ mcc set_scp_ioptest 4
Search string:
00000000: 10 04 54 53 50 58 5f 47  72 6f 75 70 5f 4e 61 6d |..TSPX_G roup_Nam|
00000010: 65                                               |e                 |
Search control point notification result code: 1
Search Results Object ID: 0x000000000107
Search Control Point set
```

After the successful search, the search results object ID has a value:

```
uart:~$ mcc read_search_results_obj_id
Search Results Object ID: 0x000000000107
```

Media Control Service (MCS) The media control service (mcs) and the associated media player (mpl) typically reside on devices that can provide access to, and serve, media content, like PCs and smartphones.

As mentioned above, the media player (mpl) has the player logic, while the media control service (mcs) has the GATT-based interface. This separation is done so that the media player can also be used without the GATT-based interface.

Using the media control service and the media player The media control service and the media player are in general controlled remotely, from the media control client.

Before use, the media control client must be initialized by the command `mpl init`.

As for the client, the `bt` commands are used for connecting - `bt init` followed by `bt connect <address> <address type>` (or `bt advertise on` if the server is advertising).

Example Usage

Setup

```
uart:~$ bt init
Bluetooth initialized

uart:~$ mpl init
[Large amounts of debug output]

uart:~$ bt connect R:F9:33:3B:67:D2:A7
Connection pending
Connected: R:F9:33:3B:67:D2:A7
```

When connected Control is done from the client.

The server will give debug output related to the various operations performed by the client.

Example: Debug output by the server when the client gives the "next track" command:

```
[00:13:29.932,373] <dbg> bt_mcs.control_point_write: Opcode: 49
[00:13:29.932,403] <dbg> bt_mpl.mpl_operation_set: opcode: 49, param: 536880068
[00:13:29.932,403] <dbg> bt_mpl.paused_state_operation_handler: Operation opcode:␣
↪49
[00:13:29.932,495] <dbg> bt_mpl.do_next_track: Track ID before: 0x000000000104
[00:13:29.932,586] <dbg> bt_mpl.do_next_track: Track ID after: 0x000000000105
[00:13:29.932,617] <dbg> bt_mcs.mpl_track_changed_cb: Notifying track change
[00:13:29.932,708] <dbg> bt_mcs.mpl_track_title_cb: Notifying track title:␣
↪Interlude #4 (Vesper Dreams)
[00:13:29.932,800] <dbg> bt_mcs.mpl_track_duration_cb: Notifying track duration:␣
↪13500
[00:13:29.932,861] <dbg> bt_mcs.mpl_track_position_cb: Notifying track position: 0
[00:13:29.933,044] <dbg> bt_mcs.mpl_current_track_id_cb: Notifying current track␣
↪ID: 0x000000000105
[00:13:29.933,258] <dbg> bt_mcs.mpl_next_track_id_cb: Notifying next track ID:␣
↪0x000000000106
[00:13:29.933,380] <dbg> bt_mcs.mpl_operation_cb: Notifying control point -␣
↪opcode: 49, result: 1
```

Some server commands are available. These commands force notifications of the various characteristics, for testing that the client receives notifications. The values sent in the notifications caused by these testing commands are independent of the media player, so they do not correspond the actual values of the characteristics nor to the actual state of the media player.

Example: Force (fake value) notification of the track duration:

```
uart:~$ mpl duration_changed_cb
[00:15:17.491,058] <dbg> bt_mcs.mpl_track_duration_cb: Notifying track duration:␣
↪12000
```

Bluetooth: Telephone Bearer Service Shell This document describes how to run the call control functionality, both as a client and as a (telephone bearer service (TBS)) server. Note that in the examples below, some lines of debug have been removed to make this shorter and provide a better overview.

Telephone Bearer Service Client The telephone bearer service client will typically exist on a resource restricted device, such as headphones, but may also exist on e.g. phones or laptops. The call control client will also thus typically be the advertiser. The client can control the states of calls on a server using the call control point.

It is necessary to have `CONFIG_BT_TBS_CLIENT_LOG_LEVEL_DBG` enabled for using the client interactively.

Using the telephone bearer service client When the Bluetooth stack has been initialized (`bt init`), and a device has been connected, the telephone bearer service client can discover TBS on the connected device calling `tbs_client discover`, which will start a discovery for the TBS UUIDs and store the handles, and optionally subscribe to all notifications (default is to subscribe to all).

Since a server may have multiple TBS instances, most of the tbs_client commands will take an index (starting from 0) as input. Joining calls require at least 2 call IDs, and all call indexes shall be on the same TBS instance.

A server will also have a GTBS instance, which is an abstraction layer for all the telephone bearers on the server. If the server has both GTBS and TBS, the client may subscribe and use either when sending requests if `BT_TBS_CLIENT_GTBS` is enabled.

```
tbs_client --help
tbs_client - Bluetooth TBS_CLIENT shell commands
Subcommands:
   discover                      :Discover TBS [subscribe]
   set_signal_reporting_interval  :Set the signal reporting interval
                                   [<{instance_index, gtbs}>] <interval>
   originate                     :Originate a call [<{instance_index, gtbs}>]
                                   <uri>
   terminate                     :terminate a call [<{instance_index, gtbs}>]
                                   <id>
   accept                        :Accept a call [<{instance_index, gtbs}>] <id>
   hold                          :Place a call on hold [<{instance_index,
                                   gtbs}>] <id>
   retrieve                      :Retrieve a held call [<{instance_index,
                                   gtbs}>] <id>
   read_provider_name            :Read the bearer name [<{instance_index,
                                   gtbs}>]
   read_bearer_uci               :Read the bearer UCI [<{instance_index, gtbs}>]
   read_technology               :Read the bearer technology [<{instance_index,
                                   gtbs}>]
   read_uri_list                 :Read the bearer's supported URI list
                                   [<{instance_index, gtbs}>]
   read_signal_strength          :Read the bearer signal strength
                                   [<{instance_index, gtbs}>]
   read_signal_interval          :Read the bearer signal strength reporting
                                   interval [<{instance_index, gtbs}>]
   read_current_calls            :Read the current calls [<{instance_index,
                                   gtbs}>]
   read_ccid                     :Read the CCID [<{instance_index, gtbs}>]
   read_status_flags             :Read the in feature and status value
                                   [<{instance_index, gtbs}>]
   read_uri                      :Read the incoming call target URI
                                   [<{instance_index, gtbs}>]
   read_call_state               :Read the call state [<{instance_index, gtbs}>]
   read_remote_uri               :Read the incoming remote URI
                                   [<{instance_index, gtbs}>]
   read_friendly_name            :Read the friendly name of an incoming call
                                   [<{instance_index, gtbs}>]
   read_optional_opcodes         :Read the optional opcodes [<{instance_index,
                                   gtbs}>]
```

In the following examples, notifications from GTBS are ignored, unless otherwise specified.

Example usage

Setup

```
uart:~$ bt init
uart:~$ bt advertise on
Advertising started
```

When connected Placing a call:

```
uart:~$ tbs_client discover
<dbg> bt_tbs_client.primary_discover_func: Discover complete, found 1 instances␣
↪(GTBS found)
```

(continues on next page)

(continued from previous page)

```
<dbg> bt_tbs_client.discover_func: Setup complete for 1 / 1 TBS
<dbg> bt_tbs_client.discover_func: Setup complete GTBS
uart:~$ tbs_client originate 0 tel:123
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the dialing␣
↪state with URI tel:123
<dbg> bt_tbs_client.call_cp_notify_handler: Status: success for the originate␣
↪opcode for call 0x00
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the alerting␣
↪state with URI tel:123
<call answered by peer device, and status notified by TBS server>
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the active state␣
↪with URI tel:123
```

Placing a call on GTBS:

```
uart:~$ tbs_client originate 0 tel:123
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the dialing␣
↪state with URI tel:123
<dbg> bt_tbs_client.call_cp_notify_handler: Status: success for the originate␣
↪opcode for call 0x00
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the alerting␣
↪state with URI tel:123
<call answered by peer device, and status notified by TBS server>
<dbg> bt_tbs_client.notify_handler: Index 0
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x01 is in the active state␣
↪with URI tel:123
```

It is necessary to set an outgoing caller ID before placing a call.

Accepting incoming call from peer device:

```
<dbg> bt_tbs_client.incoming_uri_notify_handler: tel:123
<dbg> bt_tbs_client.in_call_notify_handler: tel:456
<dbg> bt_tbs_client.friendly_name_notify_handler: Peter
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x05 is in the incoming␣
↪state with URI tel:456
uart:~$ tbs_client accept 0 5
<dbg> bt_tbs_client.call_cp_callback_handler: Status: success for the accept␣
↪opcode for call 0x05
<dbg> bt_tbs_client.current_calls_notify_handler: Call 0x05 is in the active state␣
↪with URI tel
```

Terminate call:

```
uart:~$ tbs_client terminate 0 5
<dbg> bt_tbs_client.termination_reason_notify_handler: ID 0x05, reason 0x06
<dbg> bt_tbs_client.call_cp_notify_handler: Status: success for the terminate␣
↪opcode for call 0x05
<dbg> bt_tbs_client.current_calls_notify_handler:
```

Telephone Bearer Service (TBS) The telephone bearer service is a service that typically resides on devices that can make calls, including calls from apps such as Skype, e.g. (smart)phones

and PCs.

It is necessary to have `CONFIG_BT_TBS_LOG_LEVEL_DBG` enabled for using the TBS server interactively.

Using the telephone bearer service TBS can be controlled locally, or by a remote device (when in a call). For example a remote device may initiate a call to the device with the TBS server, or the TBS server may initiate a call to remote device, without a TBS_CLIENT client. The TBS implementation is capable of fully controlling any call. Omitting an index for commands where a `<instance_index>` can be supplied, defaults to the GTBS bearer.

```
tbs --help
tbs - Bluetooth TBS shell commands
Subcommands:
   init                        :Initialize TBS
   authorize                   :Authorize the current connection
   accept                      :Accept call <call_index>
   terminate                   :Terminate call <call_index>
   hold                        :Hold call <call_index>
   retrieve                    :Retrieve call <call_index>
   originate                   :Originate call [<instance_index>] <uri>
   join                        :Join calls <id> <id> [<id> [<id> [...]]]
   incoming                    :Simulate incoming remote call [<{instance_index,
                                gtbs}>] <local_uri> <remote_uri>
                                <remote_friendly_name>
   remote_answer               :Simulate remote answer outgoing call <call_index>
   remote_retrieve             :Simulate remote retrieve <call_index>
   remote_terminate            :Simulate remote terminate <call_index>
   remote_hold                 :Simulate remote hold <call_index>
   set_bearer_provider_name    :Set the bearer provider name [<{instance_index,
                                gtbs}>] <name>
   set_bearer_technology       :Set the bearer technology [<{instance_index,
                                gtbs}>] <technology>
   set_bearer_signal_strength  :Set the bearer signal strength [<{instance_index,
                                gtbs}>] <strength>
   set_status_flags            :Set the bearer feature and status value
                                [<{instance_index, gtbs}>] <feature_and_status>
   set_uri_scheme              :Set the URI prefix list <bearer_idx>
↪<uri1[,uri2[,uri3[,...]]]>
   print_calls                 :Output all calls in the debug log
```

Example Usage

Setup

```
uart:~$ bt init
uart:~$ bt connect P:xx:xx:xx:xx:xx:xx
```

When connected Answering a call for a peer device originated by a client:

```
<dbg> bt_tbs.write_call_cp: Index 0: Processing the originate opcode
<dbg> bt_tbs.originate_call: New call with call index 1
<dbg> bt_tbs.write_call_cp: Index 0: Processed the originate opcode with status␣
↪success for call index 1
uart:~$ tbs remote_answer 1
TBS succeeded for call_id: 1
```

Incoming call from a peer device, accepted by client:

```
uart:~$ tbs incoming 0 tel:123 tel:456 Peter
TBS succeeded for call_id: 4
<dbg> bt_tbs.bt_tbs_remote_incoming: New call with call index 4
<dbg> bt_tbs.write_call_cp: Index 0: Processed the accept opcode with status␣
↪success for call index 4
```

Bluetooth: Telephone and Media Audio Profile Shell This document describes how to run the Telephone and Media Audio Profile functionality. Unlike most other low-layer profiles, TMAP is a profile that exists and has a service (TMAS) on all devices. Thus both the initiator and acceptor (or central and peripheral) should do a discovery of the remote device's TMAS to see what TMAP roles they support.

Using the TMAP Shell When the Bluetooth stack has been initialized (`bt init`), the TMAS can be registered by calling `tmap init`.

```
tmap --help
tmap - Bluetooth TMAP shell commands
Subcommands:
  init          :Initialize and register the TMAS
  discover      :Discover TMAS on remote device
```

Bluetooth: Public Broadcast Profile Shell This document describes how to run the Public Broadcast Profile functionality. PBP does not have an associated service. Its purpose is to enable a faster, more efficient discovery of Broadcast Sources that are transmitting audio with commonly used codec configurations.

Using the PBP Shell When the Bluetooth stack has been initialized (`bt init`), the Public Broadcast Profile is ready to run. To set the Public Broadcast Announcement features call `pbp set_features`.

```
pbp --help
pbp - Bluetooth PBP shell commands
Subcommands:
  set_features    :Set the Public Broadcast Announcement features
```

Bluetooth: A2DP Shell The `a2dp` command exposes parts of the A2DP API.

The following examples assume that you have two devices already connected.

A2DP Connection Demonstrate the flow of creating an A2DP connection:

- Both sides register A2DP callbacks using `a2dp register_cb`.
- Either side establishes an A2DP connection, which will create the AVDTP Signaling channel, using `a2dp connect`.
- Either side can get the ACL connection using `a2dp get_conn`.
- Either side can disconnect the A2DP connection using `a2dp disconnect`.

Device A (initiator)

```
uart:~$ a2dp register_cb
success
uart:~$ a2dp connect
```

(continues on next page)

(continued from previous page)

```
Bonded with XX:XX:XX:XX:XX:XX
Security changed: XX:XX:XX:XX:XX:XX level 2
a2dp connected
uart:~$ a2dp get_conn
a2dp conn is: 0xXXXXXXXX
uart:~$ a2dp disconnect
a2dp disconnected
```

Device B (acceptor)

```
uart:~$ a2dp register_cb
success
<input `a2dp connect` in initiator side>
Connected: XX:XX:XX:XX:XX:XX
Bonded with XX:XX:XX:XX:XX:XX
Security changed: XX:XX:XX:XX:XX:XX level 2
a2dp connected
<input `a2dp disconnect` in initiator side>
a2dp disconnected
```

Basic A2DP Operations Demonstrate the flow of basic A2DP operations:

- Source and Sink sides register stream endpoints using `a2dp register_ep source sbc` and `a2dp register_ep sink sbc`.
- Create an A2DP connection based on *a2dp connection*.
- Initiator discovers remote device's stream endpoints using `a2dp discover_peer_eps 0x0104`.
- Initiator configures the stream to create the stream after discovering remote endpoints using `a2dp configure`.
- Initiator establishes the stream using `a2dp establish`.
- Sink sends delay report using `a2dp send_delay_report`.
- Initiator starts the media using `a2dp start`.
- Source tests media sending using `a2dp send_media` to send one test packet data.
- Initiator suspends the media using `a2dp suspend`.
- Initiator releases the media using `a2dp release`.

> **Note**
>
> The initiator is the A2DP source role and the acceptor is the A2DP sink role in the following logs. The delay report can only be sent by the sink role. The media data can only be sent by the source role.

Device A (initiator)

```
uart:~$ a2dp register_ep source sbc
SBC source endpoint is registered
uart:~$ a2dp discover_peer_eps 0x0104
endpoint id: 1, (sink), (idle):
  codec type: SBC
  sample frequency:
          44100
```

(continues on next page)

(continued from previous page)

```
          48000
  channel mode:
          Mono
          Stereo
          Joint-Stereo
  Block Length:
          16
  Subbands:
          8
  Allocation Method:
          Loudness
  Bitpool Range: 18 - 35
uart:~$ a2dp configure
success to configure
stream configured
uart:~$ a2dp establish
success to establish
stream established
<input `a2dp send_delay_report` in sink side>
receive delay report and accept
received delay report: 1 1/10ms
uart:~$ a2dp start
success to start
stream started
uart:~$ a2dp send_media
frames num: 1, data length: 160
data: 1, 2, 3, 4, 5, 6 ......
uart:~$ a2dp suspend
success to suspend
stream suspended
uart:~$ a2dp release
success to release
stream released
```

Device B (acceptor)

```
uart:~$ a2dp register_ep sink sbc
SBC sink endpoint is registered
<input `a2dp configure` in initiator side>
receive requesting config and accept
sample rate 44100Hz
stream configured
<input `a2dp establish` in initiator side>
receive requesting establishment and accept
stream established
uart:~$ a2dp send_delay_report
success to send report delay
<input `a2dp start` in initiator side>
receive requesting start and accept
stream started
<input `a2dp send_media` in source side>
received, num of frames: 1, data length: 160
data: 1, 2, 3, 4, 5, 6 ......
<input `a2dp suspend` in initiator side>
receive requesting suspend and accept
stream suspended
```

(continues on next page)

(continued from previous page)

```
<input `a2dp release` in initiator side>
receive requesting release and accept
stream released
```

Abort Operation Demonstrate the abort operation:

- Establish an A2DP stream based on *basic a2dp operations*.
- Initiator aborts the stream using `a2dp abort`.

Device A (initiator)

```
uart:~$ a2dp abort
success to abort
stream released
```

Device B (acceptor)

```
<input `a2dp abort` in initiator side>
receive requesting abort and accept
stream released
```

Get Configuration and Reconfigure Operation Demonstrate the get configuration and reconfigure operations:

- Establish an A2DP stream based on *basic a2dp operations*.
- Initiator gets configuration using `a2dp get_config`.
- Initiator reconfigures the stream using `a2dp reconfigure`.

Device A (initiator)

```
uart:~$ a2dp get_config
get config result: 0
sample rate 44100Hz
uart:~$ a2dp reconfigure
success to configure
stream configured
```

Device B (acceptor)

```
<input `a2dp get_config` in initiator side>
receive get config request and accept
<input `a2dp reconfigure` in initiator side>
receive requesting reconfig and accept
sample rate 44100Hz
stream configured
```

Bluetooth: Classic: AVRCP Shell This document describes how to use the Bluetooth Classic AVRCP (Audio/Video Remote Control Profile) functionality via shell commands. The `avrcp` command exposes both Controller (CT) and Target (TG) roles for exercising AVRCP control and browsing features.

There are two sub-commands: `avrcp ct` and `avrcp tg`. The `avrcp ct` sub-command provides **Controller (CT)** functionality, and `avrcp tg` sub-command provides **Target (TG)** functionality.

Prerequisites Before running the `avrcp` shell, make sure your build enables Bluetooth Classic and the shell. An ACL BR/EDR connection to the peer device must be established first (typically via the general `bt` shell commands) before AVRCP control or browsing connections can be created.

Commands All commands can only be used after the ACL connection has been established except `avrcp ct register_cb` and `avrcp tg register_cb`.

The `avrcp` commands:

```
uart:~$ avrcp
avrcp - Bluetooth AVRCP shell commands
Subcommands:
  connect                : connect AVRCP
  disconnect             : disconnect AVRCP
  browsing_connect       : connect browsing AVRCP
  browsing_disconnect    : disconnect browsing AVRCP
  ct                     : AVRCP CT shell commands
  tg                     : AVRCP TG shell commands
```

The `avrcp ct` commands:

```
uart:~$ avrcp ct
ct - AVRCP CT shell commands
Subcommands:
  register_cb                 : register avrcp ct callbacks
  get_unit                    : get unit info
  get_subunit                 : get subunit info
  get_caps                    : get capabilities <cap_id: company or events>
  play                        : request a play at the remote player
  pause                       : request a pause at the remote player
  register_notification       : register notify <event_id> [playback_interval]
  set_browsed_player          : set browsed player <player_id>
  get_folder_items            : [none]
  change_path                 : [none]
  get_item_attrs              : get item attrs [scope]
  get_total_number_of_items   : get total number of items [scope]
  search                      : search [search_string]
  list_app_attrs              : [none]
  list_app_vals               : List App vals <attr_id>
  get_app_curr                : Get curr player app setting val [attr1] [attr2] ...
  set_app_val                 : Set app setting Val <attr1> <val1> [<attr2>
↪<val2>] ...
  get_app_attr_text           : Get app setting attrs text <attr1> [attr2] ...
  get_app_val_text            : Get setting vals Text <attr_id> <val1> [val2] ...
  inform_displayable_char     : Inform Displayable Character Set <charset_id1>␣
↪[charset_id2] ...
  inform_batt                 : Inform Battery Status Of CT <Battery status>
  set_absolute_volume         : set absolute volume <volume>
  get_element_attrs           : get element attrs [identifier] [attr1] [attr2] ...
  get_play_status             : [none]
  set_addressed_player        : set addressed player <player_id>
  play_item                   : PlayItem <scope> <uid_hex> <uid_counter>
  add_to_now_playing          : AddToNowPlaying <scope> <uid_hex> <uid_counter>
```

The `avrcp tg` commands:

```
uart:~$ avrcp tg
tg - AVRCP TG shell commands
Subcommands:
  register_cb                              : register avrcp tg callbacks
  send_unit_rsp                            : send unit info response
  send_subunit_rsp                         : [none]
  send_get_caps_rsp                        : send get capabilities response␣
↪[status]
  send_notification_rsp                    : send notify rsp <event_id> <type>␣
↪[value...]
  send_browsed_player_rsp                  : Send SetBrowsedPlayer response
  send_get_folder_items_rsp                : send get folder items [status]
  send_change_path_rsp                     : send change path [status]
  send_get_item_attrs_rsp                  : send get item attrs [status]
  send_get_total_number_of_items_rsp       : send get total number of items␣
↪[status]
  send_search_rsp                          : search [status]
  send_browsing_general_reject             : send browsing general reject␣
↪[reason]
  send_passthrough_rsp                     : send_passthrough_rsp <op/opvu>
↪<opid> <state>
  send_list_player_app_setting_attrs_rsp   : send attrs rsp <num> [attr_id...]
  send_list_player_app_setting_vals_rsp    : send vals rsp <num> [val_id...]
  send_get_curr_player_app_setting_val_rsp : send current vals rsp <num_pairs>␣
↪[attr val]...
  send_set_player_app_setting_val_rsp      : set app setting val rsp [status]
  send_get_player_app_setting_attr_text_rsp : send get player app setting attr␣
↪text rsp [status]
  send_get_player_app_setting_val_text_rsp : send get player app setting val␣
↪text rsp [status]
  send_inform_displayable_char_rsp         : send displayable char rsp [status]
  send_inform_batt_status_of_ct_rsp        : send inform batt rsp [status]
  send_get_element_attrs_rsp               : send get element attrs␣
↪response<large: 1>
  send_absolute_volume_rsp                 : send absolute volume rsp <volume>
  send_get_play_status_rsp                 : send get play status [status]
  send_set_addressed_player_rsp            : send set addressed player rsp␣
↪[status]
  send_play_item_rsp                       : send play item rsp [status]
  send_add_to_now_playing_rsp              : send add to now playing rsp [status]
```

Basic AVRCP Operations

Demonstrate the flow of basic AVRCP operations:

- Both sides register AVRCP callbacks using `avrcp ct register_cb` and `avrcp tg register_cb`.
- Create an AVRCP connection using `avrcp connect`.
- CT gets capabilities from TG using `avrcp ct get_caps events`.
- CT registers for notifications using `avrcp ct register_notification 0x01`.
- CT requests playback control using `avrcp ct play`.
- CT gets current play status using `avrcp ct get_play_status`.
- CT gets element attributes (metadata) using `avrcp ct get_element_attrs`.
- CT sets volume using `avrcp ct set_absolute_volume 50`.

- CT pauses playback using `avrcp ct pause`.

> **Note**
>
> CT (Controller) sends commands and TG (Target) responds. For notification-capable events, TG typically sends an **INTERIM** response immediately after registration to indicate the current value, and later a **CHANGED** response when the value actually changes.

Device A (CT - Controller)

```
uart:~$ avrcp ct register_cb
AVRCP CT callbacks registered
uart:~$ avrcp connect
AVRCP CT connected
uart:~$ avrcp ct get_caps events
GetCapabilities : status=0x04
Remote supported EventID = 0x01
Remote supported EventID = 0x02
Remote supported EventID = 0x0d
uart:~$ avrcp ct register_notification 0x01
Get capabilities command sent successfully: cap_id=events
Sent register notification event_id=0x01
AVRCP notification rsp: tid=0x01, status=0x04, event_id=0x01
 Notification type: INTERIM
 PLAYBACK_STATUS_CHANGED: status=0x00
uart:~$ avrcp ct play
Passthrough PRESSED command sent successfully: opid=0x44
Passthrough RELEASED command sent successfully: opid=0x44
<input `avrcp tg send_passthrough_rsp op play pressed` in TG side>
AVRCP passthrough command accepted, operation id = 0x44, state = 0
<input `avrcp tg send_passthrough_rsp op play released` in TG side>
AVRCP passthrough command accepted, operation id = 0x44, state = 1
<input `avrcp tg send_notification_rsp 0x01 changed 1` in TG side>
AVRCP notification rsp: tid=0x01, status=0x04, event_id=0x01
 Notification type: CHANGED
 PLAYBACK_STATUS_CHANGED: status=0x01
uart:~$ avrcp ct get_play_status
AVRCP GetPlayStatus
getplaystatus : status=0x04
GetPlayStatus: len=180000 ms, pos=30000 ms, status=0x01
 status: PLAYING
uart:~$ avrcp ct get_element_attrs
Requesting element attributes: identifier=0x0000000000000000, num_attrs=0
AVRCP CT get element attrs command sent
GetElementAttributes : status=0x04
AVRCP GetElementAttributes response received, tid=0x05, num_attrs=7
 Attr[0]: ID=0x00000001 (TITLE), charset=0x006a, len=11
   Value: "Test Title"
 Attr[1]: ID=0x00000002 (ARTIST), charset=0x006a, len=11
   Value: "Test Artist"
uart:~$ avrcp ct set_absolute_volume 50
set absolute volume absolute_volume=0x32
AVRCP set absolute volume rsp: tid=0x02, status=0x04, volume=0x32
uart:~$ avrcp ct pause
Passthrough PRESSED command sent successfully: opid=0x46
Passthrough RELEASED command sent successfully: opid=0x46
```

(continues on next page)

(continued from previous page)

```
<input `avrcp tg send_passthrough_rsp op pause pressed` in TG side>
AVRCP passthrough command accepted, operation id = 0x46, state = 0
<input `avrcp tg send_passthrough_rsp op pause released` in TG side>
AVRCP passthrough command accepted, operation id = 0x46, state = 1
```

Device B (TG - Target)

```
uart:~$ avrcp tg register_cb
AVRCP TG callbacks registered
<input `avrcp connect` in CT side>
AVRCP TG connected
<input `avrcp ct get_caps events` in CT side>
AVRCP get capabilities command received: cap_id 0x03 (EVENTS_SUPPORTED)
uart:~$ avrcp tg send_get_caps_rsp
Get capabilities response sent successfully
<input `avrcp ct register_notification 0x01` in CT side>
receive register notification request event_id=0x01
uart:~$ avrcp tg send_notification_rsp 0x01 interim 0
Sent notification rsp event_id=0x01 type=interim
<input `avrcp ct play` in CT side>
receive passthrough command: op_id=0x44 (PLAY)
uart:~$ avrcp tg send_passthrough_rsp op play pressed
Passthrough opid=0x44 (STANDARD), state=pressed sent successfully
uart:~$ avrcp tg send_passthrough_rsp op play released
Passthrough opid=0x44 (STANDARD), state=released sent successfully
uart:~$ avrcp tg send_notification_rsp 0x01 changed 1
Sent notification rsp event_id=0x01 type=changed
<input `avrcp ct get_play_status` in CT side>
receive get play status request
uart:~$ avrcp tg send_get_play_status_rsp
GetPlayStatus rsp sent
<input `avrcp ct get_element_attrs` in CT side>
AVRCP GetElementAttributes command received
uart:~$ avrcp tg send_get_element_attrs_rsp 0
Sending standard GetElementAttributes response (7 attrs)
GetElementAttributes response sent successfully
<input `avrcp ct set_absolute_volume 50` in CT side>
AVRCP set_absolute_volume_req: tid=0x06, absolute_volume=0x32
uart:~$ avrcp tg send_absolute_volume_rsp 50
Set absolute volume response sent successfully
<input `avrcp ct pause` in CT side>
AVRCP passthrough command received: opid = 0x46
uart:~$ avrcp tg send_passthrough_rsp op pause pressed
send passthrough response
uart:~$ avrcp tg send_passthrough_rsp op pause released
send passthrough response
```

AVRCP Connection The AVRCP profile supports both control and browsing connections. The control connection is used for basic remote control functionality, while the browsing connection allows browsing of media content.

Control Connection Establish AVRCP control connection:

1. Register callbacks (CT side):

```
uart:~$ avrcp ct register_cb
AVRCP CT callbacks registered
```

2. Register callbacks (TG side):

```
uart:~$ avrcp tg register_cb
AVRCP TG callbacks registered
```

3. Connect AVRCP:

```
uart:~$ avrcp connect
AVRCP CT connected
AVRCP TG connected
```

4. Disconnect AVRCP:

```
uart:~$ avrcp disconnect
AVRCP CT disconnected
AVRCP TG disconnected
```

Browsing Connection After control connection is established, browsing connection can be initiated:

1. Connect browsing:

```
uart:~$ avrcp browsing_connect
AVRCP browsing connect request sent
AVRCP CT browsing connected
AVRCP TG browsing connected
```

2. Disconnect browsing:

```
uart:~$ avrcp browsing_disconnect
AVRCP browsing disconnect request sent
AVRCP CT browsing disconnected
AVRCP TG browsing disconnected
```

Basic Playback Control Control playback from CT side:

Play Command

```
uart:~$ avrcp ct play
Passthrough PRESSED command sent successfully: opid=0x44
Passthrough RELEASED command sent successfully: opid=0x44
```

Pause Command

```
uart:~$ avrcp ct pause
Passthrough PRESSED command sent successfully: opid=0x46
Passthrough RELEASED command sent successfully: opid=0x46
```

Get Capabilities Query supported capabilities:

Company ID

```
uart:~$ avrcp ct get_caps company
Get capabilities command sent successfully: cap_id=company
```

(continues on next page)

(continued from previous page)

```
GetCapabilities : status=0x04
Remote CompanyID = 0x001958
```

Events Supported

```
uart:~$ avrcp ct get_caps events
Get capabilities command sent successfully: cap_id=events
GetCapabilities : status=0x04
Remote supported EventID = 0x01
Remote supported EventID = 0x02
Remote supported EventID = 0x03
Remote supported EventID = 0x04
Remote supported EventID = 0x0d
```

TG Response

```
uart:~$ avrcp tg send_get_caps_rsp
Sending company ID capability rsp: 0x001958
```

Media Metadata Operations

Get Element Attributes Retrieve metadata for the currently playing media:

CT Request

```
uart:~$ avrcp ct get_element_attrs
Requesting element attributes: identifier=0x0000000000000000, num_attrs=0
AVRCP CT get element attrs command sent
GetElementAttributes : status=0x04
AVRCP GetElementAttributes response received, tid=0x00, num_attrs=7
 Attr[0]: ID=0x00000001 (TITLE), charset=0x006a, len=11
   Value: "Test Title"
 Attr[1]: ID=0x00000002 (ARTIST), charset=0x006a, len=11
   Value: "Test Artist"
 Attr[2]: ID=0x00000003 (ALBUM), charset=0x006a, len=10
   Value: "Test Album"
 Attr[3]: ID=0x00000004 (TRACK_NUMBER), charset=0x006a, len=1
   Value: "1"
 Attr[4]: ID=0x00000005 (TOTAL_TRACKS), charset=0x006a, len=2
   Value: "10"
 Attr[5]: ID=0x00000006 (GENRE), charset=0x006a, len=4
   Value: "Rock"
 Attr[6]: ID=0x00000007 (PLAYING_TIME), charset=0x006a, len=6
   Value: "240000"
```

TG Response

```
uart:~$ avrcp tg send_get_element_attrs_rsp 0
Sending standard GetElementAttributes response (7 attrs)
GetElementAttributes response sent successfully
```

Play Status Get current playback status:

CT Request

```
uart:~$ avrcp ct get_play_status
AVRCP GetPlayStatus
getplaystatus : status=0x04
GetPlayStatus: len=180000 ms, pos=30000 ms, status=0x01
 status: PLAYING
```

TG Response

```
uart:~$ avrcp tg send_get_play_status_rsp
GetPlayStatus rsp sent
```

Volume Control Set absolute volume:

CT Set Volume

```
uart:~$ avrcp ct set_absolute_volume 50
set absolute volume absolute_volume=0x32
AVRCP set absolute volume rsp: tid=0x01, status=0x04, volume=0x32
```

TG Response

```
uart:~$ avrcp tg send_absolute_volume_rsp 50
Set absolute volume response sent successfully
```

Event Notifications Register for notifications and handle events:

Register for Volume Change Notification CT Register

```
uart:~$ avrcp ct register_notification 0x0d
Sent register notification event_id=0x0d
AVRCP notification rsp: tid=0x02, status=0x04, event_id=0x0d
 Notification type: INTERIM
 VOLUME_CHANGED: absolute_volume=0x0a
AVRCP notify_changed_cb received: event_id=0x0d
 Notification type: CHANGED
 VOLUME_CHANGED: absolute_volume=0x14
```

TG Send Notification

```
uart:~$ avrcp tg send_notification_rsp 0x0d interim 10
Sent notification rsp event_id=0x0d type=interim

uart:~$ avrcp tg send_notification_rsp 0x0d changed 20
Sent notification rsp event_id=0x0d type=changed
```

Browsing Operations

Set Browsed Player CT Request

```
uart:~$ avrcp ct set_browsed_player 1
AVRCP send set browsed player req
AVRCP set browsed player success, tid = 0
  UID Counter: 1
  Number of Items: 100
  Charset ID: 0x006A
```

(continues on next page)

(continued from previous page)

```
  Folder Depth: 1
  charset_id  : 0x006A
  Get folder Name (hex)  :
00000000: 4d 75 73 69 63                                    |Music            |
```

TG Response

```
uart:~$ avrcp tg send_browsed_player_rsp
Send set browsed player response, status = 0x04
```

Get Folder Items CT Request

```
uart:~$ avrcp ct get_folder_items
Sent GetFolderItems command
AVRCP get folder items success, tid = 1
  UID Counter: 1
  Number of Items: 1
Media Player Item:
  item_len   : 28
  player_id   : 1
  major_type  : 0x01
  sub_type    : 0x00000000
  play_status : 0x00
  charset_id  : 0x006A
  name_len    : 4
  charset_id  : 0x006A
  Name (hex)  :
00000000: 44:65:6d:6f                                       |Demo
```

TG Response

```
uart:~$ avrcp tg send_get_folder_items_rsp
TG: Sent GetFolderItems response
```

Player Application Settings List and get configure player application settings:

List Available Settings CT Request

```
uart:~$ avrcp ct list_app_attrs
Sent list player app setting attrs
list player app setting attrs : status=0x04
attr =0x01 (EQUALIZER)
attr =0x02 (REPEAT_MODE)
```

TG Response

```
uart:~$ avrcp tg send_list_player_app_setting_attrs_rsp 2 0x01 0x02
TG: Sent list player app setting attrs response
```

Get Current Settings CT Request

```
uart:~$ avrcp ct get_app_curr 1 2
Sent get_curr_player_app_setting_val num=2
get curr player app setting val : status=0x04
```

(continues on next page)

(continued from previous page)

```
attr_id :1 val 1
attr_id :2 val 1
```

TG Response

```
uart:~$ avrcp tg send_get_curr_player_app_setting_val_rsp 2 0x01 0x01 0x02 0x02
TG: Send get curr player app setting val rsp (num=2)
```

Bluetooth: Classic: GOEP Shell This document describes how to run the Bluetooth Classic GOEP functionality. The goep command exposes the Bluetooth Classic GOEP Shell commands.

Commands The goep commands:

```
uart:~$ goep
goep - Bluetooth GOEP shell commands
Subcommands:
  register-rfcomm    : <channel>
  connect-rfcomm     : <channel>
  disconnect-rfcomm  :
  register-l2cap     : <psm>
  connect-l2cap      : <psm>
  disconnect-l2cap   :
  alloc-buf          : Alloc tx buffer
  release-buf        : Free allocated tx buffer
  add-header         : Adding header sets
  client             : Client sets
  server             : Server sets
```

The goep `client` commands:

```
uart:~$ goep client
client - Client sets
Subcommands:
  conn     : <mopl>
  disconn  :
  put      : <final: true, false>
  get      : <final: true, false>
  abort    :
  setpath  : [parent] [create]
  action   : <final: true, false>
```

The goep `server` commands:

```
uart:~$ goep server
server - Server sets
Subcommands:
  reg      : [UUID 128]
  unreg    :
  conn     : <rsp: continue, success, error> <mopl> [rsp_code]
  disconn  : <rsp: continue, success, error> [rsp_code]
  put      : <rsp: continue, success, error> [rsp_code]
  get      : <rsp: continue, success, error> [rsp_code]
  abort    : <rsp: continue, success, error> [rsp_code]
  setpath  : <rsp: continue, success, error> [rsp_code]
  action   : <rsp: continue, success, error> [rsp_code]
```

Connect GOEP Transport The ACL connection should be established before creating the GOEP transport connection.

The transport is based on L2CAP Channel:

L2CAP Server

```
uart:~$ goep register-l2cap 0
L2CAP server (psm 1001) is registered
Security changed: XX:XX:XX:XX:XX:XX level 2
GOEP 0x20005600 transport connected on 0x20004dc8
uart:~$
```

L2CAP Client

```
uart:~$ goep connect-l2cap 1001
GOEP L2CAP connection pending
Security changed: XX:XX:XX:XX:XX:XX level 2
GOEP 0x20005600 transport connected on 0x20004dc8
uart:~$
```

The transport is based on RFCOMM Channel:

RFCOMM Server

```
uart:~$ goep register-rfcomm 0
RFCOMM server (channel 06) is registered
Security changed: XX:XX:XX:XX:XX:XX level 2
GOEP 0x20005600 transport connected on 0x20004dc8
uart:~$
```

RFCOMM Client

```
uart:~$ goep connect-rfcomm 6
GOEP RFCOMM connection pending
Security changed: XX:XX:XX:XX:XX:XX level 2
GOEP 0x20005600 transport connected on 0x20004dc8
uart:~$
```

Disconnect GOEP transport The transport is based on L2CAP Channel:

One Side

```
GOEP 0x20005600 transport disconnected
uart:~$
```

Another Side

```
uart:~$ goep disconnect-l2cap
GOEP L2CAP disconnection pending
GOEP 0x20005600 transport disconnected
uart:~$
```

The transport is based on RFCOMM Channel:

One Side

```
GOEP 0x20005600 transport disconnected
uart:~$
```

Another Side

```
uart:~$ goep disconnect-rfcomm
GOEP RFCOMM disconnection pending
GOEP 0x20005600 transport disconnected
uart:~$
```

Connect to OBEX Server OBEX Server

```
uart:~$ goep server reg
uart:~$
OBEX server 0x20005850 conn req, version 10, mopl 00ff
uart:~$ goep server conn success 255
uart:~$
```

OBEX Client

```
uart:~$ goep client conn 255
OBEX client 0x20005818 conn rsp, rsp_code Success, version 10, mopl 00ff
uart:~$
```

Disconnect from OBEX Server OBEX Server

```
OBEX server 0x20005850 disconn req
uart:~$ goep server disconn success
uart:~$
```

OBEX Client

```
uart:~$ goep client disconn
OBEX client 0x20005818 disconn rsp, rsp_code Success
uart:~$
```

OBEX Put Operation OBEX Server

```
uart:~$
OBEX server 0x20005850 put req, final false, data len 12
HI c3 Len 4
00000000: 00 00 00 09                                      |....             |
HI 48 Len 4
00000000: 12 34 56 78                                      |.4Vx             |
uart:~$ goep server put continue
OBEX server 0x20005850 put req, final true, data len 8
HI 49 Len 5
00000000: 12 34 56 78 90                                   |.4Vx.            |
uart:~$ goep server put success
uart:~$
```

OBEX Client

```
uart:~$ goep alloc-buf
uart:~$ goep add-header len 9
uart:~$ goep add-header body 12345678
uart:~$ goep client put false
OBEX client 0x20005818 put rsp, rsp_code Continue, data len 0
uart:~$ goep alloc-buf
uart:~$ goep add-header end_body 1234567890
uart:~$ goep client put true
```

(continues on next page)

(continued from previous page)

```
OBEX client 0x20005818 put rsp, rsp_code Success, data len 0
uart:~$
```

OBEX Get Operation OBEX Server

```
uart:~$ goep alloc-buf
uart:~$ goep add-header len 9
uart:~$ goep add-header body 12345678
uart:~$ goep server get continue
OBEX server 0x20005850 get req, final true, data len 0
uart:~$ goep alloc-buf
uart:~$ goep add-header end_body 1234567890
uart:~$
uart:~$ goep server get success
uart:~$
```

OBEX Client

```
uart:~$ goep client get true
OBEX client 0x20005818 get rsp, rsp_code Continue, data len 12
HI c3 Len 4
00000000: 00 00 00 09                                      |....             |
HI 48 Len 4
00000000: 12 34 56 78                                      |.4Vx             |
uart:~$ goep client get true
OBEX client 0x20005818 get rsp, rsp_code Success, data len 8
HI 49 Len 5
00000000: 12 34 56 78 90                                   |.4Vx.            |
uart:~$
```

Bluetooth: Classic: HFP Shell This document describes how to run the Bluetooth Classic HFP functionality. The `hfp` command exposes the Bluetooth Classic HFP Shell commands.

There are two sub-commands, `hfp hf` and `hfp ag`.

The `hfp hf` is for Hands-Free Profile (HF) functionality, and the `hfp ag` is for Audio Gateway (AG) functionality.

Commands All commands can only be used after the ACL connection has been established except `hfp hf reg` and `hfp ag reg`.

The `hfp` commands:

```
uart:~$ hfp
hfp - Bluetooth HFP shell commands
Subcommands:
  hf  : HFP HF shell commands
  ag  : HFP AG shell commands
```

The `hfp hf` commands:

```
uart:~$ hfp hf
hf - HFP HF shell commands
Subcommands:
  reg                           : [none]
  connect                       : <channel>
```

(continues on next page)

(continued from previous page)

```
  disconnect                    : [none]
  sco_disconnect                : [none]
  cli                           : <enable/disable>
  vgm                           : <gain>
  vgs                           : <gain>
  operator                      : [none]
  audio_connect                 : [none]
  auto_select_codec             : <enable/disable>
  select_codec                  : Codec ID
  set_codecs                    : Codec ID Map
  accept                        : <call index>
  reject                        : <call index>
  terminate                     : <call index>
  hold_incoming                 : <call index>
  query_respond_hold_status     : [none]
  number_call                   : <phone number>
  memory_dial                   : <memory location>
  redial                        : [none]
  turn_off_ecnr                 : [none]
  call_waiting_notify           : <enable/disable>
  release_all_held              : [none]
  set_udub                      : [none]
  release_active_accept_other   : [none]
  hold_active_accept_other      : [none]
  join_conversation             : [none]
  explicit_call_transfer        : [none]
  release_specified_call        : <call index>
  private_consultation_mode     : <call index>
  voice_recognition             : <activate/deactivate>
  ready_to_accept_audio         : [none]
  request_phone_number          : [none]
  transmit_dtmf_code            : <call index> <code(set 0-9, #,*,A-D)>
  query_subscriber              : [none]
  indicator_status              : <Activate/deactivate AG indicators bitmap>
  enhanced_safety               : <enable/disable>
  battery                       : <level>
```

The `hfp ag` commands:

```
uart:~$ hfp ag
ag - HFP AG shell commands
Subcommands:
  reg                      : [none]
  connect                  : <channel>
  disconnect               : [none]
  sco_disconnect           : [none]
  ongoing_calls            : <yes or no>
  set_ongoing_calls        : <number> <type> <status> <dir> [all]
  remote_incoming          : <number>
  hold_incoming            : <number>
  remote_reject            : <call index>
  remote_accept            : <call index>
  remote_terminate         : <call index>
  remote_ringing           : <call index>
  outgoing                 : <number>
  reject                   : <call index>
```

(continues on next page)

(continued from previous page)

```
  accept                  : <call index>
  hold                    : <call index>
  retrieve                : <call index>
  terminate               : <call index>
  vgm                     : <gain>
  vgs                     : <gain>
  operator                : <mode> <operator>
  audio_connect           : <codec id>
  inband_ringtone         : <enable/disable>
  explicit_call_transfer  : [none]
  voice_recognition       : <activate/deactivate>
  vre_state               : <[R-ready][S-send][P-processing]>
  vre_text                : <[R-ready][S-send][P-processing]> <id> <type>
                            <operation> <text string>
  subscriber              : <empty/notempty>
  signal_strength         : <signal strength>
  roaming_status          : <roaming status>
  battery_level           : <battery level>
  service_availability    : <yes/no>
  hf_indicator            : <indicator> <enable/disable>
```

HFP AG SLC The `hfp ag` subcommand provides functionality for HFP AG in Bluetooth Classic.

1. Register HFP AG:

```
uart:~$ hfp ag reg
```

2. Connect to HFP HF:

```
uart:~$ hfp ag connect 1
```

3. Connection is established with the HF device:

```
Security changed: XX:XX:XX:XX:XX:XX level 2
AG received codec id bit map 2
AG connected
AG received vgm 0
AG received vgs 0
```

4. Disconnect from HFP HF:

```
uart:~$ hfp ag disconnect
```

5. Connection is broken:

```
AG disconnected
```

HFP HF SLC The `hfp hf` subcommand provides functionality for HFP HF in Bluetooth Classic.

1. Register HFP HF:

```
uart:~$ hfp hf reg
```

2. Connect to HFP AG:

```
uart:~$ hfp hf connect 2
```

3. Connection is established with the AG device:

```
Security changed: XX:XX:XX:XX:XX:XX level 2
HF service 0
HF signal 0
HF roam 0
HF battery 0
HF ring: in-band
HF connected
```

4. Disconnect from HFP HF:

```
uart:~$ hfp hf disconnect
```

5. Connection is broken:

```
HF disconnected
```

Call outgoing Place a call with the Phone number supplied by the AG:

Outgoing Call Sequence on AG side

```
uart:~$ hfp ag outgoing 123456
AG outgoing call 0x20007690, number 123456
AG SCO connected 0x20005248
AG SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
uart:~$ hfp ag remote_ringing 0
AG call 0x20007690 start ringing mode 1
uart:~$ hfp ag remote_accept 0
AG call 0x20007690 accept
```

Outgoing Call Sequence on HF side

```
uart:~$ hfp hf auto_select_codec enable
HF call 0x20007408 outgoing
codec negotiation: 1
codec auto selected: id 1
HF SCO connected 0x20005248
HF SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
HF remote call 0x20007408 start ringing
HF call 0x20007408 accepted
```

Place a call with the Phone number supplied by the HF:

Outgoing Call Sequence on AG side

```
uart:~$
AG number call
AG outgoing call 0x20007690, number 123456789
AG SCO connected 0x20005248
AG SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
```

(continues on next page)

(continued from previous page)

```
uart:~$ hfp ag remote_ringing 0
AG call 0x20007690 start ringing mode 1
uart:~$ hfp ag remote_accept 0
AG call 0x20007690 accept
```

Outgoing Call Sequence on HF side

```
uart:~$ hfp hf auto_select_codec enable
uart:~$ hfp hf number_call 123456789
HF start dialing call: err 0
HF call 0x20007408 outgoing
codec negotiation: 1
codec auto selected: id 1
HF SCO connected 0x20005248
HF SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
HF remote call 0x20007408 start ringing
HF call 0x20007408 accepted
```

Call incoming Answer incoming call from the AG:

Incoming Call Sequence on AG side

```
uart:~$ hfp ag remote_incoming 123456
AG incoming call 0x20007690, number 123456
AG call 0x20007690 start ringing mode 1
AG SCO connected 0x20005248
AG SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
uart:~$ hfp ag accept 0
AG call 0x20007690 accept
```

Incoming Call Sequence on HF side

```
uart:~$ hfp hf auto_select_codec enable
HF call 0x20007408 incoming
codec negotiation: 1
codec auto selected: id 1
HF SCO connected 0x20005248
HF SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
```

(continues on next page)

(continued from previous page)

```
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 accepted
```

Answer incoming call from the HF:

Incoming Call Sequence on AG side

```
uart:~$ hfp ag remote_incoming 123456
AG incoming call 0x20007690, number 123456
AG codec negotiation result 0
AG call 0x20007690 start ringing mode 1
AG SCO connected 0x20005248
AG SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
AG call 0x20007690 accept
```

Incoming Call Sequence on HF side

```
uart:~$ hfp hf auto_select_codec enable
HF call 0x20007408 incoming
codec negotiation: 1
codec auto selected: id 1
HF SCO connected 0x20005248
HF SCO info:
  SCO handle 0x0008
  SCO air mode 2
  SCO link type 2
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
HF call 0x20007408 ring
HF call 0x20007408 CLIP 123456 0
uart:~$ hfp hf accept 0
HF call 0x20007408 accepted
```

Call termination After the call (outgoing or incoming) is accepted, it can be terminated from either the AG (Audio Gateway) or HF (Hands-Free) side.

Terminate a call process from the AG:

Call termination on AG side

```
uart:~$ hfp ag terminate 0
AG call 0x20007690 terminate
AG SCO disconnected 0x20005248 (reason 22)
```

Call termination on HF side

```
HF call 0x20007408 terminated
HF SCO disconnected 0x20005248 (reason 22)
```

Terminate a call process from the HF:

Call termination on AG side

```
AG call 0x20007690 terminate
AG SCO disconnected 0x20005248 (reason 22)
```

Call termination on HF side

```
uart:~$ hfp hf terminate 0
HF call 0x20007408 terminated
HF SCO disconnected 0x20005248 (reason 22)
```

Terminate a call process from the remote:

Call termination on AG side

```
uart:~$ hfp ag remote_terminate 0
AG call 0x20007690 terminate
AG SCO disconnected 0x20005248 (reason 22)
```

Call termination on HF side

```
HF call 0x20007408 terminated
HF SCO disconnected 0x20005248 (reason 22)
```

Bluetooth: Classic: L2CAP Shell This document describes how to run the Bluetooth Classic L2CAP functionality. The `br l2cap` command exposes the Bluetooth Classic L2CAP Shell commands.

Commands The `br l2cap` commands:

```
uart:~$ br l2cap
l2cap - [none]
Subcommands:
  register    : <psm> <mode: none, ret, fc, eret, stream> [hold_credit]
                [mode_optional] [extended_control]
  connect     : <psm> <mode: none, ret, fc, eret, stream> [hold_credit]
                [mode_optional] [extended_control]
  disconnect  : [none]
  send        : [number of packets] [length of packet(s)]
  credits     : [none]
  echo        : L2CAP BR ECHO commands
  connless    : L2CAP connectionless commands
```

The `br l2cap echo` commands:

```
uart:~$ br l2cap echo
echo - L2CAP BR ECHO commands
```

(continues on next page)

(continued from previous page)

```
Subcommands:
  register    : [none]
  unregister  : [none]
  req         : <length of data>
  rsp         : <identifier> <length of data>
```

The `br l2cap connless` commands:

```
uart:~$ br l2cap connless
connless - L2CAP connectionless commands
Subcommands:
  register    : <psm> [sec level]
  unregister  : [none]
  send        : <psm> <length of data>
```

Connection-oriented L2CAP

1. [Server] Register L2CAP Server:

When the Bluetooth stack has been initialized (`bt init`), the L2CAP server can be registered by calling `br l2cap register`.

```
uart:~$ br l2cap register 1001 none
L2CAP psm 4097 registered
```

2. [Client] Create L2CAP connection:

The command can only be used after the ACL connection has been established.

```
uart:~$ br l2cap connect 1001 none
L2CAP connection pending
```

3. L2CAP connection is established:

```
uart:~$
Security changed: XX:XX:XX:XX:XX:XX level 2
Incoming BR/EDR conn 0x20004848
Channel 0x20000b18 connected
It is basic mode
```

3. Send L2CAP data to remote:

```
uart:~$ br l2cap send
Rem 0
```

4. L2CAP data is received:

```
uart:~$
Incoming data channel 0x20000b18 len 200
00000000: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000010: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000020: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
00000080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
```

(continues on next page)

(continued from previous page)

```
00000090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
000000A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
000000B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |........ ........|
000000C0: 00 00 00 00 00 00 00 00                          |........         |
```

5. Disconnect L2CAP connection:

```
uart:~$ br l2cap disconnect
```

6. L2CAP connection is broken:

```
Channel 0x20000b18 disconnected
```

L2CAP echo The echo subcommand provides functionality for L2CAP echo requests and responses in Bluetooth Classic.

The commands can only be used after the ACL connection has been established.

1. Listen for L2CAP echo request an L2CAP echo response:

```
uart:~$ br l2cap echo register
```

2. Stop listening for L2CAP echo request an L2CAP echo response:

```
uart:~$ br l2cap echo unregister
```

3. Send L2CAP echo request:

```
uart:~$ br l2cap echo req 1
```

4. Echo request is received:

```
Incoming ECHO REQ data identifier 4 len 1
00000000: 00                                                |.                |
```

5. Send L2CAP echo response:

```
uart:~$ br l2cap echo rsp 4 1
```

6. Echo response is received:

```
uart:~$
Incoming ECHO RSP data len 1
00000000: 00                                                |.                |
```

Connectionless L2CAP The connless subcommand provides functionality for connectionless L2CAP communication in Bluetooth Classic, allowing packet transmission without establishing a L2CAP connection.

The subcommand is controlled by `CONFIG_BT_L2CAP_CONNLESS`.

The commands can only be used after the ACL connection has been established.

1. Listen for connectionless L2CAP packets:

```
uart:~$ br l2cap connless register 1001
Register connectionless callbacks with PSM 0x1001
```

2. Send data

```
uart:~$ br l2cap connless send 1001 1
Sending connectionless data with PSM 0x1001
```

3. Connectionless data is received:

```
Incoming connectionless data psm 0x1001 len 1
00000000: 00                                             |.                |
```

Bluetooth: MAP Shell This document describes how to run the Bluetooth Classic MAP (Message Access Profile) shell functionality. The `map` command exposes the Bluetooth Classic MAP shell commands.

Commands

Main Commands The `map` commands:

```
uart:~$ map
map - Bluetooth MAP shell commands
Subcommands:
  add_header  : Adding header sets
  mce         : MCE commands
  mse         : MSE commands
```

MCE Commands The `map mce` commands:

```
uart:~$ map mce
mce - MCE commands
Subcommands:
  mas  : MCE MAS commands
  mns  : MCE MNS commands
```

MCE MAS Commands The `map mce mas` commands:

```
uart:~$ map mce mas
mas - MCE MAS commands
Subcommands:
  discover            : [none]
  select              : <instance_id>
  rfcomm_connect      : <channel> <instance_id> [supported_features]
  rfcomm_disconnect   : [none]
  l2cap_connect       : <psm> <instance_id> [supported_features]
  l2cap_disconnect    : [none]
  connect             : [none]
  disconnect          : [none]
  abort               : [none]
  set_folder          : <path: "/" | ".." | "../folder" | "folder">
  set_ntf_reg         : [none]
  get_folder_listing  : [srmp]
  get_msg_listing     : [name <folder_name>] [srmp]
  get_msg             : <msg_handle> [srmp]
  set_msg_status      : <msg_handle>
  push_msg            : [folder_name]
  update_inbox        : [none]
  get_mas_inst_info   : [srmp]
```

(continues on next page)

(continued from previous page)

```
  set_owner_status    : [none]
  get_owner_status    : [srmp]
  get_convo_listing   : [srmp]
  set_ntf_filter      : [none]
```

MCE MNS Commands The `map mce mns` commands:

```
uart:~$ map mce mns
mns - MCE MNS commands
Subcommands:
  register           : [none]
  rfcomm_register    : [none]
  rfcomm_disconnect  : [none]
  l2cap_register     : [none]
  l2cap_disconnect   : [none]
  connect            : <rsp: success, error> [rsp_code]
  disconnect         : <rsp: success, error> [rsp_code]
  abort              : <rsp: success, error> [rsp_code]
  send_event         : <rsp: noerror, error> [rsp_code] [srmp]
```

MSE Commands The `map mse` commands:

```
uart:~$ map mse
mse - MSE commands
Subcommands:
  mas  : MSE MAS commands
  mns  : MSE MNS commands
```

MSE MAS Commands The `map mse mas` commands:

```
uart:~$ map mse mas
mas - MSE MAS commands
Subcommands:
  select              : <instance_id>
  register            : [none]
  rfcomm_register     : [none]
  rfcomm_disconnect   : [none]
  l2cap_register      : [none]
  l2cap_disconnect    : [none]
  connect             : <rsp: success, error> [rsp_code]
  disconnect          : <rsp: success, error> [rsp_code]
  abort               : <rsp: success, error> [rsp_code]
  set_folder          : <rsp: success, error> [rsp_code]
  set_ntf_reg         : <rsp: noerror, error> [rsp_code]
  get_folder_listing  : <rsp: noerror, error> [rsp_code] [srmp]
  get_msg_listing     : <rsp: noerror, error> [rsp_code] [srmp]
  get_msg             : <rsp: noerror, error> [rsp_code] [srmp]
  set_msg_status      : <rsp: noerror, error> [rsp_code]
  push_msg            : <rsp: noerror, error> [rsp_code] [srmp]
  update_inbox        : <rsp: noerror, error> [rsp_code]
  get_mas_inst_info   : <rsp: noerror, error> [rsp_code] [srmp]
  set_owner_status    : <rsp: noerror, error> [rsp_code]
  get_owner_status    : <rsp: noerror, error> [rsp_code] [srmp]
  get_convo_listing   : <rsp: noerror, error> [rsp_code] [srmp]
  set_ntf_filter      : <rsp: noerror, error> [rsp_code]
```

MSE MNS Commands The `map mse mns` commands:

```
uart:~$ map mse mns
mns - MSE MNS commands
Subcommands:
  discover            : [none]
  rfcomm_connect      : <channel> [supported_features]
  rfcomm_disconnect   : [none]
  l2cap_connect       : <psm> [supported_features]
  l2cap_disconnect    : [none]
  connect             : [none]
  disconnect          : [none]
  abort               : [none]
  send_event          : [none]
```

Application Parameter Commands The `map add_header app_param` commands:

```
uart:~$ map add_header app_param
app_param - Application parameter commands
Subcommands:
  list                          : List all added application parameters
  clear                         : Clear all application parameters
  max_list_count                : <count: 0-65535>
  list_start_offset             : <offset: 0-65535>
  filter_msg_type               : <type_mask: hex>
  filter_period_begin           : <time: YYYYMMDDTHHMMSS[+/-HHMM]>
  filter_period_end             : <time: YYYYMMDDTHHMMSS[+/-HHMM]>
  filter_read_status            : <no_filter/unread/read>
  filter_recipient              : <recipient_address>
  filter_originator             : <originator_address>
  filter_priority               : <no_filter/high/non_high>
  attachment                    : <on/off>
  transparent                   : <on/off>
  retry                         : <on/off>
  new_message                   : <on/off>
  notification_status           : <on/off>
  mas_instance_id               : <id: 0-255>
  parameter_mask                : <mask: hex>
  folder_listing_size           : <size: 0-65535>
  listing_size                  : <size: 0-65535>
  subject_length                : <length: 0-255>
  charset                       : <native/utf8>
  fraction_request              : <first/next>
  fraction_deliver              : <more/last>
  status_indicator              : <read/deleted/extended>
  status_value                  : <yes/no>
  mse_time                      : <time: YYYYMMDDTHHMMSS[+/-HHMM]>
  database_id                   : <id_string: max 32 chars>
  convo_listing_ver             : <version_string: max 32 chars>
  presence_availability         : <unknown/offline/online/away/dnd/busy/meeting>
  presence_text                 : <text>
  last_activity                 : <time: YYYYMMDDTHHMMSS[+/-HHMM]>
  filter_last_activity_begin    : <time: YYYYMMDDTHHMMSS[+/-HHMM]>
  filter_last_activity_end      : <time: YYYYMMDDTHHMMSS[+/-HHMM]>
  chat_state                    : <unknown/inactive/active/composing/paused/gone>
  conversation_id               : <id_string: max 32 chars>
  folder_ver                    : <version_string: max 32 chars>
```

(continues on next page)

(continued from previous page)

```
  filter_msg_handle              : <handle_string: max 16 chars>
  notification_filter_mask       : <mask: hex>
  convo_parameter_mask           : <mask: hex>
  owner_uci                      : <uci_string>
  extended_data                  : <data_string>
  map_supported_features         : [none]
  message_handle                 : <handle_string: max 16 chars>
  modify_text                    : <prepend/replace>
```

Usage Examples

SDP Discovery

MCE Discover MSE The MCE can discover MSE services on the remote device:

```
uart:~$ map mce mas discover
Found RFCOMM channel 0x06
Found GOEP L2CAP PSM 0x1003
Found MAP profile version 0x0104
Found MAP features 0x007fffff
Found service name: MAP MAS 0
Found MAP instance ID 0
Found MAP MSG type 0x1f
uart:~$
```

MSE Discover MCE The MSE can discover MCE services on the remote device:

```
uart:~$ map mse mns discover
Found RFCOMM channel 0x06
Found GOEP L2CAP PSM 0x1003
Found MAP profile version 0x0104
Found MAP features 0x0077ffff
Found service name: MAP MNS
uart:~$
```

Connect MAP Transport The ACL connection should be established before creating the MAP transport connection.

MAS Transport Connection

RFCOMM Transport MCE

```
uart:~$ map mce mas rfcomm_connect 6 0 7fffff
Security changed: XX:XX:XX:XX:XX:XX level 2
MCE MAS RFCOMM connected: 0x20005a00, addr: XX:XX:XX:XX:XX:XX, instance_id: 0
Selected MCE MAS instance_id 0 as default
uart:~$
```

MSE

```
uart:~$ map mse mas register
MSE MAS server(s) registered
```

(continues on next page)

(continued from previous page)

```
uart:~$ map mse mas rfcomm_register
RFCOMM server (channel 06, mas_instance_id 0, map_supported_features 007fffff) is␣
↪registered
MSE MAS SDP record 0 registered
RFCOMM server (channel 07, mas_instance_id 1, map_supported_features 007fffff) is␣
↪registered
MSE MAS SDP record 1 registered
Security changed: XX:XX:XX:XX:XX:XX level 2
MSE MAS RFCOMM connected: 0x20005c00, addr: XX:XX:XX:XX:XX:XX, instance_id: 0
Selected MSE MAS instance_id 0 as default
uart:~$
```

L2CAP Transport MCE

```
uart:~$ map mce mas l2cap_connect 1003 0 7fffff
Security changed: XX:XX:XX:XX:XX:XX level 2
MCE MAS L2CAP connected: 0x20005a00, addr: XX:XX:XX:XX:XX:XX, instance_id: 0
Selected MCE MAS instance_id 0 as default
uart:~$
```

MSE

```
uart:~$ map mse mas register
MSE MAS server(s) registered
uart:~$ map mse mas l2cap_register
L2CAP server (psm 1003, mas_instance_id 0, map_supported_features 007fffff) is␣
↪registered
MSE MAS SDP record 0 registered
L2CAP server (psm 1005, mas_instance_id 1, map_supported_features 007fffff) is␣
↪registered
MSE MAS SDP record 1 registered
Security changed: XX:XX:XX:XX:XX:XX level 2
MSE MAS L2CAP connected: 0x20005c00, addr: XX:XX:XX:XX:XX:XX, instance_id: 0
Selected MSE MAS instance_id 0 as default
uart:~$
```

MNS Transport Connection

RFCOMM Transport MCE

```
uart:~$ map mce mns register
MCE MNS server(s) registered
uart:~$ map mce mns rfcomm_register
RFCOMM server (channel 06, map_supported_features 0077ffff) is registered
MCE MNS SDP record registered
MCE MNS RFCOMM connected: 0x20005b00, addr: XX:XX:XX:XX:XX:XX
uart:~$
```

MSE

```
uart:~$ map mse mns rfcomm_connect 6 77ffff
MSE MNS RFCOMM connected: 0x20005d00, addr: XX:XX:XX:XX:XX:XX
uart:~$
```

L2CAP Transport MCE

```
uart:~$ map mce mns register
MCE MNS server(s) registered
uart:~$ map mce mns l2cap_register
L2CAP server (psm 1003, map_supported_features 0077ffff) is registered
MCE MNS SDP record registered
MCE MNS L2CAP connected: 0x20005b00, addr: XX:XX:XX:XX:XX:XX
uart:~$
```

MSE

```
uart:~$ map mse mns l2cap_connect 1003 77ffff
MSE MNS L2CAP connected: 0x20005d00, addr: XX:XX:XX:XX:XX:XX
uart:~$
```

Disconnect MAP Transport

MAS Transport Disconnection

RFCOMM Transport (MCE initiates) MCE

```
uart:~$ map mce mas rfcomm_disconnect
MCE MAS RFCOMM disconnected: 0x20005a00
No connected MCE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

MSE

```
MSE MAS RFCOMM disconnected: 0x20005c00
No connected MSE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

RFCOMM Transport (MSE initiates) MCE

```
MCE MAS RFCOMM disconnected: 0x20005a00
No connected MCE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

MSE

```
uart:~$ map mse mas rfcomm_disconnect
MSE MAS RFCOMM disconnected: 0x20005c00
No connected MSE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

L2CAP Transport (MCE initiates) MCE

```
uart:~$ map mce mas l2cap_disconnect
MCE MAS L2CAP disconnected: 0x20005a00
No connected MCE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

MSE

```
MSE MAS L2CAP disconnected: 0x20005c00
No connected MSE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

L2CAP Transport (MSE initiates) MCE

```
MCE MAS L2CAP disconnected: 0x20005a00
No connected MCE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

MSE

```
uart:~$ map mse mas l2cap_disconnect
MSE MAS L2CAP disconnected: 0x20005c00
No connected MSE MAS instance available for conn XX:XX:XX:XX:XX:XX
uart:~$
```

MNS Transport Disconnection

RFCOMM Transport (MSE initiates) MCE

```
MCE MNS RFCOMM disconnected: 0x20005b00
uart:~$
```

MSE

```
uart:~$ map mse mns rfcomm_disconnect
MSE MNS RFCOMM disconnected: 0x20005d00
uart:~$
```

RFCOMM Transport (MCE initiates) MCE

```
uart:~$ map mce mns rfcomm_disconnect
MCE MNS RFCOMM disconnected: 0x20005b00
uart:~$
```

MSE

```
MSE MNS RFCOMM disconnected: 0x20005d00
uart:~$
```

L2CAP Transport (MSE initiates) MCE

```
MCE MNS L2CAP disconnected: 0x20005b00
uart:~$
```

MSE

```
uart:~$ map mse mns l2cap_disconnect
MSE MNS L2CAP disconnected: 0x20005d00
uart:~$
```

L2CAP Transport (MCE initiates) MCE

```
uart:~$ map mce mns l2cap_disconnect
MCE MNS L2CAP disconnected: 0x20005b00
uart:~$
```

MSE

```
MSE MNS L2CAP disconnected: 0x20005d00
uart:~$
```

Connect to MAP Server

MCE MAS Connect to MSE MAS MCE

```
uart:~$ map mce mas connect
MCE MAS 0x20005a00 conn rsp, rsp_code Success, version 10, mopl 0fff
HI 4a Len 16
00000000: bb 58 2b 40 42 0c 11 db  b0 de 08 00 20 0c 9a 66 |.X+@B... .... ..f|
HI cb Len 4
Conn ID: 00000001
uart:~$
```

MSE

```
MSE MAS 0x20005c00 conn req, version 10, mopl 0fff
HI 46 Len 16
00000000: bb 58 2b 40 42 0c 11 db  b0 de 08 00 20 0c 9a 66 |.X+@B... .... ..f|
HI 4c Len 6
T 29 L 4
00000000: 00 77 ff ff                                      |.w..              |
uart:~$ map mse mas connect success
uart:~$
```

MSE MNS Connect to MCE MNS MSE

```
uart:~$ map mse mns connect
MSE MNS 0x20005d00 conn rsp, rsp_code Success, version 10, mopl 0fff
HI 4a Len 16
00000000: bb 58 2b 41 42 0c 11 db  b0 de 08 00 20 0c 9a 66 |.X+AB... .... ..f|
HI cb Len 4
Conn ID: 00000001
uart:~$
```

MCE

```
MCE MNS 0x20005b00 conn req, version 10, mopl 0fff
HI 46 Len 16
00000000: bb 58 2b 41 42 0c 11 db  b0 de 08 00  20 0c 9a 66 |.X+AB... .... ..f|
uart:~$ map mce mns connect success
uart:~$
```

Disconnect from MAP Server

MCE MAS Disconnect from MSE MAS MCE

```
uart:~$ map mce mas disconnect
MCE MAS 0x20005a00 disconn rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 disconn req
HI c0 Len 4
Conn ID: 00000001
uart:~$ map mse mas disconnect success
uart:~$
```

MSE MNS Disconnect from MCE MNS MSE

```
uart:~$ map mse mns disconnect
MSE MNS 0x20005d00 disconn rsp, rsp_code Success
uart:~$
```

MCE

```
MCE MNS 0x20005b00 disconn req
HI c0 Len 4
Conn ID: 00000001
uart:~$ map mce mns disconnect success
uart:~$
```

MAP Operations

Set Notification Registration MCE

```
uart:~$ map mce mas set_ntf_reg
Adding default application parameters:
  Notification status: 0x01
MCE MAS 0x20005a00 set_ntf_reg rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 set_ntf_reg req, final true, data len 52
HI cb Len 4
Conn ID: 00000001
HI 42 Len 34
00000000: 78 2d 62 74 2f 4d 41 50  2d 4e 6f 74 69 66 69 63 |x-bt/MAP -Notific|
00000010: 61 74 69 6f 6e 52 65 67  69 73 74 72 61 74 69 6f |ationReg istratio|
00000020: 6e 00                                            |n.               |
HI 4c Len 3
T 0e L 1
00000000: 01                                               |.                |
HI 49 Len 1
00000000: 30                                               |0                |
uart:~$ map mse mas set_ntf_reg noerror
uart:~$
```

Set Folder MCE

```
uart:~$ map mce mas set_folder /
MCE MAS 0x20005a00 set_folder rsp, rsp_code Success
uart:~$ map mce mas set_folder telecom
MCE MAS 0x20005a00 set_folder rsp, rsp_code Success
uart:~$ map mce mas set_folder ..
MCE MAS 0x20005a00 set_folder rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 set_folder req, flags 2, data len 8
HI cb Len 4
Conn ID: 00000001
HI 01 Len 0
uart:~$ map mse mas set_folder success
MSE MAS 0x20005c00 set_folder req, flags 2, data len 24
HI cb Len 4
Conn ID: 00000001
HI 01 Len 16
00000000: 00 74 00 65 00 6c 00 65  00 63 00 6f 00 6d 00 00 |.t.e.l.e .c.o.m..|
uart:~$ map mse mas set_folder success
MSE MAS 0x20005c00 set_folder req, flags 3, data len 8
HI cb Len 4
Conn ID: 00000001
HI 01 Len 0
uart:~$ map mse mas set_folder success
uart:~$
```

Get Folder Listing

RFCOMM Transport When using RFCOMM transport, the same shell command can be called multiple times until the complete message is delivered.

MCE

```
uart:~$ map mce mas get_folder_listing
MCE MAS 0x20005a00 get_folder_listing rsp, rsp_code Continue, data len 252
HI 48 Len 249
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mas get_folder_listing
MCE MAS 0x20005a00 get_folder_listing rsp, rsp_code Success, data len 4
HI 49 Len 1
00000000: 00                                               |.                 |
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_folder_listing req, final true, data len 30
HI cb Len 4
Conn ID: 00000001
HI 42 Len 22
00000000: 78 2d 6f 62 65 78 2f 66  6f 6c 64 65 72 2d 6c 69 |x-obex/f older-li|
00000010: 73 74 69 6e 67 00                                |sting.            |
uart:~$ map mse mas get_folder_listing noerror
MSE MAS 0x20005c00 get_folder_listing req, final true, data len 0
```

(continues on next page)

(continued from previous page)

```
uart:~$ map mse mas get_folder_listing noerror
uart:~$
```

L2CAP Transport with SRM When using L2CAP transport with SRM enabled by default, the GET request does not need to be sent again.

MCE

```
uart:~$ map mce mas get_folder_listing
MCE MAS 0x20005c98 get_folder_listing rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 48 Len 247
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
MCE MAS 0x20005a00 get_folder_listing rsp, rsp_code Success, data len 6
HI 49 Len 3
00000000: 67 3e 00                                         |g>.              |
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_folder_listing req, final true, data len 32
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 22
00000000: 78 2d 6f 62 65 78 2f 66  6f 6c 64 65 72 2d 6c 69 |x-obex/f older-li|
00000010: 73 74 69 6e 67 00                                |sting.           |
...
uart:~$ map mse mas get_folder_listing noerror
uart:~$ map mse mas get_folder_listing noerror
uart:~$
```

L2CAP Transport with SRM and SRMP When using L2CAP transport with SRM enabled and SRMP enabled, the GET request needs to continue sending until SRMP is disabled.

MCE

```
uart:~$ map mce mas get_folder_listing srmp
MCE MAS 0x20005a00 get_folder_listing rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 48 Len 247
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
MCE MAS 0x20005a00 get_folder_listing rsp, rsp_code Success, data len 6
HI 49 Len 3
00000000: 67 3e 00                                         |g>.              |
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_folder_listing req, final true, data len 34
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
HI 42 Len 22
00000000: 78 2d 6f 62 65 78 2f 66  6f 6c 64 65 72 2d 6c 69 |x-obex/f older-li|
00000010: 73 74 69 6e 67 00                                |sting.            |
uart:~$ map mse mas get_folder_listing noerror
MSE MAS 0x20005c00 get_folder_listing req, final true, data len 0
uart:~$ map mse mas get_folder_listing noerror
uart:~$
```

Get Message Listing

RFCOMM Transport MCE

```
uart:~$ map mce mas get_msg_listing
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Continue, data len 252
HI 4c Len 29
T 0d L 1
00000000: 00                                               |.                 |
T 19 L 20
00000000: 32 30 32 35 30 31 30 31  54 31 32 30 30 30 30 2b |20250101 T120000+|
00000010: 30 30 30 30                                      |0000              |
T 12 L 2
00000000: 00 02                                            |..                |
HI 48 Len 217
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mas get_msg_listing
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Success, data len 52
HI 49 Len 49
00000000: 22 20 73 65 6e 74 3d 22  6e 6f 22 20 70 72 6f 74 |" sent=" no" prot|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_msg_listing req, final true, data len 32
HI cb Len 4
Conn ID: 00000001
HI 42 Len 21
00000000: 78 2d 62 74 2f 4d 41 50  2d 6d 73 67 2d 6c 69 73 |x-bt/MAP -msg-lis|
00000010: 74 69 6e 67 00                                   |ting.             |
HI 01 Len 0
uart:~$ map mse mas get_msg_listing noerror
Adding default application parameters:
  New message: 0
  MSE time: 20250101T120000+0000
  Listing size: 2
MSE MAS 0x20005c00 get_msg_listing req, final true, data len 0
```

(continues on next page)

(continued from previous page)

```
uart:~$ map mse mas get_msg_listing noerror
uart:~$
```

L2CAP Transport with SRM MCE

```
uart:~$ map mce mas get_msg_listing
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 4c Len 29
T 0d L 1
00000000: 00                                              |.                 |
T 19 L 20
00000000: 32 30 32 35 30 31 30 31  54 31 32 30 30 30 30 2b |20250101 T120000+|
00000010: 30 30 30 30                                     |0000              |
T 12 L 2
00000000: 00 02                                           |..                |
HI 48 Len 215
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
...
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Continue, data len 252
HI 48 Len 249
00000000: 2e 63 6f 6d 22 20 72 65  63 69 70 69 65 6e 74 5f |.com" re cipient_|
...
MCE MAS 0x20005c98 get_msg_listing rsp, rsp_code Success, data len 54
HI 49 Len 51
00000000: 65 73 22 20 73 65 6e 74  3d 22 6e 6f 22 20 70 72 |es" sent ="no" pr|
00000010: 6f 74 65 63 74 65 64 3d  22 6e 6f 22 2f 3e 0d 0a |otected= "no"/>..|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_msg_listing req, final true, data len 34
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 21
00000000: 78 2d 62 74 2f 4d 41 50  2d 6d 73 67 2d 6c 69 73 |x-bt/MAP -msg-lis|
00000010: 74 69 6e 67 00                                  |ting.             |
uart:~$ map mse mas get_msg_listing noerror
Adding default application parameters:
  New message: 0
  MSE time: 20250101T120000+0000
  Listing size: 2
uart:~$ map mse mas get_msg_listing noerror
uart:~$ map mse mas get_msg_listing noerror
uart:~$
```

L2CAP Transport with SRM and SRMP MCE

```
uart:~$ map mce mas get_msg_listing srmp
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Continue, data len 252
HI 97 Len 1
```

(continues on next page)

(continued from previous page)

```
OBEX SRM: 01
HI 4c Len 29
T 0d L 1
00000000: 00                                                |.                |
T 19 L 20
00000000: 32 30 32 35 30 31 30 31  54 31 32 30 30 30 30 2b |20250101 T120000+|
00000010: 30 30 30 30                                       |0000             |
T 12 L 2
00000000: 00 02                                             |..               |
HI 48 Len 215
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
...
uart:~$ map mce mas get_msg_listing
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Continue, data len 252
HI 48 Len 249
00000000: 2e 63 6f 6d 22 20 72 65  63 69 70 69 65 6e 74 5f |.com" re cipient_|
00000010: 6e 61 6d 65 3d 22 42 6f  62 22 20 72 65 63 69 70 |name="Bo b" recip|
...
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Success, data len 54
HI 49 Len 51
00000000: 65 73 22 20 73 65 6e 74  3d 22 6e 6f 22 20 70 72 |es" sent ="no" pr|
00000010: 6f 74 65 63 74 65 64 3d  22 6e 6f 22 2f 3e 0d 0a |otected= "no"/>..|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_msg_listing req, final true, data len 36
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
HI 42 Len 21
00000000: 78 2d 62 74 2f 4d 41 50  2d 6d 73 67 2d 6c 69 73 |x-bt/MAP -msg-lis|
00000010: 74 69 6e 67 00                                    |ting.            |
HI 01 Len 0
uart:~$ map mse mas get_msg_listing noerror
Adding default application parameters:
  New message: 0
  MSE time: 20250101T120000+0000
  Listing size: 2
MSE MAS 0x20005c00 get_msg_listing req, final true, data len 0
uart:~$ map mse mas get_msg_listing noerror
uart:~$ map mse mas get_msg_listing noerror
uart:~$
```

Get Message

RFCOMM Transport MCE

```
uart:~$ map mce mas get_msg 0000000000000001
Adding default application parameters:
  Attachment: OFF (0x00)
```

(continues on next page)

(continued from previous page)

```
  Charset: UTF-8 (0x01)
MCE MAS 0x20005a00 get_msg rsp, rsp_code Continue, data len 252
HI 48 Len 249
00000000: 42 45 47 49 4e 3a 42 4d  53 47 0d 0a 56 45 52 53 |BEGIN:BM SG..VERS|
00000010: 49 4f 4e 3a 31 2e 30 0d  0a 53 54 41 54 55 53 3a |ION:1.0. .STATUS:|
...
uart:~$ map mce mas get_msg 0000000000000001
MCE MAS 0x20005a00 get_msg rsp, rsp_code Success, data len 87
HI 49 Len 84
00000000: 3c 3e 3f 2f 0a 2d 20 4e  75 6d 62 65 72 73 3a 20 |<>?/.- N umbers: |
00000010: 30 31 32 33 34 35 36 37  38 39 0a 0a 45 6e 64 20 |01234567 89..End |
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_msg req, final true, data len 67
HI cb Len 4
Conn ID: 00000001
HI 42 Len 13
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 00          |x-bt/mes sage.   |
HI 01 Len 34
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                            |..               |
HI 4c Len 6
T 0a L 1
00000000: 00                                               |.                |
T 14 L 1
00000000: 01                                               |.                |
uart:~$ map mse mas get_msg noerror
MSE MAS 0x20005c00 get_msg req, final true, data len 0
uart:~$ map mse mas get_msg noerror
uart:~$
```

L2CAP Transport with SRM

MCE

```
uart:~$ map mce mas get_msg 0000000000000001
Adding default application parameters:
  Attachment: OFF (0x00)
  Charset: UTF-8 (0x01)
MCE MAS 0x20005a00 get_msg rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 48 Len 247
00000000: 42 45 47 49 4e 3a 42 4d  53 47 0d 0a 56 45 52 53 |BEGIN:BM SG..VERS|
00000010: 49 4f 4e 3a 31 2e 30 0d  0a 53 54 41 54 55 53 3a |ION:1.0. .STATUS:|
...
MCE MAS 0x20005a00 get_msg rsp, rsp_code Success, data len 89
HI 49 Len 86
00000000: 2c 2e 3c 3e 3f 2f 0a 2d  20 4e 75 6d 62 65 72 73 |,.<>?/.-  Numbers|
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_msg req, final true, data len 69
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 13
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 00          |x-bt/mes sage.   |
HI 01 Len 34
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                            |..               |
HI 4c Len 6
T 0a L 1
00000000: 00                                               |.                |
T 14 L 1
00000000: 01                                               |.                |
uart:~$ map mse mas get_msg noerror
uart:~$ map mse mas get_msg noerror
uart:~$
```

L2CAP Transport with SRM and SRMP

MCE

```
uart:~$ map mce mas get_msg 0000000000000001 srmp
Adding default application parameters:
  Attachment: OFF (0x00)
  Charset: UTF-8 (0x01)
MCE MAS 0x20005a00 get_msg rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 48 Len 247
00000000: 42 45 47 49 4e 3a 42 4d  53 47 0d 0a 56 45 52 53 |BEGIN:BM SG..VERS|
...
uart:~$ map mce mas get_msg 0000000000000001
MCE MAS 0x20005a00 get_msg rsp, rsp_code Continue, data len 252
HI 48 Len 249
00000000: 37 38 39 0d 0a 45 4e 44  3a 56 43 41 52 44 0d 0a |789..END :VCARD..|
00000010: 42 45 47 49 4e 3a 42 42  4f 44 59 0d 0a 43 48 41 |BEGIN:BB ODY..CHA|
...
MCE MAS 0x20005a00 get_msg rsp, rsp_code Success, data len 89
HI 49 Len 86
00000000: 2c 2e 3c 3e 3f 2f 0a 2d  20 4e 75 6d 62 65 72 73 |,.<>?/.-  Numbers|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_msg req, final true, data len 71
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
HI 42 Len 13
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 00          |x-bt/mes sage.   |
HI 01 Len 34
```

(continues on next page)

(continued from previous page)

```
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                           |..                |
HI 4c Len 6
T 0a L 1
00000000: 00                                              |.                 |
T 14 L 1
00000000: 01                                              |.                 |
uart:~$ map mse mas get_msg noerror
MSE MAS 0x20005c00 get_msg req, final true, data len 0
uart:~$ map mse mas get_msg noerror
uart:~$ map mse mas get_msg noerror
uart:~$
```

Set Message Status MCE

```
uart:~$ map mce mas set_msg_status 0000000000000001
Adding default application parameters:
  Status indicator: 0x00
  Status value: 0x00
MCE MAS 0x20005a00 set_msg_status rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 set_msg_status req, final true, data len 77
HI cb Len 4
Conn ID: 00000001
HI 01 Len 34
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                           |..                |
HI 42 Len 19
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 53 74 61 74 |x-bt/mes sageStat|
00000010: 75 73 00                                        |us.               |
HI 4c Len 6
T 17 L 1
00000000: 00                                              |.                 |
T 18 L 1
00000000: 00                                              |.                 |
HI 49 Len 1
00000000: 30                                              |0                 |
uart:~$ map mse mas set_msg_status noerror
uart:~$
```

Push Message

RFCOMM Transport MCE

```
uart:~$ map mce mas push_msg
Adding default application parameters:
  Charset: UTF-8 (0x01)
MCE MAS 0x20005a00 push_msg rsp, rsp_code Continue, data len 0
...
uart:~$ map mce mas push_msg
```

(continues on next page)

(continued from previous page)

```
MCE MAS 0x20005a00 push_msg rsp, rsp_code Success, data len 37
HI 01 Len 34
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                           |..                |
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 push_msg req, final false, data len 252
HI cb Len 4
Conn ID: 00000001
HI 42 Len 13
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 00         |x-bt/mes sage.    |
HI 01 Len 0
HI 4c Len 3
T 14 L 1
00000000: 01                                              |.                 |
HI 48 Len 219
00000000: 42 45 47 49 4e 3a 42 4d  53 47 0d 0a 56 45 52 53 |BEGIN:BM SG..VERS|
00000010: 49 4f 4e 3a 31 2e 30 0d  0a 53 54 41 54 55 53 3a |ION:1.0. .STATUS:|
...
uart:~$ map mse mas push_msg noerror
MSE MAS 0x20005c00 push_msg req, final true, data len 117
HI 49 Len 114
00000000: 61 63 74 65 72 73 3a 20  40 23 24 25 26 2a 28 29 |acters:  @#$%&*()|
...
uart:~$ map mse mas push_msg noerror
uart:~$
```

L2CAP Transport with SRM

MCE

```
uart:~$ map mce mas push_msg
Adding default application parameters:
  Charset: UTF-8 (0x01)
MCE MAS 0x20005a00 push_msg rsp, rsp_code Continue, data len 2
HI 97 Len 1
OBEX SRM: 01
...
uart:~$ map mce mas push_msg
uart:~$ map mce mas push_msg
MCE MAS 0x20005a00 push_msg rsp, rsp_code Success, data len 37
HI 01 Len 34
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                           |..                |
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 push_msg req, final false, data len 252
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
```

(continues on next page)

(continued from previous page)

```
OBEX SRM: 01
HI 42 Len 13
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 00          |x-bt/mes sage.   |
HI 01 Len 0
HI 4c Len 3
T 14 L 1
00000000: 01                                                |.                 |
HI 48 Len 217
00000000: 42 45 47 49 4e 3a 42 4d  53 47 0d 0a 56 45 52 53 |BEGIN:BM SG..VERS|
00000010: 49 4f 4e 3a 31 2e 30 0d  0a 53 54 41 54 55 53 3a |ION:1.0. .STATUS:|
...
uart:~$ map mse mas push_msg noerror
MSE MAS 0x20005c00 push_msg req, final false, data len 252
HI 48 Len 249
00000000: 54 46 2d 38 3a 2b 30 31  32 33 34 35 36 37 38 39 |TF-8:+01 23456789|
...
MSE MAS 0x20005c00 push_msg req, final true, data len 119
HI 49 Len 116
00000000: 61 72 61 63 74 65 72 73  3a 20 40 23 24 25 26 2a |aracters : @#$%&*|
...
uart:~$ map mse mas push_msg noerror
uart:~$
```

L2CAP Transport with SRM and SRMP

MCE

```
uart:~$ map mce mas push_msg
Adding default application parameters:
  Charset: UTF-8 (0x01)
MCE MAS 0x20005a00 push_msg rsp, rsp_code Continue, data len 4
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
...
uart:~$ map mce mas push_msg
MCE MAS 0x20005c98 push_msg rsp, rsp_code Continue, data len 0
uart:~$ map mce mas push_msg
MCE MAS 0x20005a00 push_msg rsp, rsp_code Success, data len 37
HI 01 Len 34
00000000: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 30 |.0.0.0.0 .0.0.0.0|
00000010: 00 30 00 30 00 30 00 30  00 30 00 30 00 30 00 31 |.0.0.0.0 .0.0.0.1|
00000020: 00 00                                             |..                |
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 push_msg req, final false, data len 252
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 13
00000000: 78 2d 62 74 2f 6d 65 73  73 61 67 65 00          |x-bt/mes sage.   |
HI 01 Len 0
HI 4c Len 3
```

(continues on next page)

(continued from previous page)

```
T 14 L 1
00000000: 01                                              |.                 |
HI 48 Len 217
00000000: 42 45 47 49 4e 3a 42 4d  53 47 0d 0a 56 45 52 53 |BEGIN:BM SG..VERS|
00000010: 49 4f 4e 3a 31 2e 30 0d  0a 53 54 41 54 55 53 3a |ION:1.0. .STATUS:|
...
uart:~$ map mse mas push_msg noerror srmp
MSE MAS 0x20005c00 push_msg req, final false, data len 252
HI 48 Len 249
00000000: 54 46 2d 38 3a 2b 30 31  32 33 34 35 36 37 38 39 |TF-8:+01 23456789|
...
uart:~$ map mse mas push_msg noerror
MSE MAS 0x20005c00 push_msg req, final true, data len 119
HI 49 Len 116
00000000: 61 72 61 63 74 65 72 73  3a 20 40 23 24 25 26 2a |aracters : @#$%&*|
...
uart:~$ map mse mas push_msg noerror
uart:~$
```

Update Inbox MCE

```
uart:~$ map mce mas update_inbox
MCE MAS 0x20005a00 update_inbox rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 update_inbox req, final true, data len 35
HI cb Len 4
Conn ID: 00000001
HI 42 Len 23
00000000: 78 2d 62 74 2f 4d 41 50  2d 6d 65 73 73 61 67 65 |x-bt/MAP -message|
00000010: 55 70 64 61 74 65 00                             |Update.           |
HI 49 Len 1
00000000: 30                                               |0                 |
uart:~$ map mse mas update_inbox noerror
uart:~$
```

Get MAS Instance Information

RFCOMM Transport MCE

```
uart:~$ map mce mas get_mas_inst_info
Adding default application parameters:
  Instance ID: 0
MCE MAS 0x20005a00 get_mas_inst_info rsp, rsp_code Success, data len 21
HI 49 Len 18
00000000: 4d 41 53 20 49 6e 73 74  61 6e 63 65 20 49 6e 66 |MAS Inst ance Inf|
00000010: 6f 00                                            |o.                |
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_mas_inst_info req, final true, data len 42
HI cb Len 4
```

(continues on next page)

(continued from previous page)

```
Conn ID: 00000001
HI 42 Len 28
00000000: 78 2d 62 74 2f 4d 41 53  49 6e 73 74 61 6e 63 65 |x-bt/MAS Instance|
00000010: 49 6e 66 6f 72 6d 61 74  69 6f 6e 00             |Informat ion.    |
HI 4c Len 3
T 0f L 1
00000000: 00                                               |.                |
uart:~$ map mse mas get_mas_inst_info noerror
uart:~$
```

L2CAP Transport with SRM MCE

```
uart:~$ map mce mas get_mas_inst_info
Adding default application parameters:
  Instance ID: 0
MCE MAS 0x20005a00 get_mas_inst_info rsp, rsp_code Success, data len 23
HI 97 Len 1
OBEX SRM: 01
HI 49 Len 18
00000000: 4d 41 53 20 49 6e 73 74  61 6e 63 65 20 49 6e 66 |MAS Inst ance Inf|
00000010: 6f 00                                            |o.               |
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_mas_inst_info req, final true, data len 44
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 28
00000000: 78 2d 62 74 2f 4d 41 53  49 6e 73 74 61 6e 63 65 |x-bt/MAS Instance|
00000010: 49 6e 66 6f 72 6d 61 74  69 6f 6e 00             |Informat ion.    |
HI 4c Len 3
T 0f L 1
00000000: 00                                               |.                |
uart:~$ map mse mas get_mas_inst_info noerror
uart:~$
```

L2CAP Transport with SRM and SRMP MCE

```
uart:~$ map mce mas get_mas_inst_info srmp
Adding default application parameters:
  Instance ID: 0
MCE MAS 0x20005a00 get_mas_inst_info rsp, rsp_code Success, data len 23
HI 97 Len 1
OBEX SRM: 01
HI 49 Len 18
00000000: 4d 41 53 20 49 6e 73 74  61 6e 63 65 20 49 6e 66 |MAS Inst ance Inf|
00000010: 6f 00                                            |o.               |
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_mas_inst_info req, final true, data len 46
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
HI 42 Len 28
00000000: 78 2d 62 74 2f 4d 41 53  49 6e 73 74 61 6e 63 65 |x-bt/MAS Instance|
00000010: 49 6e 66 6f 72 6d 61 74  69 6f 6e 00             |Informat ion.    |
HI 4c Len 3
T 0f L 1
00000000: 00                                               |.                |
uart:~$ map mse mas get_mas_inst_info noerror
uart:~$
```

Set Owner Status MCE

```
uart:~$ map mce mas set_owner_status
Adding default application parameters:
  Chat state: 0x00
MCE MAS 0x20005a00 set_owner_status rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 set_owner_status req, final true, data len 35
HI cb Len 4
Conn ID: 00000001
HI 42 Len 17
00000000: 78 2d 62 74 2f 6f 77 6e  65 72 53 74 61 74 75 73 |x-bt/own erStatus|
00000010: 00                                               |.                |
HI 4c Len 3
T 21 L 1
00000000: 00                                               |.                |
HI 49 Len 1
00000000: 30                                               |0                |
uart:~$ map mse mas set_owner_status noerror
uart:~$
```

Get Owner Status

RFCOMM Transport MCE

```
uart:~$ map mce mas get_owner_status
MCE MAS 0x20005a00 get_owner_status rsp, rsp_code Success, data len 18
HI 4c Len 12
T 1c L 1
00000000: 00                                               |.                |
T 1d L 7
00000000: 55 6e 6b 6e 6f 77 6e                             |Unknown          |
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_owner_status req, final true, data len 25
HI cb Len 4
Conn ID: 00000001
HI 42 Len 17
00000000: 78 2d 62 74 2f 6f 77 6e  65 72 53 74 61 74 75 73 |x-bt/own erStatus|
00000010: 00                                               |.                 |
uart:~$ map mse mas get_owner_status noerror
Adding default application parameters:
  Presence:  0x00
  Presence text: Unknown
uart:~$
```

L2CAP Transport with SRM MCE

```
uart:~$ map mce mas get_owner_status
MCE MAS 0x20005a00 get_owner_status rsp, rsp_code Success, data len 20
HI 97 Len 1
OBEX SRM: 01
HI 4c Len 12
T 1c L 1
00000000: 00                                               |.                 |
T 1d L 7
00000000: 55 6e 6b 6e 6f 77 6e                             |Unknown           |
HI 49 Len 0
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_owner_status req, final true, data len 27
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 17
00000000: 78 2d 62 74 2f 6f 77 6e  65 72 53 74 61 74 75 73 |x-bt/own erStatus|
00000010: 00                                               |.                 |
uart:~$ map mse mas get_owner_status noerror
Adding default application parameters:
  Presence:  0x00
  Presence text: Unknown
uart:~$
```

L2CAP Transport with SRM and SRMP MCE

```
uart:~$ map mce mas get_owner_status srmp
MCE MAS 0x20005a00 get_owner_status rsp, rsp_code Success, data len 20
HI 97 Len 1
OBEX SRM: 01
HI 4c Len 12
T 1c L 1
00000000: 00                                               |.                 |
T 1d L 7
00000000: 55 6e 6b 6e 6f 77 6e                             |Unknown           |
HI 49 Len 0
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_owner_status req, final true, data len 29
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
HI 42 Len 17
00000000: 78 2d 62 74 2f 6f 77 6e  65 72 53 74 61 74 75 73 |x-bt/own erStatus|
00000010: 00                                               |.                 |
uart:~$ map mse mas get_owner_status noerror
Adding default application parameters:
  Presence:  0x00
  Presence text: Unknown
uart:~$
```

Get Conversation Listing

RFCOMM Transport MCE

```
uart:~$ map mce mas get_convo_listing
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Continue, data len 252
HI 4c Len 72
T 12 L 2
00000000: 00 02                                            |..                |
T 1a L 32
00000000: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
00000010: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
T 1b L 32
00000000: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
00000010: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
HI 48 Len 174
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mas get_convo_listing
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Success, data len 117
HI 49 Len 114
00000000: 22 20 6e 61 6d 65 3d 22  42 6f 62 22 20 70 72 65 |" name=" Bob" pre|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_convo_listing req, final true, data len 31
HI cb Len 4
Conn ID: 00000001
HI 42 Len 23
00000000: 78 2d 62 74 2f 4d 41 50  2d 63 6f 6e 76 6f 2d 6c |x-bt/MAP -convo-l|
00000010: 69 73 74 69 6e 67 00                             |isting.           |
uart:~$ map mse mas get_convo_listing noerror
Adding default application parameters:
  Listing size: 2
  Database ID: 00000000000000010000000000000001
  Conversation version counter: 00000000000000010000000000000001
MSE MAS 0x20005c00 get_convo_listing req, final true, data len 0
```

(continues on next page)

(continued from previous page)

```
uart:~$ map mse mas get_convo_listing noerror
uart:~$
```

L2CAP Transport with SRM MCE

```
uart:~$ map mce mas get_convo_listing
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 4c Len 72
T 12 L 2
00000000: 00 02                                           |..                |
T 1a L 32
00000000: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
00000010: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
T 1b L 32
00000000: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
00000010: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
HI 48 Len 172
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Success, data len 119
HI 49 Len 116
00000000: 32 31 22 20 6e 61 6d 65  3d 22 42 6f 62 22 20 70 |21" name ="Bob" p|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_convo_listing req, final true, data len 33
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 23
00000000: 78 2d 62 74 2f 4d 41 50  2d 63 6f 6e 76 6f 2d 6c |x-bt/MAP -convo-l|
00000010: 69 73 74 69 6e 67 00                            |isting.           |
uart:~$ map mse mas get_convo_listing noerror
Adding default application parameters:
  Listing size: 2
  Database ID: 00000000000000010000000000000001
  Conversation version counter: 00000000000000010000000000000001
uart:~$ map mse mas get_convo_listing noerror
uart:~$
```

L2CAP Transport with SRM and SRMP MCE

```
uart:~$ map mce mas get_convo_listing srmp
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Continue, data len 252
HI 97 Len 1
OBEX SRM: 01
HI 4c Len 72
T 12 L 2
00000000: 00 02                                           |..                |
```

(continues on next page)

(continued from previous page)

```
T 1a L 32
00000000: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
00000010: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
T 1b L 32
00000000: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
00000010: 30 30 30 30 30 30 30 30  30 30 30 30 30 30 30 31 |00000000 00000001|
HI 48 Len 172
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mas get_convo_listing
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Continue, data len 252
HI 48 Len 249
00000000: 79 65 73 22 20 76 65 72  73 69 6f 6e 5f 63 6f 75 |yes" ver sion_cou|
...
MCE MAS 0x20005a00 get_convo_listing rsp, rsp_code Success, data len 119
HI 49 Len 116
00000000: 32 31 22 20 6e 61 6d 65  3d 22 42 6f 62 22 20 70 |21" name ="Bob" p|
...
uart:~$
```

MSE

```
MSE MAS 0x20005c00 get_convo_listing req, final true, data len 35
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
HI 42 Len 23
00000000: 78 2d 62 74 2f 4d 41 50  2d 63 6f 6e 76 6f 2d 6c |x-bt/MAP -convo-l|
00000010: 69 73 74 69 6e 67 00                             |isting.          |
uart:~$ map mse mas get_convo_listing noerror
Adding default application parameters:
  Listing size: 2
  Database ID: 00000000000000010000000000000001
  Conversation version counter: 00000000000000010000000000000001
MSE MAS 0x20005c00 get_convo_listing req, final true, data len 0
uart:~$ map mse mas get_convo_listing noerror
uart:~$ map mse mas get_convo_listing noerror
uart:~$
```

Set Notification Filter MCE

```
uart:~$ map mce mas set_ntf_filter
Adding default application parameters:
  Notification filter mask: 0x00007fff
MCE MAS 0x20005a00 set_ntf_filter rsp, rsp_code Success
uart:~$
```

MSE

```
MSE MAS 0x20005c00 set_ntf_filter req, final true, data len 50
HI cb Len 4
Conn ID: 00000001
```

(continues on next page)

(continued from previous page)

```
HI 42 Len 29
00000000: 78 2d 62 74 2f 4d 41 50  2d 6e 6f 74 69 66 69 63 |x-bt/MAP -notific|
00000010: 61 74 69 6f 6e 2d 66 69  6c 74 65 72 00          |ation-fi lter.   |
HI 4c Len 6
T 25 L 4
00000000: 00 00 7f ff                                      |....             |
HI 49 Len 1
00000000: 30                                               |0                |
uart:~$ map mse mas set_ntf_filter noerror
uart:~$
```

Send Event

RFCOMM Transport MSE

```
uart:~$ map mse mns send_event
Adding default application parameters:
  Instance ID: 0
MSE MNS 0x20005d00 send_event rsp, rsp_code Continue, data len 0
uart:~$ map mse mns send_event
MSE MNS 0x20005d00 send_event rsp, rsp_code Success, data len 0
uart:~$
```

MCE

```
MCE MNS 0x20005b00 send_event req, final false, data len 252
HI cb Len 4
Conn ID: 00000001
HI 42 Len 22
00000000: 78 2d 62 74 2f 4d 41 50  2d 65 76 65 6e 74 2d 72 |x-bt/MAP -event-r|
00000010: 65 70 6f 72 74 00                                |eport.           |
HI 4c Len 3
T 0f L 1
00000000: 00                                               |.                |
HI 48 Len 213
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mns send_event noerror
MCE MNS 0x20005b00 send_event req, final true, data len 8
HI 49 Len 5
00000000: 6f 72 74 3e 00                                   |ort>.            |
uart:~$ map mce mns send_event noerror
uart:~$
```

L2CAP Transport with SRM MSE

```
uart:~$ map mse mns send_event
Adding default application parameters:
  Instance ID: 0
MSE MNS 0x20005d00 send_event rsp, rsp_code Continue, data len 2
HI 97 Len 1
OBEX SRM: 01
uart:~$ map mse mns send_event
```

(continues on next page)

(continued from previous page)

```
MSE MNS 0x20005d00 send_event rsp, rsp_code Success, data len 0
uart:~$
```

MCE

```
MCE MNS 0x20005b00 send_event req, final false, data len 252
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 22
00000000: 78 2d 62 74 2f 4d 41 50  2d 65 76 65 6e 74 2d 72 |x-bt/MAP -event-r|
00000010: 65 70 6f 72 74 00                                |eport.           |
HI 4c Len 3
T 0f L 1
00000000: 00                                               |.                |
HI 48 Len 211
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mns send_event noerror
MCE MNS 0x20005b00 send_event req, final true, data len 10
HI 49 Len 7
00000000: 65 70 6f 72 74 3e 00                             |eport>.          |
uart:~$ map mce mns send_event noerror
uart:~$
```

L2CAP Transport with SRM and SRMP

MSE

```
uart:~$ map mse mns send_event
Adding default application parameters:
  Instance ID: 0
MSE MNS 0x20005d00 send_event rsp, rsp_code Continue, data len 4
HI 97 Len 1
OBEX SRM: 01
HI 98 Len 1
OBEX SRMP: 01
uart:~$ map mse mns send_event
MSE MNS 0x20005d00 send_event rsp, rsp_code Success, data len 0
uart:~$
```

MCE

```
MCE MNS 0x20005b00 send_event req, final false, data len 252
HI cb Len 4
Conn ID: 00000001
HI 97 Len 1
OBEX SRM: 01
HI 42 Len 22
00000000: 78 2d 62 74 2f 4d 41 50  2d 65 76 65 6e 74 2d 72 |x-bt/MAP -event-r|
00000010: 65 70 6f 72 74 00                                |eport.           |
HI 4c Len 3
T 0f L 1
00000000: 00                                               |.                |
HI 48 Len 211
00000000: 3c 3f 78 6d 6c 20 76 65  72 73 69 6f 6e 3d 27 31 |<?xml ve rsion='1|
```

(continues on next page)

(continued from previous page)

```
00000010: 2e 30 27 20 65 6e 63 6f  64 69 6e 67 3d 27 75 74 |.0' enco ding='ut|
...
uart:~$ map mce mns send_event noerror srmp
MCE MNS 0x20005b00 send_event req, final true, data len 10
HI 49 Len 7
00000000: 65 70 6f 72 74 3e 00                             |eport>.          |
uart:~$ map mce mns send_event noerror
uart:~$
```

Application Parameters The MAP shell provides commands to add application parameters to requests. These parameters must be added before sending the corresponding operation command.

Adding Application Parameters Application parameters can be added using the `map add_header app_param` commands:

```
uart:~$ map add_header app_param max_list_count 10
Added MaxListCount: 10
uart:~$ map add_header app_param list_start_offset 0
Added ListStartOffset: 0
uart:~$ map add_header app_param charset utf8
Added Charset: utf8 (1)
uart:~$
```

Listing Application Parameters To view all added application parameters:

```
uart:~$ map add_header app_param list
Added application parameters (3):
  [0] Tag=0x01, Len=2
00000000: 00 0a                                            |..               |
  [1] Tag=0x02, Len=2
00000000: 00 00                                            |..               |
  [2] Tag=0x14, Len=1
00000000: 01                                               |.                |
uart:~$
```

Clearing Application Parameters To clear all added application parameters:

```
uart:~$ map add_header app_param clear
Cleared all application parameters
uart:~$
```

Example: Get Message Listing with Filters

```
uart:~$ map add_header app_param max_list_count 10
Added MaxListCount: 10
uart:~$ map add_header app_param filter_msg_type 1f
Added FilterMessageType: 0x1f
  EMAIL=Y, SMS_GSM=Y, SMS_CDMA=Y, MMS=Y, IM=Y
uart:~$ map add_header app_param filter_read_status unread
Added FilterReadStatus: unread (1)
uart:~$ map mce mas get_msg_listing
MCE MAS 0x20005a00 get_msg_listing rsp, rsp_code Continue, data len 252
```

(continues on next page)

(continued from previous page)

```
...
uart:~$ map add_header app_param clear
Cleared all application parameters
uart:~$
```

Additional Information

RFCOMM vs L2CAP Transport

RFCOMM Transport For GET or PUT operations, the same shell command must be called multiple times until the complete message is delivered.

L2CAP Transport with SRM When SRM (Single Response Mode) is enabled, GET requests do not need to be repeated, and the server sends all response packets. For PUT operations, the client sends all request packets without waiting for intermediate responses, and the server sends a single response upon receiving the final PUT request.

L2CAP Transport with SRM and SRMP When both SRM and SRMP (Single Response Mode Parameter) are enabled, GET requests and PUT responses need to continue sending until SRMP is disabled.

SRM (Single Response Mode) SRM is automatically enabled on L2CAP transport (GOEP v2.0).

GET Operations When SRM is enabled for GET operations:

- The client sends a single GET request.
- The server sends multiple response packets without waiting for additional requests.
- The `srmp` parameter can be used to control the flow (wait for next request).

PUT Operations When SRM is enabled for PUT operations:

- The client sends multiple PUT requests without waiting for additional responses.
- The server sends a single PUT response when receiving the final PUT request.
- The `srmp` parameter can be used to control the flow (wait for next response).

Multiple Instance Support The MAP shell supports multiple MAS instances:

- MCE can connect to multiple MSE MAS instances simultaneously
- MSE can register multiple MAS instances with different instance IDs
- Use the `select` command to switch between instances

```
uart:~$ map mce mas select 0
Selected MCE MAS instance_id 0 as default
uart:~$
```

Bluetooth: Classic: PBAP Shell This document describes how to run the Bluetooth Classic PBAP (Phonebook Access Profile) functionality. The `pbap` command exposes the Bluetooth Classic PBAP Shell commands.

There are two sub-commands, `pbap pce` and `pbap pse`.

The `pbap pce` is for Phonebook Client Equipment (PCE) functionality, and the `pbap pse` is for Phonebook Server Equipment (PSE) functionality.

Commands All commands can only be used after the ACL connection has been established except `pbap pce sdp_reg` and `pbap pse rfcomm_register` and `pbap pse l2cap_register`.

The `pbap` commands:

```
uart:~$ pbap
pbap - Bluetooth pbap shell commands
Subcommands:
  alloc_buf                       : Alloc tx buffer
  release_buf                     : Free allocated tx buffer
  pce                             : Client sets
  pse                             : Server sets
  add_header_auth_challenge       : <password>
  add_header_auth_response        : <password>
  add_ap                          : add application param
```

The `pbap pce` commands:

```
uart:~$ pbap pce
pce - Client sets
Subcommands:
  sdp_reg                         : [none]
  sdp_discover                    : [none]
  connect_rfcomm                  : <channel>
  disconnect_rfcomm               : [none]
  connect_l2cap                   : <channel>
  disconnect_l2cap                : [none]
  connect                         : <mopl>
  disconnect                      : [none]
  pull_pb                         : <name> [srmp]
  pull_vcard_listing              : <name> [srmp]
  pull_vcard_entry                : <name> [srmp]
  set_phone_book                  : <Flags> [name]
  abort                           : [none]
```

The `pbap pse` commands:

```
uart:~$ pbap pse
pse - Server sets
Subcommands:
  rfcomm_register                 : [none]
  l2cap_register                  : [none]
  register                        : [none]
  connect_rsp                     : <mopl> <rsp: unauth,success,error> [rsp_code]
  disconnect_rsp                  : <rsp: success,error> [rsp_code]
  pull_phone_book_rsp             : <rsp: noerror, error> [rsp_code] [srmp]
  pull_vcard_listing_rsp          : <rsp: noerror, error> [rsp_code] [srmp]
  pull_vcard_entry_rsp            : <rsp: noerror, error> [rsp_code] [srmp]
  set_phone_book_rsp              : <rsp: success, error> [rsp_code]
  abort_rsp                       : <rsp: success, error> [rsp_code]
```

The pbap add_ap (Application Parameters) commands:

```
uart:~$ pbap add_ap
add_ap - add application param
Subcommands:
  Order                           : <indexed/alphanumeric/phonetic>
  SearchValue                     : <text>
  SearchAttribute                 : <name/number/sound>
  MaxListCount                    : <0x0000-0xffff>
  ListStartOffset                 : <0x0000-0xffff>
  PropertySelector                : <64 bits mask :␣
↪bt_pbap_appl_param_property_mask>
  Format                          : <v2.1/v3.0>
  PhonebookSize                   : <0x0000-0xffff>
  NewMissedCalls                  : <0x00-0xff>
  PrimaryFolderVersion            : <16bytes>
  SecondaryFolderVersion          : <16bytes>
  vCardSelector                   : <64 bits mask :␣
↪bt_pbap_appl_param_property_mask>
  DatabaseIdentifier              : <16bytes>
  vCardSelectorOperator           : <or/and>
  ResetNewMissedCalls             :
  PbapSupportedFeatures           : [supported features]
```

PBAP PCE SLC The pbap pce subcommand provides functionality for PBAP PCE (Phonebook Client Equipment) in Bluetooth Classic.

1. Register PCE SDP:

```
uart:~$ pbap pce sdp_reg
```

2. Discover PSE SDP:

```
uart:~$ pbap pce sdp_discover
SDP PBAP data@0x2000d4a0 (len 54) hint 0 from remote XX:XX:XX:XX:XX:XX
PSE rfcomm channel param 0x0001
PSE l2cap psm param 0x0019
PSE feature param 0x00000019
```

3. Connect to PSE via RFCOMM:

```
uart:~$ pbap pce connect_rfcomm 0x01
PBAP PCE rfcomm transport connected on 0x20005dd8
```

4. Establish OBEX connection:

```
uart:~$ pbap alloc_buf
uart:~$ pbap pce connect 300
pbap connect result Success, mopl 256
Connection ID: 0x00000001
Processing successful connection...
Connection established successfully (no auth required)
```

5. Disconnect from PSE:

```
uart:~$ pbap pce disconnect
pbap disconnect result OK
PBAP PCE rfcomm transport disconnected
```

PBAP PSE SLC The pbap pse subcommand provides functionality for PBAP PSE (Phonebook Server Equipment) in Bluetooth Classic.

1. Register PSE RFCOMM server:

```
uart:~$ pbap pse rfcomm_register
RFCOMM server (channel 01) is registered
```

2. Register PSE L2CAP server:

```
uart:~$ pbap pse l2cap_register
L2cap server (psm 01) is registered
```

3. Register PSE:

```
uart:~$ pbap pse register
```

4. Accept PCE connection and respond:

```
uart:~$ pbap alloc_buf
uart:~$ pbap pse connect_rsp 300 success
pbap connect version 1, mopl 256
Connection established with authentication
```

5. Disconnect from PCE:

```
uart:~$ pbap alloc_buf
uart:~$ pbap pse disconnect_rsp success
pbap disconnect requested by pce
```

Pull Phonebook Pull complete phonebook from PSE:

PCE side

```
uart:~$ pbap alloc_buf
uart:~$ pbap pce pull_pb "telecom/pb"
pbap pull phonebook result Continue
Application Parameters (length: 2):
  PhonebookSize: 0x0002 (2)

=========body=========
BEGIN:VCARD
VERSION:2.1
FN;CHARSET=UTF-8:descvs
N;CHARSET=UTF-8:descvs
END:VCARD
=========body=========

please send pull cmd again
uart:~$ pbap pce pull_pb "telecom/pb"
pbap pull phonebook result Success
Application Parameters (length: 2):
  PhonebookSize: 0x0002 (2)

=========body=========
BEGIN:VCARD
VERSION:2.1
FN;CHARSET=UTF-8:descvs
N;CHARSET=UTF-8:descvs
```

(continues on next page)

(continued from previous page)

```
END:VCARD
=========body=========
```

PSE side

```
uart:~$ pbap alloc_buf
pbap_pse get pull_phone_book request
name = telecom/pb
Application Parameters (length: 0):

uart:~$ pbap alloc_buf
uart:~$ pbap pse pull_phone_book_rsp noerror
Suspend after sending a single response and await the PCE request

uart:~$ pbap alloc_buf
uart:~$ pbap pse pull_phone_book_rsp noerror
Keep sending responses continuously until rsp_code is success
```

Pull vCard Listing Pull vCard listing to discover contacts:

PCE side

```
uart:~$ pbap alloc_buf
uart:~$ pbap add_ap MaxListCount 0x0005
uart:~$ pbap add_ap ListStartOffset 0x0000
uart:~$ pbap pce pull_vcard_listing "telecom/pb"
pbap pull vcard_listing result Continue
Application Parameters (length: 6):
  MaxListCount: 0x0005 (5)
  ListStartOffset: 0x0000 (0)

=========body=========
<?xml version="1.0"?><!DOCTYPE vcard-listing SYSTEM "vcard-listing.dtd">
<vCard-listing version="1.0"><card handle="1.vcf" name="qwe"/>
<card handle="2.vcf" name="qwe"/>
=========body=========

please send pull cmd again
uart:~$ pbap pce pull_vcard_listing "telecom/pb"
pbap pull vcard_listing result Success
=========body=========
<card handle="1.vcf" name="qwe"/><card handle="2.vcf" name="qwe"/>
/<vCard-listing>
=========body=========
```

PSE side

```
uart:~$ pbap alloc_buf
pbap_pse get pull_vcard_listing request
name = telecom/pb
Application Parameters (length: 6):
  MaxListCount: 0x0005 (5)
  ListStartOffset: 0x0000 (0)

uart:~$ pbap alloc_buf
uart:~$ pbap pse pull_vcard_listing_rsp noerror
```

(continues on next page)

(continued from previous page)

```
Suspend after sending a single response and await the PCE request

uart:~$ pbap alloc_buf
uart:~$ pbap pse pull_vcard_listing_rsp noerror
```

Pull vCard Entry Pull a specific vCard entry:

PCE side

```
uart:~$ pbap alloc_buf
uart:~$ pbap add_ap Format v2.1
uart:~$ pbap pce pull_vcard_entry "telecom/pb/1.vcf"
pbap pull vcard_entry result Continue
Application Parameters (length: 1):
  Format: 0x00 (vCard 2.1)

=========body=========
BEGIN:VCARD
VERSION:2.1
FN:
N:
TEL;X-0:1155
=========body=========

uart:~$ pbap pce pull_vcard_entry "telecom/pb/1.vcf"
pbap pull vcard_entry result Success

=========body=========
X-IRMC-CALL-DATETIME;DIALED:20220913T110607
END:VCARD
=========body=========
```

PSE side

```
uart:~$ pbap alloc_buf
pbap_pse get pull_vcard_entry request
name = telecom/pb/1.vcf
Application Parameters (length: 1):
  Format: 0x00 (vCard 2.1)

uart:~$ pbap alloc_buf
uart:~$ pbap pse pull_vcard_entry_rsp noerror
Suspend after sending a single response and await the PCE request

uart:~$ pbap alloc_buf
uart:~$ pbap pse pull_vcard_entry_rsp noerror
```

Set Phone Book Change the current phonebook directory:

PCE side

```
uart:~$ pbap alloc_buf
uart:~$ pbap pce set_phone_book 0x02 "telecom"
PBAP set phonebook result OK

uart:~$ pbap alloc_buf
```

(continues on next page)

(continued from previous page)

```
uart:~$ pbap pce set_phone_book 0x02 "pb"
PBAP set phonebook result OK

uart:~$ pbap alloc_buf
uart:~$ pbap pce set_phone_book 0x02
PBAP set phonebook result OK
```

PSE side

```
uart:~$ pbap alloc_buf
pbap_pse get set_phone_book request
set phonebook to children telecom folder

uart:~$ pbap alloc_buf
uart:~$ pbap pse set_phone_book_rsp success

uart:~$ pbap alloc_buf
pbap_pse get set_phone_book request
set phonebook to children pb folder

uart:~$ pbap alloc_buf
uart:~$ pbap pse set_phone_book_rsp success

uart:~$ pbap alloc_buf
pbap_pse get set_phone_book request
set phonebook to parent folder

uart:~$ pbap alloc_buf
uart:~$ pbap pse set_phone_book_rsp success
```

Abort Operation Abort an ongoing operation:

PCE side

```
uart:~$ pbap pce abort
abort success.
```

PSE side

```
uart:~$ pbap alloc_buf
pbap_pse get receive abort req

uart:~$ pbap pse abort_rsp success
```

Authentication PBAP can use OBEX authentication for secure connections. The authentication process involves challenge-response mechanism.

PCE with authentication challenge:

PCE side

```
uart:~$ pbap alloc_buf
uart:~$ pbap add_header_auth_challenge "password123"
uart:~$ pbap pce connect 255
pbap connect result Unauthorized, mopl 255
PSE requires authentication
Action required:
```

(continues on next page)

(continued from previous page)

```
  1. Allocate new tx_buf (pbap alloc_buf)
  2. Add auth_response header (pbap add_header_auth_response <password>)
  3. Add original auth_challenge header (pbap add_header_auth_challenge <password>)
  4. Re-send connect request (pbap pce connect <mopl>)

uart:~$ pbap alloc_buf
uart:~$ pbap add_header_auth_response "password123"
uart:~$ pbap add_header_auth_challenge "password123"
uart:~$ pbap pce connect 255
pbap connect result Success, mopl 255
Connection ID: 0x00000001
Authentication verified successfully
Connection established with authentication
```

PSE side

```
uart:~$ pbap alloc_buf
uart:~$ pbap pse register

uart:~$ pbap alloc_buf
pbap connect version 1, mopl 255
PCE authentication required - add auth_response to connect_rsp tx_buf

uart:~$ pbap alloc_buf
uart:~$ pbap add_header_auth_challenge "password123"
uart:~$ pbap pse connect_rsp 255 unauth

uart:~$ pbap alloc_buf
pbap connect version 1, mopl 255
auth success
Connection established with authentication

uart:~$ pbap alloc_buf
uart:~$ pbap pse connect_rsp 255 success
```

Bluetooth: Classic: SPP Shell This document describes how to run the Bluetooth Classic SPP functionality. The spp command exposes the Bluetooth Classic SPP Shell commands.

Commands The spp commands:

```
uart:~$ spp
spp - Bluetooth SPP sh commands
Subcommands:
  register_with_channel  : <channel> [limited credit value] [hold_credit]
  register_with_uuid     : <bt-uuid16(e.g. 1101)|bt-uuid32|bt-uuid128(e.g.
                           00001101-0000-1000-8000-00805F9B34FB)> [limited
                           credit value] [hold_credit]
  connect_by_channel     : <channel> [limited credit value] [hold_credit]
  connect_by_uuid        : <bt-uuid16(e.g. 1101)|bt-uuid32|bt-uuid128(e.g.
                           00001101-0000-1000-8000-00805F9B34FB)> [limited
                           credit value] [hold_credit]
  send                   : send [length of packet(s)]
  rls                    : [overrun|parity|framing]
  disconnect             : [none]
  recv_complete          : [none]
```

Server Register Register with channel

```
uart:~$ spp register_with_channel 9
SPP: server registered (channel=9)
```

Register with UUID

```
uart:~$ spp register_with_uuid 00001101-0000-1000-8000-00805F9B34FB
SPP: server registered (uuid=00001101-0000-1000-8000-00805f9b34fb, channel=9)
```

Connect The ACL connection should be established before creating the SPP connection.

Server (incoming)

```
uart:~$ spp register_with_channel 9
SPP: server registered (channel=9)
SPP: accepted incoming connection (conn=0x20004dc8)
SPP: connected (ep=0x20000d20, channel=9)
```

Client by channel

```
uart:~$ spp connect_by_channel 9
SPP: connect started (channel=9)
SPP: connected (ep=0x20000d20, channel=9)
```

Client by UUID

```
uart:~$ spp connect_by_uuid 00001101-0000-1000-8000-00805F9B34FB
SPP: connect started (uuid=00001101-0000-1000-8000-00805f9b34fb)
SPP: connected (ep=0x20000d20, channel=9)
```

Send Data

```
uart:~$ spp send 5
SPP: tx data (len=5)
```

Disconnect

```
uart:~$ spp disconnect
SPP: disconnecting...
SPP: disconnected (ep=0x20000d20)
```

Flow Control Server

```
uart:~$ spp register_with_channel 0 1 hold_credit
SPP: server registered (channel=6)
SPP: accepted incoming connection (conn=0x20005c70)
SPP: connected (ep=0x20006508, channel=6)
SPP: rx data (ep=0x20006508, len=10)
00000000: ff ff ff ff ff ff ff ff  ff ff                  |........ ..      |
uart:~$ spp recv_complete
SPP: rx data (ep=0x20006508, len=10)
00000000: ff ff ff ff ff ff ff ff  ff ff                  |........ ..      |
uart:~$ spp recv_complete
uart:~$ spp recv_complete
SPP: no inprogress buffer
uart:~$ spp send 10
```

(continues on next page)

(continued from previous page)

```
SPP: tx data (len=10)
uart:~$ spp send 10
SPP: tx data (len=10)
```

Client

```
uart:~$ spp connect_by_channel 6 1 hold_credit
SPP: connect started (channel=6)
Security changed: A0:CD:F3:77:F3:A0 level 2
SPP: connected (ep=0x20006448, channel=6)
uart:~$ spp send 10
SPP: tx data (len=10)
uart:~$ spp send 10
SPP: tx data (len=10)
uart:~$
uart:~$ spp recv_complete
SPP: rx data (ep=0x20006448, len=10)
00000000: ff ff ff ff ff ff ff ff  ff ff                  |........ ..      |
uart:~$ spp recv_complete
SPP: rx data (ep=0x20006448, len=10)
00000000: ff ff ff ff ff ff ff ff  ff ff                  |........ ..      |
uart:~$ spp recv_complete
uart:~$ spp recv_complete
SPP: no inprogress buffer
```

Bluetooth: GAP Shell The GAP shell is the "main" shell of Bluetooth and handles connection management, scanning, advertising, and more.

Identities Identities are a Zephyr host concept, allowing a single physical device to behave like multiple logical Bluetooth devices.

The shell allows the creation of multiple identities, to a maximum that is set by the Kconfig symbol `CONFIG_BT_ID_MAX`. To create a new identity, use `bt id-create` command. You can then use it by selecting it with its ID `bt id-select <id>`. Finally, you can list all the available identities with `id-show`.

Scan for devices Start scanning by using the `bt scan on` command. Depending on the environment you're in, you may see a lot of lines printed on the shell. To stop the scan, run `bt scan off`, the scrolling should stop.

Here is an example of what you can expect:

```
uart:~$ bt scan on
Bluetooth active scan enabled
[DEVICE]: R:CB:01:1A:2D:6E:AE, AD evt type 0, RSSI -78  C:1 S:1 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: R:20:C2:EE:59:85:5B, AD evt type 3, RSSI -62  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: R:E3:72:76:87:2F:E8, AD evt type 3, RSSI -74  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: R:1E:19:25:8A:CB:84, AD evt type 3, RSSI -67  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: R:26:42:F3:D5:A0:86, AD evt type 3, RSSI -73  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: R:0C:61:D1:B9:5D:9E, AD evt type 3, RSSI -87  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
```

(continues on next page)

(continued from previous page)

```
[DEVICE]: R:20:C2:EE:59:85:5B, AD evt type 3, RSSI -66  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
[DEVICE]: R:25:3F:7A:EE:0F:55, AD evt type 3, RSSI -83  C:0 S:0 D:0 SR:0 E:0 Prim:␣
↪LE 1M, Secn: No packets, Interval: 0x0000 (0 us), SID: 0xff
uart:~$ bt scan off
Scan successfully stopped
```

As you can see, this can lead to a high number of results. To reduce that number and easily find a specific device, you can enable scan filters. There are four types of filters: by name, by RSSI, by address and by periodic advertising interval. To apply a filter, use the `bt scan-set-filter` command followed by the type of filters. You can add multiple filters by using the commands again.

For example, if you want to look only for devices with the name *test shell*:

```
uart:~$ bt scan-filter-set name "test shell"
```

Or if you want to look for devices at a very close range:

```
uart:~$ bt scan-filter-set rssi -40
RSSI cutoff set at -40 dB
```

Finally, if you want to remove all filters:

```
uart:~$ bt scan-filter-clear all
```

You can use the command `bt scan on` to create an *active* scanner, meaning that the scanner will ask the advertisers for more information by sending a *scan request* packet. Alternatively, you can create a *passive scanner* by using the `bt scan passive` command, so the scanner will not ask the advertiser for more information.

With `CONFIG_BT_SCAN_EXT_FILTER_POLICY` enabled, the `bt scan --ext-filter-policy on` command starts the scanner with the extended scanner filter policy. The Controller then also reports directed advertisements whose target address is a resolvable private address it was unable to resolve. The target address of a directed advertisement is printed on its own line. This requires a Controller that supports the Extended Scanner Filter Policies, and the command fails otherwise.

Connecting to a device To connect to a device, you need to know its address and type of address and use the `bt connect` command with the address and the type as arguments.

Here is an example:

```
uart:~$ bt connect R:52:84:F6:BD:CE:48
Connection pending
Connected: R:52:84:F6:BD:CE:48
Remote LMP version 5.3 (0x0c) subversion 0xffff manufacturer 0x05f1
LE Features: 0x000000000001412f
LE PHY updated: TX PHY LE 2M, RX PHY LE 2M
LE conn  param req: int (0x0018, 0x0028) lat 0 to 42
LE conn param updated: int 0x0028 lat 0 to 42
```

You can list the active connections of the shell using the `bt connections` command. The shell maximum number of connections is defined by `CONFIG_BT_MAX_CONN`. You can disconnect from a connection with the `bt disconnect <address: P:XX:XX:XX:XX:XX or R:XX:XX:XX:XX:XX>` command.

> **Note**
>
> If you were scanning just before, you can connect to the last scanned device by simply running the `bt connect` command.
>
> Alternatively, you can use the `bt connect-name <name>` command to automatically enable scanning with a name filter and connect to the first match.

Advertising Begin advertising by using the `bt advertise on` command. This will use the default parameters and advertise a resolvable private address with the name of the device. You can choose to use the identity address instead by running the `bt advertise on identity` command. To stop advertising use the `bt advertise off` command.

To enable more advanced features of advertising, you should create an advertiser using the `bt adv-create` command. Parameters for the advertiser can be passed either at the creation of it or by using the `bt adv-param` command. To begin advertising with this newly created advertiser, use the `bt adv-start` command, and then the `bt adv-stop` command to stop advertising.

When using the custom advertisers, you can choose if it will be connectable or scannable. This leads to four options: `conn-scan`, `conn-nscan`, `nconn-scan` and `nconn-nscan`. Those parameters are mandatory when creating an advertiser or updating its parameters.

For example, if you want to create a connectable and scannable advertiser and start it:

```
uart:~$ bt adv-create conn-scan
Created adv id: 0, adv: 0x200022f0
uart:~$ bt adv-start
Advertiser[0] 0x200022f0 set started
```

You may notice that with this, the custom advertiser does not advertise the device name; you need to add it. Continuing from the previous example:

```
uart:~$ bt adv-stop
Advertiser set stopped
uart:~$ bt adv-data dev-name
uart:~$ bt adv-start
Advertiser[0] 0x200022f0 set started
```

You should now see the name of the device in the advertising data. You can also set a custom name by using `name <custom name>` instead of `dev-name`. It is also possible to set the advertising data manually with the `bt adv-data` command. The following example shows how to set the advertiser name with it using raw advertising data:

```
uart:~$ bt adv-create conn-scan
Created adv id: 0, adv: 0x20002348
uart:~$ bt adv-data 1009426C7565746F6F74682D5368656C6C
uart:~$ bt adv-start
Advertiser[0] 0x20002348 set started
```

The data must be formatted according to the Bluetooth Core Specification (see version 5.3, vol. 3, part C, 11). In this example, the first octet is the size of the data (the data and one octet for the data type), the second one is the type of data, `0x09` is the Complete Local Name and the remaining data are the name in ASCII. So, on the other device you should see the name *Bluetooth-Shell*.

When advertising, if others devices use an *active* scanner, you may receive *scan request* packets. To visualize those packets, you can add `scan-reports` to the parameters of your advertiser.

Directed Advertising It is possible to use directed advertising on the shell if you want to reconnect to a device. The following example demonstrates how to create a directed advertiser

with the address specified right after the parameter `directed`. The `low` parameter indicates that we want to use the low duty cycle mode, and the `dir-rpa` parameter is required if the remote device is privacy-enabled and supports address resolution of the target address in directed advertisement.

```
uart:~$ bt adv-create conn-scan directed R:D7:54:03:CE:F3:B4 low dir-rpa
Created adv id: 0, adv: 0x20002348
```

After that, you can start the advertiser and then the target device will be able to reconnect.

Extended Advertising Let's now have a look at some extended advertising features. To enable extended advertising, use the `ext-adv` parameter.

```
uart:~$ bt adv-create conn-nscan ext-adv
Created adv id: 0, adv: 0x200022f0
uart:~$ bt adv-start
Advertiser[0] 0x200022f0 set started
```

This will create an extended advertiser, that is connectable and non-scannable.

Encrypted Advertising Data Zephyr has support for the Encrypted Advertising Data feature. The `bt encrypted-ad` sub-commands allow managing the advertising data of a given advertiser.

To encrypt the advertising data, key materials need to be provided, that can be done with `bt encrypted-ad set-keys <session key> <init vector>`. The session key is 16 bytes long and the initialisation vector is 8 bytes long.

You can add advertising data by using `bt encrypted-ad add-ad` and `bt encrypted-ad add-ead`. The former will take add one advertising data structure (as defined in the Core Specification), when the later will read the given data, encrypt them and then add the generated encrypted advertising data structure. It's possible to mix encrypted and non-encrypted data, when done adding advertising data, `bt encrypted-ad commit-ad` can be used to apply the change to the data to the selected advertiser. After that the advertiser can be started as described previously. It's possible to clear the advertising data by using `bt encrypted-ad clear-ad`.

On the Central side, it's possible to decrypt the received encrypted advertising data by setting the correct keys material as described earlier and then enabling the decrypting of the data with `bt encrypted-ad decrypt-scan on`.

> **Note**
>
> To see the advertising data in the scan report `bt scan-verbose-output` need to be enabled.

> **Note**
>
> It's possible to increase the length of the advertising data by increasing the value of `CONFIG_BT_CTLR_ADV_DATA_LEN_MAX` and `CONFIG_BT_CTLR_SCAN_DATA_LEN_MAX`.

Here is a simple example demonstrating the usage of EAD:

Peripheral

```
uart:~$ bt init
...
uart:~$ bt adv-create conn-nscan ext-adv
Created adv id: 0, adv: 0x81769a0
uart:~$ bt encrypted-ad set-keys 9ba22d3824efc70feb800c80294cba38 2e83f3d4d47695b6
```

(continues on next page)

(continued from previous page)

```
session key set to:
00000000: 9b a2 2d 38 24 ef c7 0f  eb 80 0c 80 29 4c ba 38 |..-8$... ....)L.8|
initialisation vector set to:
00000000: 2e 83 f3 d4 d4 76 95 b6                          |.....v..         |
uart:~$ bt encrypted-ad add-ad 06097368656C6C
uart:~$ bt encrypted-ad add-ead 03ffdead03ffbeef
uart:~$ bt encrypted-ad commit-ad
Advertising data for Advertiser[0] 0x81769a0 updated.
uart:~$ bt adv-start
Advertiser[0] 0x81769a0 set started
```

Central

```
uart:~$ bt init
...
uart:~$ bt scan-verbose-output on
uart:~$ bt encrypted-ad set-keys 9ba22d3824efc70feb800c80294cba38 2e83f3d4d47695b6
session key set to:
00000000: 9b a2 2d 38 24 ef c7 0f  eb 80 0c 80 29 4c ba 38 |..-8$... ....)L.8|
initialisation vector set to:
00000000: 2e 83 f3 d4 d4 76 95 b6                          |.....v..         |
uart:~$ bt encrypted-ad decrypt-scan on
Received encrypted advertising data will now be decrypted using provided key␣
↪materials.
uart:~$ bt scan on
Bluetooth active scan enabled
[DEVICE]: R:68:49:30:68:49:30, AD evt type 5, RSSI -59   shell C:1 S:0 D:0 SR:0␣
↪E:1 Prim: LE 1M, Secn: LE 2M, Interval: 0x0000 (0 us), SID: 0x0
        [SCAN DATA START - EXT_ADV]
        Type 0x09:    shell
        Type 0x31: Encrypted Advertising Data: 0xe2, 0x17, 0xed, 0x04, 0xe7, 0x02,␣
↪0x1d, 0xc9, 0x40, 0x07, uart:~0x18, 0x90, 0x6c, 0x4b, 0xfe, 0x34, 0xad
        [START DECRYPTED DATA]
        Type 0xff: 0xde, 0xad
        Type 0xff: 0xbe, 0xef
        [END DECRYPTED DATA]
        [SCAN DATA END]
...
```

Filter Accept List It's possible to create a list of allowed addresses that can be used to connect to those addresses automatically. Here is how to do it:

```
uart:~$ bt fal-add R:47:38:76:EA:29:36
uart:~$ bt fal-add R:66:C8:80:2A:05:73
uart:~$ bt fal-connect on
```

The shell will then connect to the first available device. In the example, if both devices are advertising at the same time, we will connect to the first address added to the list.

The Filter Accept List can also be used for scanning or advertising by using the option `fal`. For example, if we want to scan for a bunch of selected addresses, we can set up a Filter Accept List:

```
uart:~$ bt fal-add R:65:4B:9E:83:AF:73
uart:~$ bt fal-add R:73:72:82:B4:8F:B9
uart:~$ bt fal-add R:5D:85:50:1C:72:64
uart:~$ bt scan on fal
```

You should see only those three addresses reported by the scanner.

Enabling security When connected to a device, you can enable multiple levels of security, here is the list for Bluetooth LE:

- **1** No encryption and no authentication;
- **2** Encryption and no authentication;
- **3** Encryption and authentication;
- **4** Bluetooth LE Secure Connection.

To enable security, use the `bt security <level>` command. For levels requiring authentication (level 3 and above), you must first set the authentication method. To do it, you can use the `bt auth all` command. After that, when you will set the security level, you will be asked to confirm the passkey on both devices. On the shell side, do it with the command `bt auth-passkey-confirm`.

Pairing Enabling authentication requires the devices to be bondable. By default the shell is bondable. You can make the shell not bondable using `bt bondable off`. You can list all the devices you are paired with using the command `bt bonds`.

The maximum number of paired devices is set using `CONFIG_BT_MAX_PAIRED`. You can remove a paired device using `bt clear <address: P:XX:XX:XX:XX:XX:XX or R:XX:XX:XX:XX:XX:XX>` or remove all paired devices with the command `bt clear all`.

Bluetooth: GATT Shell The following examples assume that you have two devices already connected.

To perform service discovery on the client side, use the `gatt discover` command. This should print all the services that are available on the GATT server.

On the server side, you can register pre-defined test services using the `gatt register` command. When done, you should see the newly added services on the client side when running the discovery command.

You can now subscribe to those new services on the client side. Here is an example on how to subscribe to the test service:

```
uart:~$ gatt subscribe 26 25
Subscribed
```

The server can now notify the client with the command `gatt notify`.

Another option available through the GATT command is initiating the MTU exchange. To do it, use the `gatt exchange-mtu` command. To update the shell maximum MTU, you need to update Kconfig symbols in the configuration file of the shell. For more details, see bluetooth_mtu_update.

Bluetooth: Isochronous Channels Shell

Commands

```
iso --help
iso - Bluetooth ISO shell commands
Subcommands:
   cig_create  :[dir=tx,rx,txrx] [interval] [packing] [framing] [latency] [sdu]
               [phy] [rtn]
   cig_term    :Terminate the CIG
   connect     :Connect ISO Channel
   listen      :<dir=tx,rx,txrx> [security level]
```

(continues on next page)

(continued from previous page)

```
  send        :Send to ISO Channel [count]
  disconnect  :Disconnect ISO Channel
  create-big  :Create a BIG as a broadcaster [enc <broadcast code>]
  broadcast   :Broadcast on ISO channels
  sync-big    :Synchronize to a BIG as a receiver <BIS bitfield> [mse] [timeout]
              [enc <broadcast code>]
  term-big    :Terminate a BIG
```

Unicast examples

1. [Central] Create CIG:

Requires to be connected:

```
uart:~$ iso cig_create
CIG created
```

2. [Peripheral] Listen to ISO connections

```
uart:~$ iso listen txrx
```

3. [Central] Connect ISO channel:

```
uart:~$ iso connect
ISO Connect pending...
ISO Channel 0x20000f88 connected
```

4. Send data:

```
uart:~$ iso send
send: 40 bytes of data
ISO sending...
```

5. Disconnect ISO channel:

```
uart:~$ iso disconnect
ISO Disconnect pending...
ISO Channel 0x20000f88 disconnected with reason 0x16
```

Broadcast examples

Setting up broadcaster

```
uart:~$ bt init
Bluetooth initialized
uart:~$ bt adv-create nconn-nscan ext-adv
Created adv id: 0, adv: 0x200025d0
uart:~$ bt per-adv-param
uart:~$ iso create-big
BIG created
ISO Channel 0x200008c0 connected
uart:~$
uart:~$ bt adv-start
Advertiser[0] 0x200025d0 set started
uart:~$
uart:~$ bt per-adv on
Periodic advertising started
```

(continues on next page)

(continued from previous page)

```
uart:~$
uart:~$ iso broadcast
send: 247 bytes of data with PSN 4350
ISO broadcasting...
```

If encrypted broadcast is required, then a broadcast code can be provided as

```
uart:~$ iso create-big enc 00112233445566778899aabbccddffff
BIG created
```

Setting up sync receiver

```
uart:~$ bt init
Bluetooth initialized
uart:~$ bt scan on
[DEVICE]: R:42:0F:7A:40:AE:21, AD evt type 5, RSSI -27  C:0 S:0 D:0 SR:0 E:1 Prim:␣
↪LE 1M, Secn: LE 2M, Interval: 0x0780 (2400000 us), SID: 0x0
uart:~$ bt per-adv-sync-create R:42:0F:7A:40:AE:21 0
Per adv sync pending
PER_ADV_SYNC[0]: [DEVICE]: R:42:0F:7A:40:AE:21 synced, Interval 0x0780 (2400000␣
↪us), PHY LE 2M, SD 0x0000, PAST peer not present
PER_ADV_SYNC[0]: [DEVICE]: R:42:0F:7A:40:AE:21, tx_power 127, RSSI -28, CTE 0,␣
↪data length 0
BIG_INFO PER_ADV_SYNC[0]: [DEVICE]: R:42:0F:7A:40:AE:21, sid 0x00, num_bis 1, nse␣
↪0x04, interval 0x0008 (10000 us), bn 0x01, pto 0x01, irc 0x03, max_pdu 0x00f7,␣
↪sdu_interval 0x2710, max_sdu 0x00f7, phy LE 2M, framing 0x00, not encrypted
uart:~$ iso sync-big 1
BIG syncing
ISO Channel 0x200008c0 connected
```

If encrypted broadcast is required, then a broadcast code can be provided as

```
uart:~$ iso sync-big 1 enc 00112233445566778899aabbccddffff
BIG syncing
ISO Channel 0x200008c0 connected
```

Bluetooth: L2CAP Shell The `l2cap` command exposes parts of the L2CAP API. The following example shows how to register a LE PSM, connect to it from another device and send 3 packets of 14 octets each.

The example assumes that the two devices are already connected.

On device A, register the LE PSM:

```
uart:~$ l2cap register 29
L2CAP psm 41 sec_level 1 registered
```

On device B, connect to the registered LE PSM and send data:

```
uart:~$ l2cap connect 29
Chan sec: 1
L2CAP connection pending
Channel 0x20000210 connected
Channel 0x20000210 status 1
uart:~$ l2cap send 3 14
Rem 2
Rem 1
```

(continues on next page)

(continued from previous page)

```
Rem 0
Outgoing data channel 0x20000210 transmitted
Outgoing data channel 0x20000210 transmitted
Outgoing data channel 0x20000210 transmitted
```

On device A, you should have received the data:

```
Incoming conn 0x20002398
Channel 0x20000210 status 1
Channel 0x20000210 connected
Channel 0x20000210 requires buffer
Incoming data channel 0x20000210 len 14
00000000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff       |........ ......  |
Channel 0x20000210 requires buffer
Incoming data channel 0x20000210 len 14
00000000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff       |........ ......  |
Channel 0x20000210 requires buffer
Incoming data channel 0x20000210 len 14
00000000: ff ff ff ff ff ff ff ff  ff ff ff ff ff ff       |........ ......  |
```

Bluetooth Shell Setup and Usage First you need to build and flash your board with the Bluetooth shell. For how to do that, see the *Getting Started Guide*. The Bluetooth shell itself is located in tests/bluetooth/shell/.

When it's done, connect to the CLI using your favorite serial terminal application. You should see the following prompt:

```
uart:~$
```

For more details on general usage of the shell, see *Shell*.

The first step is enabling Bluetooth. To do so, use the `bt init` command. The following message is printed to confirm Bluetooth has been initialized.

```
uart:~$ bt init
Bluetooth initialized
Settings Loaded
[00:02:26.771,148] <inf> fs_nvs: nvs_mount: 8 Sectors of 4096 bytes
[00:02:26.771,148] <inf> fs_nvs: nvs_mount: alloc wra: 0, fe8
[00:02:26.771,179] <inf> fs_nvs: nvs_mount: data wra: 0, 0
[00:02:26.777,984] <inf> bt_hci_core: hci_vs_init: HW Platform: Nordic␣
↪Semiconductor (0x0002)
[00:02:26.778,015] <inf> bt_hci_core: hci_vs_init: HW Variant: nRF52x (0x0002)
[00:02:26.778,045] <inf> bt_hci_core: hci_vs_init: Controller: Zephyr Bluetooth␣
↪Controller (0x00) manufacturer 0x05f1 Version 3.2 Build 99
[00:02:26.778,656] <inf> bt_hci_core: bt_init: No ID address. App must call␣
↪settings_load()
[00:02:26.794,738] <inf> bt_hci_core: bt_dev_show_info: Identity:␣
↪R:EB:BF:36:26:42:09
[00:02:26.794,769] <inf> bt_hci_core: bt_dev_show_info: HCI: version 5.3 (0x0c)␣
↪revision 0x0000, manufacturer 0x05f1
[00:02:26.794,799] <inf> bt_hci_core: bt_dev_show_info: LMP: version 5.3 (0x0c)␣
↪subver 0xffff
```

Logging You can configure the logging level per module at runtime. This depends on the maximum logging level that is compiled in. To configure, use the `log` command. Here are some examples:

- List the available modules and their current logging level

```
uart:~$ log status
```

- Disable logging for *bt_hci_core*

```
uart:~$ log disable bt_hci_core
```

- Enable error logs for *bt_att* and *bt_smp*

```
uart:~$ log enable err bt_att bt_smp
```

- Disable logging for all modules

```
uart:~$ log disable
```

- Enable warning logs for all modules

```
uart:~$ log enable wrn
```

4.1.3 USB

Legacy USB device support

USB device support (deprecated)

- *Overview*
- *Supported USB classes*
  - *Audio*
  - *Bluetooth HCI USB transport layer*
  - *CDC ACM*
    - *Console over CDC ACM UART*
    - *CDC ACM UART as backend*
    - *POSIX default tty ECHO mitigation*
  - *DFU*
  - *USB Human Interface Devices (HID) support*
  - *Mass Storage Class*
  - *Networking*
- *Binary Device Object Store (BOS) support*
- *Interface number and endpoint address assignment*
- *Testing over USBIP in native_sim*
- *USB Vendor and Product identifiers*

Overview The USB device stack is a hardware independent interface between USB device controller driver and USB device class drivers or customer applications. It is a port of the LPCUSB device stack and has been modified and expanded over time. It provides the following functionalities:

- Uses the *USB device controller driver API (deprecated)* provided by the device controller drivers to interact with the USB device controller.
- Responds to standard device requests and returns standard descriptors, essentially handling 'Chapter 9' processing, specifically the standard device requests in table 9-3 from the universal serial bus specification revision 2.0.
- Provides a programming interface to be used by USB device classes or customer applications. The APIs is described in include/zephyr/usb/usb_device.h

> **Note**
>
> All APIs listed in *USB device support APIs (deprecated)* and the functions that depend on them are deprecated and will be remove in v4.5.0. Please use the new USB device support represented by the APIs in *USB device support APIs*.

Supported USB classes

Audio There is an experimental implementation of the Audio class. It follows specification version 1.00 (`bcdADC 0x0100`) and supports synchronous synchronisation type only. See usb-audio-headphones-microphone and usb-audio-headset samples for reference.

Bluetooth HCI USB transport layer Bluetooth HCI USB transport layer implementation uses *HCI RAW channel* to expose HCI interface to the host. It is not fully in line with the description in the Bluetooth specification and consists only of an interface with the endpoint configuration:

- HCI commands through control endpoint (host-to-device only)
- HCI events through interrupt IN endpoint
- ACL data through one bulk IN and one bulk OUT endpoints

A second interface for the voice channels has not been implemented as there is no support for this type in *Bluetooth*. It is not a big problem under Linux if HCI USB transport layer is the only interface that appears in the configuration, the btusb driver would not try to claim a second (isochronous) interface. The consequence is that if HCI USB is used in a composite configuration and is the first interface, then the Linux btusb driver will claim both the first and the next interface, preventing other composite functions from working. Because of this problem, HCI USB should not be used in a composite configuration. This problem is fixed in the implementation for new USB support.

See bluetooth_hci_usb sample for reference.

CDC ACM The CDC ACM class is used as backend for different subsystems in Zephyr. However, its configuration may not be easy for the inexperienced user. Below is a description of the different use cases and some pitfalls.

The interface for CDC ACM user is *Universal Asynchronous Receiver-Transmitter (UART)* driver API. But there are two important differences in behavior to a real UART controller:

- Data transfer is only possible after the USB device stack has been initialized and started, until then any data is discarded
- If device is connected to the host, it still needs an application on the host side which requests the data
- The CDC ACM poll out implementation follows the API and blocks when the TX ring buffer is full only if the hw-flow-control property is enabled and called from a non-ISR context.

The devicetree compatible property for CDC ACM UART is `zephyr,cdc-acm-uart`. CDC ACM support is automatically selected when USB device support is enabled and a compatible node in the

devicetree sources is present. If necessary, CDC ACM support can be explicitly disabled by `CONFIG_USB_CDC_ACM`. About four CDC ACM UART instances can be defined and used, limited by the maximum number of supported endpoints on the controller.

CDC ACM UART node is supposed to be child of a USB device controller node. Since the designation of the controller nodes varies from vendor to vendor, and our samples and application should be as generic as possible, the default USB device controller is usually assigned an `zephyr_udc0` node label. Often, CDC ACM UART is described in a devicetree overlay file and looks like this:

```
&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

Sample usb-cdc-acm has similar overlay files. And since no special properties are present, it may seem overkill to use devicetree to describe CDC ACM UART. The motivation behind using devicetree is the easy interchangeability of a real UART controller and CDC ACM UART in applications.

Console over CDC ACM UART With the CDC ACM UART node from above and `zephyr,console` property of the chosen node, we can describe that CDC ACM UART is to be used with the console. A similar overlay file is used by the usb-cdc-acm-console sample.

```
/ {
        chosen {
                zephyr,console = &cdc_acm_uart0;
        };
};

&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

Before the application uses the console, it is recommended to wait for the DTR signal:

```
const struct device *const dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_console));
uint32_t dtr = 0;

if (usb_enable(NULL)) {
        return;
}

while (!dtr) {
        uart_line_ctrl_get(dev, UART_LINE_CTRL_DTR, &dtr);
        k_sleep(K_MSEC(100));
}

printk("nuqneH\n");
```

CDC ACM UART as backend As for the console sample, it is possible to configure CDC ACM UART as backend for other subsystems by setting *Chosen nodes* properties.

List of few Zephyr specific chosen properties which can be used to select CDC ACM UART as backend for a subsystem or application:

- `zephyr,bt-c2h-uart` used in Bluetooth, for example see bluetooth_hci_uart
- `zephyr,ot-uart` used in OpenThread, for example see openthread-coprocessor
- `zephyr,shell-uart` used by shell for serial backend, for example see samples/subsys/shell/shell_module
- `zephyr,uart-mcumgr` used by smp-svr sample

POSIX default tty ECHO mitigation POSIX systems, like Linux, default to enabling ECHO on tty devices. Host side application can disable ECHO by calling `open()` on the tty device and issuing `ioctl()` (preferably via `tcsetattr()`) to disable echo if it is not desired. Unfortunately, there is an inherent race between the `open()` and `ioctl()` where the ECHO is enabled and any characters received (even if host application does not call `read()`) will be echoed back. This issue is especially visible when the CDC ACM port is used without any real UART on the other side because there is no arbitrary delay due to baud rate.

To mitigate the issue, Zephyr CDC ACM implementation arms IN endpoint with ZLP after device is configured. When the host reads the ZLP, which is pretty much the best indication that host application has opened the tty device, Zephyr will force `CONFIG_CDC_ACM_TX_DELAY_MS` millisecond delay before real payload is sent. This should allow sufficient time for first, and only first, application that opens the tty device to disable ECHO if ECHO is not desired. If ECHO is not desired at all from CDC ACM device it is best to set up udev rule to disable ECHO as soon as device is connected.

ECHO is particurarly unwanted when CDC ACM instance is used for Zephyr shell, because the control characters to set color sent back to shell are interpreted as (invalid) command and user will see garbage as a result. While minicom does disable ECHO by default, on exit with reset it will restore the termios settings to whatever was set on entry. Therefore, if minicom is the first application to open the tty device, the exit with reset will enable ECHO back and thus set up a problem for the next application (which cannot be mitigated at Zephyr side). To prevent the issue it is recommended either to leave minicom without reset or to disable ECHO before minicom is started.

DFU USB DFU class implementation is tightly coupled to *Device Firmware Upgrade* and *MCUBoot API*. This means that the target platform must support the *Flash Image* API.

See legacy-usb-dfu sample for reference.

USB Human Interface Devices (HID) support HID support abuses *Device Driver Model* simply to allow applications to use the `device_get_binding()`. Note that there is no HID device API as such, instead the interface is provided by `hid_ops`. The default instance name is `HID_n`, where n can be {0, 1, 2, ...} depending on the `CONFIG_USB_HID_DEVICE_COUNT`.

Each HID instance requires a HID report descriptor. The interface to the core and the report descriptor must be registered using `usb_hid_register_device()`.

As the USB HID specification is not only used by the USB subsystem, the USB HID API reference is split into two parts, *Human Interface Devices (HID)* and *USB HID Class API (deprecated)*. HID helper macros from *Human Interface Devices (HID)* should be used to compose a HID report descriptor. Macro names correspond to those used in the USB HID specification.

For the HID class interface, an IN interrupt endpoint is required for each instance, an OUT interrupt endpoint is optional. Thus, the minimum implementation requirement for `hid_ops` is to provide `int_in_ready` callback.

```
#define REPORT_ID               1
static bool configured;
static const struct device *hdev;

static void int_in_ready_cb(const struct device *dev)
```

(continues on next page)

(continued from previous page)

```
{
        static uint8_t report[2] = {REPORT_ID, 0};

        if (hid_int_ep_write(hdev, report, sizeof(report), NULL)) {
                LOG_ERR("Failed to submit report");
        } else {
                report[1]++;
        }
}

static void status_cb(enum usb_dc_status_code status, const uint8_t *param)
{
        if (status == USB_DC_RESET) {
                configured = false;
        }

        if (status == USB_DC_CONFIGURED && !configured) {
                int_in_ready_cb(hdev);
                configured = true;
        }
}

static const uint8_t hid_report_desc[] = {
        HID_USAGE_PAGE(HID_USAGE_GEN_DESKTOP),
        HID_USAGE(HID_USAGE_GEN_DESKTOP_UNDEFINED),
        HID_COLLECTION(HID_COLLECTION_APPLICATION),
        HID_LOGICAL_MIN8(0x00),
        HID_LOGICAL_MAX16(0xFF, 0x00),
        HID_REPORT_ID(REPORT_ID),
        HID_REPORT_SIZE(8),
        HID_REPORT_COUNT(1),
        HID_USAGE(HID_USAGE_GEN_DESKTOP_UNDEFINED),
        HID_INPUT(0x02),
        HID_END_COLLECTION,
};

static const struct hid_ops my_ops = {
        .int_in_ready = int_in_ready_cb,
};

int main(void)
{
        int ret;

        hdev = device_get_binding("HID_0");
        if (hdev == NULL) {
                return -ENODEV;
        }

        usb_hid_register_device(hdev, hid_report_desc, sizeof(hid_report_desc),
                                &my_ops);

        ret = usb_hid_init(hdev);
        if (ret) {
                return ret;
        }
```

(continues on next page)

(continued from previous page)

```
        return usb_enable(status_cb);
}
```

If the application wishes to receive output reports via the OUT interrupt endpoint, it must enable `CONFIG_ENABLE_HID_INT_OUT_EP` and provide `int_out_ready` callback. The disadvantage of this is that Kconfig options such as `CONFIG_ENABLE_HID_INT_OUT_EP` or `CONFIG_HID_INTERRUPT_EP_MPS` apply to all instances. This design issue will be fixed in the HID class implementation for the new USB support.

See usb-hid-mouse sample for reference.

Mass Storage Class MSC follows Bulk-Only Transport specification and uses *Disk Access* to access and expose a RAM disk, emulated block device on a flash partition, or SD Card to the host. Only one disk instance can be exported at a time.

The disc to be used by the implementation is set by the `CONFIG_MASS_STORAGE_DISK_NAME` and should be the same as the name used by the disc access driver that the application wants to expose to the host. Flash, RAM, and SDMMC/MMC disk drivers use node property `disk-name` to set the disk name.

For the emulated block device on a flash partition, the flash partition and flash disk to be used must be described in the devicetree. If a storage partition is already described at the board level, application devicetree overlay must also delete `storage_partition` node first. `CONFIG_MASS_STORAGE_DISK_NAME` should be the same as `disk-name` property.

```
/delete-node/ &storage_partition;

&mx25r64 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                storage_partition: partition@0 {
                        label = "storage";
                        reg = <0x00000000 0x00020000>;
                };
        };
};

/ {
        msc_disk0 {
                compatible = "zephyr,flash-disk";
                partition = <&storage_partition>;
                disk-name = "NAND";
                cache-size = <4096>;
        };
};
```

The `disk-property` "NAND" may be confusing, but it is simply how some file systems identifies the disc. Therefore, if the application also accesses the file system on the exposed disc, default names should be used, see usb-mass sample for reference.

Networking There are three implementations that work in a similar way, providing a virtual Ethernet connection between the remote (USB host) and Zephyr network support.

- CDC ECM class, enabled with `CONFIG_USB_DEVICE_NETWORK_ECM`

- CDC EEM class, enabled with `CONFIG_USB_DEVICE_NETWORK_EEM`
- RNDIS support, enabled with `CONFIG_USB_DEVICE_NETWORK_RNDIS`

See legacy-netusb sample for reference.

Applications using RNDIS support should enable `CONFIG_USB_DEVICE_OS_DESC` for a better user experience on a host running Microsoft Windows OS.

Binary Device Object Store (BOS) support BOS handling can be enabled with Kconfig option `CONFIG_USB_DEVICE_BOS`. This option also has the effect of changing device descriptor `bcdUSB` to `0210`. The application should register descriptors such as Capability Descriptor using `usb_bos_register_cap()`. Registered descriptors are added to the root BOS descriptor and handled by the stack.

See legacy-webusb sample for reference.

Interface number and endpoint address assignment In USB terminology, a `function` is a device that provides a capability to the host, such as a HID class device that implements a keyboard. A function contains a collection of `interfaces`; at least one interface is required. An interface may contain device `endpoints`; for example, at least one input endpoint is required to implement a HID class device, and no endpoints are required to implement a USB DFU class. A USB device that combines functions is a multifunction USB device, for example, a combination of a HID class device and a CDC ACM device.

With Zephyr RTOS USB support, various combinations are possible with built-in USB classes/functions or custom user implementations. The limitation is the number of available device endpoints. Each device endpoint is uniquely addressable. The endpoint address is a combination of endpoint direction and endpoint number, a four-bit value. Endpoint number zero is used for the default control method to initialize and configure a USB device. By specification, a maximum of `15 IN` and `15 OUT` device endpoints are also available for use in functions. The actual number depends on the device controller used. Not all controllers support the maximum number of endpoints and all endpoint types. For example, a device controller might support one IN and one OUT isochronous endpoint, but only for endpoint number 8, resulting in endpoint addresses 0x88 and 0x08. Also, one controller may be able to have IN/OUT endpoints on the same endpoint number, interrupt IN endpoint 0x81 and bulk OUT endpoint 0x01, while the other may only be able to handle one endpoint per endpoint number. Information about the number of interfaces, interface associations, endpoint types, and addresses is provided to the host by the interface, interface specific, and endpoint descriptors.

Host driver for specific function, uses interface and endpoint descriptor to obtain endpoint addresses, types, and other properties. This allows function host drivers to be generic, for example, a multi-function device consisting of one or more CDC ACM and one or more CDC ECM class implementations is possible and no specific drivers are required.

Interface and endpoint descriptors of built-in USB class/function implementations in Zephyr RTOS typically have default interface numbers and endpoint addresses assigned in ascending order. During initialization, default interface numbers may be reassigned based on the number of interfaces in a given configuration. Endpoint addresses are reassigned based on controller capabilities, since certain endpoint combinations are not possible with every controller, and the number of interfaces in a given configuration. This also means that the device side class/function in the Zephyr RTOS must check the actual interface and endpoint descriptor values at runtime. This mechanism also allows as to provide generic samples and generic multifunction samples that are limited only by the resources provided by the controller, such as the number of endpoints and the size of the endpoint FIFOs.

There may be host drivers for a specific function, for example in the Linux Kernel, where the function driver does not read interface and endpoint descriptors to check interface numbers or endpoint addresses, but instead uses hardcoded values. Therefore, the host driver cannot be used in a generic way, meaning it cannot be used with different device controllers and different device configurations in combination with other functions. This may also be because the driver

is designed for a specific hardware and is not intended to be used with a clone of this specific hardware. On the contrary, if the driver is generic in nature and should work with different hardware variants, then it must not use hardcoded interface numbers and endpoint addresses. It is not possible to disable endpoint reassignment in Zephyr RTOS, which may prevent you from implementing a hardware-clone firmware. Instead, if possible, the host driver implementation should be fixed to use values from the interface and endpoint descriptor.

Testing over USBIP in native_sim A virtual USB controller implemented through USBIP might be used to test the USB device stack. Follow the general build procedure to build the USB sample for the native_sim configuration.

Run built sample with:

```
west build -t run
```

In a terminal window, run the following command to list USB devices:

```
$ usbip list -r localhost
Exportable USB devices
======================
 - 127.0.0.1
        1-1: unknown vendor : unknown product (2fe3:0100)
           : /sys/devices/pci0000:00/0000:00:01.2/usb1/1-1
           : (Defined at Interface level) (00/00/00)
           :  0 - Vendor Specific Class / unknown subclass / unknown protocol␣
↪(ff/00/00)
```

In a terminal window, run the following command to attach the USB device:

```
$ sudo usbip attach -r localhost -b 1-1
```

The USB device should be connected to your Linux host, and verified with the following commands:

```
$ sudo usbip port
Imported USB devices
====================
Port 00: <Port in Use> at Full Speed(12Mbps)
       unknown vendor : unknown product (2fe3:0100)
       7-1 -> usbip://localhost:3240/1-1
           -> remote bus/dev 001/002
$ lsusb -d 2fe3:0100
Bus 007 Device 004: ID 2fe3:0100
```

USB Vendor and Product identifiers The USB Vendor ID for the Zephyr project is `0x2FE3`. This USB Vendor ID must not be used when a vendor integrates Zephyr USB device support into its own product.

Each USB sample has its own unique Product ID. The USB maintainer, if one is assigned, or otherwise the Zephyr Technical Steering Committee, may allocate other USB Product IDs based on well-motivated and documented requests.

The following Product IDs are currently used:

Sample	PID
usb-cdc-acm	0x0001
Reserved (previously: usb-cdc-acm-composite)	0x0002
Reserved (previously: usb-hid-cdc)	0x0003
usb-cdc-acm-console	0x0004
usb-dfu (Run-Time)	0x0005
Reserved (previously: usb-hid)	0x0006
usb-hid-mouse	0x0007
usb-mass	0x0008
testusb-app	0x0009
webusb	0x000A
bluetooth_hci_usb	0x000B
Reserved (previously: bluetooth_hci_usb_h4)	0x000C
Reserved (previously: wpan-usb)	0x000D
uac2-explicit-feedback	0x000E
uac2-implicit-feedback	0x000F
uvc	0x0011
usb-dfu (DFU Mode)	0xFFFF

The USB device descriptor field `bcdDevice` (Device Release Number) represents the Zephyr kernel major and minor versions as a binary coded decimal value.

USB device support APIs (deprecated)

USB device controller driver API (deprecated) The USB device controller driver API is described in include/zephyr/drivers/usb/usb_dc.h and sometimes referred to as the `usb_dc` API.

This API has some limitations by design, it does not follow *Device Driver Model* and is being replaced by *USB device controller (UDC) driver API*.

API reference

USB Device Controller API

USB device stack API (deprecated)

API reference There are two ways to transmit data, using the 'low' level read/write API or the 'high' level transfer API.

Low level API

To transmit data to the host, the class driver should call usb_write(). Upon completion the registered endpoint callback will be called. Before sending another packet the class driver should wait for the completion of the previous write. When data is received, the registered endpoint callback is called. usb_read() should be used for retrieving the received data. For CDC ACM sample driver this happens via the OUT bulk endpoint handler (cdc_acm_bulk_out) mentioned in the endpoint array (cdc_acm_ep_data).

High level API

The usb_transfer method can be used to transfer data to/from the host. The transfer API will automatically split the data transmission into one or more USB transaction(s), depending endpoint max packet size. The class driver does not have to implement endpoint callback and should set this callback to the generic usb_transfer_ep_callback.

USB Device Core API

Related code samples

- Legacy console over USB CDC ACMOutput "Hello World!" to the console over USB CDC ACM.
- Legacy HCI USBTurn a Zephyr board into a USB Bluetooth dongle (compatible with all operating systems).
- Legacy USB Audio headsetImplement a USB Audio headset device with audio IN/OUT loopback.
- Legacy USB Audio microphone & headphonesImplement a USB Audio microphone + headphones device with audio IN/OUT loopback.
- Legacy USB CDC ACM UART sampleUse USB CDC ACM UART driver to implement a serial port echo.
- Legacy USB device netusb sampleUse USB CDC EEM/ECM or RNDIS implementation
- Legacy USB DFU (Device Firmware Upgrade)Implement device firmware upgrade using the USB DFU class driver.
- Legacy USB HID mouseImplement a basic HID mouse device.
- Legacy USB Mass StorageExpose board's RAM as a USB disk using USB Mass Storage driver.
- Legacy WebUSBReceive and echo data from a web page using WebUSB API.

USB HID Class API (deprecated) USB device specific part for HID support defined in include/zephyr/usb/class/usb_hid.h.

API Reference
HID class USB specific definitions

Related code samples

- Legacy USB HID mouseImplement a basic HID mouse device.

Binary Device Object Store (BOS) support API (deprecated)

API reference
USB BOS support

USB support

Common guidelines for USB tests and samples

Overview In general, all USB samples and tests are platform agnostic and should not require a platform-specific overlay. Though there may be exceptions to that already in the tree, the goal is to avoid platform-specific overlays. There is no obligation for a USB sample or USB test to support a specific platform.

Board configuration

Default USB device and host controller USB support uses the `zephyr_udc0` node label to assign the default USB device controller and the `zephyr_uhc0` node label to assign the default USB host controller. Depending on what a board supports, it has to assign these node labels to be able to execute samples and tests in the tree.

Supported features for the board metadata There are only two supported features for the board metadata file that are used by Twister to pick up a sample or a test. For a board with a USB controller supporting device mode, the feature is `usbd`. For a board with a USB controller supporting host mode, the feature is `usbh`.

See *Board Configuration* for more details.

Deprecated features The feature `usb_device` should not be used anymore. This feature belongs to the legacy USB device stack, which is deprecated and will be removed.

Tests Twister configuration USB tests may require additional hardware or software setup. These tests should use *fixtures* described in the table below.

Fixture	Use case
`usb_host_connec`	The test implements USB device functionality, and requires a USB host to be connected to the board under test.
`usb_device_conr`	The test implements USB host functionality, and requires a USB device to be connected to the board under test.

USB device support

Overview USB device support consists of the USB device controller (UDC) drivers , *USB device controller (UDC) driver API*, and USB device stack, *USB device stack API*. The *USB device controller (UDC) driver API* provides a generic and vendor independent interface to USB device controllers, and although, there is a clear separation between these layers, the purpose of *USB device controller (UDC) driver API* is to serve Zephyr's USB device stack exclusively.

The device stack supports multiple device controllers, meaning that if a SoC has multiple controllers, they can be used simultaneously. Full and high-speed device controllers are supported. It also provides support for registering multiple function or class instances to a configuration at runtime, or changing the configuration later. It has built-in support for several USB classes and provides an API to implement custom USB functions.

Samples

- usb-hid-keyboard
- uac2-explicit-feedback
- uac2-implicit-feedback
- uvc
- bluetooth_hci_usb
- usb-cdc-acm
- usb-cdc-acm-console
- usb-mass
- usb-hid-mouse
- zperf To build the sample for the device support, set the configuration overlay file `-DEXTRA_CONF_FILE=overlay-usbd.conf` and devicetree overlay file `-DDTC_OVERLAY_FILE=usbd_cdc_ecm.overlay` either directly or via `west`.

How to configure and enable USB device support For the USB device support samples in the Zephyr project repository, we have a common file for instantiation, configuration and initialization, samples/subsys/usb/common/sample_usbd_init.c. The following code snippets from this file are used as examples. USB Samples Kconfig options used in the USB samples and prefixed with SAMPLE_USBD_ have default values specific to the Zephyr project and the scope is limited to the project samples. In the examples below, you will need to replace these Kconfig options and other defaults with values appropriate for your application or hardware.

The USB device stack requires a context structure to manage its properties and runtime data. The preferred way to define a device context is to use the USBD_DEVICE_DEFINE macro. This creates a static usbd_context variable with a given name. Any number of contexts may be instantiated. A USB controller device can be assigned to multiple contexts, but only one context can be initialized and used at a time. Context properties must not be directly accessed or manipulated by the application.

```
/*
 * Instantiate a context named sample_usbd using the default USB device
 * controller, the Zephyr project vendor ID, and the sample product ID.
 * Zephyr project vendor ID must not be used outside of Zephyr samples.
 */
USBD_DEVICE_DEFINE(sample_usbd,
                   DEVICE_DT_GET(DT_NODELABEL(zephyr_udc0)),
                   CONFIG_SAMPLE_USBD_VID, CONFIG_SAMPLE_USBD_PID);
```

Your USB device may have manufacturer, product, and serial number string descriptors. To instantiate these string descriptors, the application should use the appropriate USBD_DESC_MANUFACTURER_DEFINE, USBD_DESC_PRODUCT_DEFINE, and USBD_DESC_SERIAL_NUMBER_DEFINE macros. String descriptors also require a single instantiation of the language descriptor using the USBD_DESC_LANG_DEFINE macro.

```
USBD_DESC_LANG_DEFINE(sample_lang);
USBD_DESC_MANUFACTURER_DEFINE(sample_mfr, CONFIG_SAMPLE_USBD_MANUFACTURER);
USBD_DESC_PRODUCT_DEFINE(sample_product, CONFIG_SAMPLE_USBD_PRODUCT);
IF_ENABLED(CONFIG_HWINFO, (USBD_DESC_SERIAL_NUMBER_DEFINE(sample_sn)));
```

String descriptors must be added to the device context at runtime before initializing the USB device with usbd_add_descriptor().

```
err = usbd_add_descriptor(&sample_usbd, &sample_lang);
if (err) {
        LOG_ERR("Failed to initialize language descriptor (%d)", err);
        return NULL;
}

err = usbd_add_descriptor(&sample_usbd, &sample_mfr);
if (err) {
        LOG_ERR("Failed to initialize manufacturer descriptor (%d)", err);
        return NULL;
}

err = usbd_add_descriptor(&sample_usbd, &sample_product);
if (err) {
        LOG_ERR("Failed to initialize product descriptor (%d)", err);
        return NULL;
}

IF_ENABLED(CONFIG_HWINFO, (
        err = usbd_add_descriptor(&sample_usbd, &sample_sn);
```

(continues on next page)

(continued from previous page)

```
))
if (err) {
        LOG_ERR("Failed to initialize SN descriptor (%d)", err);
        return NULL;
}
```

USB device requires at least one configuration instance per supported speed. The application should use USBD_CONFIGURATION_DEFINE to instantiate a configuration. Later, USB device functions are assigned to a configuration.

```
static const uint8_t attributes = (IS_ENABLED(CONFIG_SAMPLE_USBD_SELF_POWERED) ?
                                   USB_SCD_SELF_POWERED : 0) |
                                  (IS_ENABLED(CONFIG_SAMPLE_USBD_REMOTE_WAKEUP) ?
                                   USB_SCD_REMOTE_WAKEUP : 0);

/* Full speed configuration */
USBD_CONFIGURATION_DEFINE(sample_fs_config,
                          attributes,
                          CONFIG_SAMPLE_USBD_MAX_POWER, &fs_cfg_desc);

/* High speed configuration */
USBD_CONFIGURATION_DEFINE(sample_hs_config,
                          attributes,
                          CONFIG_SAMPLE_USBD_MAX_POWER, &hs_cfg_desc);
```

Each configuration instance for a specific speed must be added to the device context at runtime before the USB device is initialized using usbd_add_configuration(). Note USBD_SPEED_FS and USBD_SPEED_HS. The first full-speed or high-speed configuration will get bConfigurationValue one, and then further upward.

```
err = usbd_add_configuration(&sample_usbd, USBD_SPEED_FS,
                             &sample_fs_config);
if (err) {
        LOG_ERR("Failed to add Full-Speed configuration");
        return NULL;
}
```

Although we have already done a lot, this USB device has no function. A device can have multiple configurations with different set of functions at different speeds. A function or class can be registered on a USB device before it is initialized using usbd_register_class(). The desired configuration is specified using USBD_SPEED_FS or USBD_SPEED_HS and the configuration number. For simple cases, usbd_register_all_classes() can be used to register all available instances.

```
err = usbd_register_all_classes(&sample_usbd, USBD_SPEED_FS, 1, blocklist);
if (err) {
        LOG_ERR("Failed to add register classes");
        return NULL;
}
```

The last step in the preparation is to initialize the device with usbd_init(). After this, the configuration of the device cannot be changed. A device can be deinitialized with usbd_shutdown() and all instances can be reused, but the previous steps must be repeated. So it is possible to shutdown a device, register another type of configuration or function, and initialize it again. At the USB controller level, usbd_init() does only what is necessary to detect VBUS changes. There are controller types where the next step is only possible if a VBUS signal is present.

A function or class implementation may require its own specific configuration steps, which should be performed prior to initializing the USB device.

```
err = usbd_init(&sample_usbd);
if (err) {
        LOG_ERR("Failed to initialize device support");
        return NULL;
}
```

The final step to enable the USB device is usbd_enable(), after that, if the USB device is connected to a USB host controller, the host can start enumerating the device. The application can disable the USB device using usbd_disable().

```
ret = usbd_enable(sample_usbd);
if (ret) {
        LOG_ERR("Failed to enable device support");
        return ret;
}
```

USB Message notifications The application can register a callback using usbd_msg_register_cb() to receive message notification from the USB device support subsystem. The messages are mostly about the common device state changes, and a few specific types from the USB CDC ACM implementation.

```
err = usbd_msg_register_cb(&sample_usbd, msg_cb);
if (err) {
        LOG_ERR("Failed to register message callback");
        return NULL;
}
```

The helper function usbd_msg_type_string() can be used to convert usbd_msg_type to a human readable form for logging.

If the controller supports VBUS state change detection, the battery-powered application may want to enable the USB device only when it is connected to a host. A generic application should use usbd_can_detect_vbus() to check for this capability.

```
static void msg_cb(struct usbd_context *const usbd_ctx,
                   const struct usbd_msg *const msg)
{
        LOG_INF("USBD message: %s", usbd_msg_type_string(msg->type));

        if (msg->type == USBD_MSG_CONFIGURATION) {
                LOG_INF("\tConfiguration value %d", msg->status);
        }

        if (usbd_can_detect_vbus(usbd_ctx)) {
                if (msg->type == USBD_MSG_VBUS_READY) {
                        if (usbd_enable(usbd_ctx)) {
                                LOG_ERR("Failed to enable device support");
                        }
                }

                if (msg->type == USBD_MSG_VBUS_REMOVED) {
                        if (usbd_disable(usbd_ctx)) {
                                LOG_ERR("Failed to disable device support");
                        }
                }
        }
}
```

Built-in functions The USB device stack has built-in USB functions. Some can be used directly in the user application through a special API, such as HID or Audio class devices, while others use a general Zephyr RTOS driver API, such as MSC and CDC class implementations. The *Identification string* identifies a class or function instance (`n`) and is used as an argument to the `usbd_register_class()`.

Class or function	User API (if any)	Identification string
USB Audio 2 class	*Audio Class 2 device API*	`uac2_n`
USB CDC ACM class	*Universal Asynchronous Receiver-Transmitter (UART)*	`cdc_acm_n`
USB CDC ECM class	Ethernet device	`cdc_ecm_n`
USB Mass Storage Class (MSC)	*USB Mass Storage Class device API*	`msc_n`
USB Human Interface Devices (HID)	*HID device API*	`hid_n`
Bluetooth HCI USB transport layer	*HCI RAW channel*	`bt_hci_n`
USB Video Class (UVC)	Video device	`uvc_n`

USB Vendor and Product identifiers

The USB Vendor ID for the Zephyr project is `0x2FE3`. Zephyr USB Vendor ID must not be used when a vendor integrates Zephyr USB device support into its own product.

Each USB sample has its own unique Product ID. The USB maintainer, if one is assigned, or otherwise the Zephyr Technical Steering Committee, may allocate other USB Product IDs based on well-motivated and documented requests.

The following Product IDs are currently used:

Sample	PID
usb-cdc-acm	0x0001
Reserved (previously: usb-cdc-acm-composite)	0x0002
Reserved (previously: usb-hid-cdc)	0x0003
usb-cdc-acm-console	0x0004
usb-dfu (Run-Time)	0x0005
Reserved (previously: usb-hid)	0x0006
usb-hid-mouse	0x0007
usb-mass	0x0008
testusb-app	0x0009
webusb	0x000A
bluetooth_hci_usb	0x000B
Reserved (previously: bluetooth_hci_usb_h4)	0x000C
Reserved (previously: wpan-usb)	0x000D
uac2-explicit-feedback	0x000E
uac2-implicit-feedback	0x000F
uvc	0x0011
fido2	0x0012
i2c-tiny-usb	0x0013
usb-dfu (DFU Mode)	0xFFFF

USB device CDC ACM

The CDC ACM function provided by the USB device stack only implements Abstract Control Model Serial Emulation. Its sole purpose is to emulate serial lines, as its name suggests. Most modern operating systems should provide support for it out of the box.

The CDC ACM function is represented as a serial interface on both the host and device sides, while the user or application interface is the *Universal Asynchronous Receiver-Transmitter (UART)* driver API. This allows applications that already use the UART API to use the serial interface provided by the CDC ACM function without changing the code responsible for data communication. Only additional configuration and USB device stack initialization are required.

CDC ACM UART configuration Like the real UART controller, the virtual CDC ACM UART is described in the device tree. The devicetree compatible property for CDC ACM UART is `zephyr,cdc-acm-uart`.

CDC ACM support is automatically selected when USB device support is enabled and a compatible node in the devicetree sources is present. If necessary, CDC ACM support can be explicitly disabled by `CONFIG_USBD_CDC_ACM_CLASS`. The number of possible CDC ACM instances depends on the number of supported endpoints on the USB device controller. Each CDC ACM instance requires three endpoints: two bulk endpoints (one IN, one OUT) and one interrupt IN with a MaxPacketSize of 16. The CDC ACM node can use the `label` property to distinguish different interfaces on the host side. Below is an example of the devicetree overlay file.

```
&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

Before the application uses CDC ACM UART, it may want to wait for the DTR signal. Please refer to usb-cdc-acm on how to implement this functionality.

> **Note**
>
> To communicate with the host, beside the UART configuration, the application must enable USB device stack, for that please refer to *How to configure and enable USB device support* and read carefully the next chapter. For users and applications migrating from the legacy stack, this is the only part they need to adapt.

CDC ACM UART as serial backend Using the example above and the `zephyr,console` chosen node property, you can configure the CDC ACM UART as the console device.

```
/ {
        chosen {
                zephyr,console = &cdc_acm_uart0;
        };
};

&zephyr_udc0 {
        cdc_acm_uart0: cdc_acm_uart0 {
                compatible = "zephyr,cdc-acm-uart";
                label = "CDC_ACM_0";
        };
};
```

In the same way that the console in the above example is configured to use the CDC ACM UART, `zephyr,shell-uart` chosen node property can be used to configure the shell to use the CDC ACM UART as the serial backend. See sample shell-module and *chosen nodes documentation*.

Since the use case as a serial backend is very common and no configuration is necessary at runtime for the CDC ACM UART, the stack offers a helper that performs the steps described in *How to configure and enable USB device support*. The helper is enabled by the `CON-`

FIG_CDC_ACM_SERIAL_INITIALIZE_AT_BOOT and initializes the USB device stack with a single CDC ACM instance. Sample usb-cdc-acm-console demonstrates how to use it.

CONFIG_CDC_ACM_SERIAL_INITIALIZE_AT_BOOT should also be used by the boards like nrf52840dongle, which do not have a debug adapter but a USB device controller, and want to use CDC ACM UART as default serial backend for logging and shell. As the configuration would be identical for any board, there are common devicetree file and Kconfig file that must be included in the board's devicetree and Kconfig.defconfig files.

Using CDC ACM UART in the application CDC ACM implements a virtual UART controller and provides Interrupt-driven UART API and Polling UART API. The ASYNC API is not supported yet. If the application wants to communicate over CDC ACM UART, the preferable way is to use Interrupt-driven UART API. It is essential to understand API documentation, nevertheless, some notes below.

Interrupt-driven UART API Internally the CDC ACM UART implementation uses two ring-buffers. These take over the function of the TX/RX FIFOs (TX/RX buffers) from the *Interrupt-driven API*.

As described in the *Interrupt-driven API*, the functions uart_irq_update(), uart_irq_is_pending(), uart_irq_rx_ready(), uart_irq_tx_ready() uart_fifo_read(), and uart_fifo_fill() should be called from the interrupt handler, see uart_irq_callback_user_data_set(). To prevent undefined behavior, the implementation of these functions checks in what context they are called and fails if it is not an interrupt handler.

Also, as described in the UART API, uart_irq_is_pending() uart_irq_rx_ready(), and uart_irq_tx_ready() can only be called after uart_irq_update().

Simplified, application interrupt handler should look something like:

```
static void interrupt_handler(const struct device *dev, void *user_data)
{
        while (true) {
                uart_irq_update(dev);

                if (uart_irq_is_pending(dev) <= 0) {
                        break;
                }

                if (uart_irq_rx_ready(dev)) {
                        int len;
                        int n;

                        /* ... */
                        n = uart_fifo_read(dev, buffer, len);
                        /* ... */
                }

                if (uart_irq_tx_ready(dev)) {
                        int len;
                        int n;

                        /* ... */
                        n = uart_fifo_fill(dev, buffer, len);
                  /* ... */
                }
        }
}
```

All these functions are not directly dependent on the status of the USB device. Filling the TX FIFO does not mean that data is being sent to the host. And successfully reading the RX FIFO does not mean that the device is still connected to the host. If there is space in the TX FIFO, and the TX interrupt is enabled, `uart_irq_tx_ready()` will succeed. If there is data in the RX FIFO, and the RX interrupt is enabled, `uart_irq_rx_ready()` will succeed. Function `uart_irq_tx_complete()` is not implemented yet.

Polling UART API The CDC ACM poll out implementation follows *Polling API* and blocks when the TX FIFO is full only if the hw-flow-control property is enabled and called from a non-ISR context.

USB device support APIs

USB device controller (UDC) driver API The USB device controller driver API is described in include/zephyr/drivers/usb/udc.h and referred to as the `UDC driver` API.

UDC driver API is unstable and is subject to change without notice. It is a replacement for *USB device controller driver API (deprecated)*. If you wish to port an existing driver to UDC driver API, or add a new driver, please use drivers/usb/udc/udc_skeleton.c as a starting point.

API reference

USB Device Controller

USB buffer macros and definitions

USB device stack API USB device stack API is unstable and is subject to change without notice.

API reference

USB device core API

Related code samples

- Console over USB CDC ACMOutput "Hello World!" to the console over USB CDC ACM.
- HCI USBTurn a Zephyr board into a USB Bluetooth dongle (compatible with all operating systems).
- I2C-Tiny-USB bridgeExpose a Zephyr I2C controller to a host as a USB to I2C adapter.
- MCTP USB Endpoint Node SampleCreate an MCTP endpoint node using the USB device interface.
- USB Audio asynchronous explicit feedback sampleUSB Audio 2 explicit feedback sample playing audio on I2S.
- USB Audio asynchronous implicit feedback sampleUSB Audio 2 implicit feedback sample playing stereo and recording mono audio on I2S interface.
- USB CDC ACM UART sampleUse USB CDC ACM UART driver to implement a serial port echo.
- USB device testing applicationTest USB device drivers using a loopback function.
- USB DFUImplement a basic USB DFU device
- USB HID keyboardImplement a basic HID keyboard device.
- USB HID mouseImplement a basic HID mouse device.
- USB Mass StorageExpose board's RAM or FLASH as a USB disk using USB Mass Storage driver.

- USB MIDI2 deviceImplements a simple USB MIDI loopback and keyboard device.
- USB shellUse shell commands to interact with USB device stack.
- USB Video webcamSend video frames over USB.
- WebUSBReceive and echo data from a web page using WebUSB API.

USB device core API

HID device API HID device specific API defined in include/zephyr/usb/class/usbd_hid.h.

API Reference
USBD HID device API

Related code samples

- USB HID keyboardImplement a basic HID keyboard device.
- USB HID mouseImplement a basic HID mouse device.

Audio Class 2 device API USB Audio Class 2 device specific API defined in include/zephyr/usb/class/usbd_uac2.h.

API Reference
USB Audio Class 2 device API

Related code samples

- USB Audio asynchronous explicit feedback sampleUSB Audio 2 explicit feedback sample playing audio on I2S.
- USB Audio asynchronous implicit feedback sampleUSB Audio 2 implicit feedback sample playing stereo and recording mono audio on I2S interface.

USB Mass Storage Class device API USB Mass Storage Class device API defined in include/zephyr/usb/class/usbd_msc.h.

API Reference
USB Mass Storage Class device API

Related code samples

- USB Mass StorageExpose board's RAM or FLASH as a USB disk using USB Mass Storage driver.

MIDI 2.0 Class device API USB MIDI 2.0 device specific API defined in include/zephyr/usb/class/usbd_midi2.h.

API Reference
USB MIDI 2.0 Class device API

Related code samples

- USB MIDI2 deviceImplements a simple USB MIDI loopback and keyboard device.

MIDI2 Universal MIDI Packet definitions

Related code samples

- MIDI2 network transportExchange Universal MIDI Packets over the Network MIDI 2.0 protocol

USB DFU device API USB DFU device specific API defined in include/zephyr/usb/class/usbd_dfu.h.

API Reference
USB DFU device update API

Related code samples

- USB DFUImplement a basic USB DFU device

USB host support APIs

USB host controller (UHC) driver API The USB host controller driver API is described in include/zephyr/drivers/usb/uhc.h and referred to as the `UHC driver` API.

UHC driver API is experimental and is subject to change without notice.

Driver API reference
USB Host Controller

USB/IP protocol support

Overview New USB support includes initial support for the USB/IP protocol. It is still under development and is currently limited to supporting only one device connected to the host controller being exported.

USB/IP uses TCP/IP. Both of the underlying connectivity stacks, USB and networking, require significant memory resources, which must be considered when choosing a platform.

In the USB/IP protocol, a server exports the USB devices and a client imports them. USB/IP support in the Zephyr RTOS implements server functionality and exports a device connected to a host controller on a device running the Zephyr RTOS. A client, typically running the Linux kernel, imports this device. The USB/IP protocol is described in USB/IP protocol documentation.

To use USB/IP support, make sure the required modules are loaded on the client side.

```
modprobe vhci_hcd
modprobe usbip-core
modprobe usbip-host
```

On the client side, you will also need the **usbip** user tool. It can be installed using your Linux distribution's package management system or built from Linux kernel sources.

There are a few basic commands for everyday use. To list exported USB devices, run the following command:

```
$ usbip list -r 192.0.2.1
Exportable USB devices
======================
 - 192.0.2.1
        1-1: NordicSemiconductor : unknown product (2fe3:0001)
           : /sys/bus/usb/devices/usb1/1-1
           : Miscellaneous Device / ? / Interface Association (ef/02/01)
           :  0 - Communications / Abstract (modem) / None (02/02/00)
           :  1 - CDC Data / Unused / unknown protocol (0a/00/00)
```

To attach an exported device with busid 1-1:

```
$ sudo usbip attach -r 192.0.2.1 -b 1-1
```

To detach an exported device on port 0:

```
$ sudo usbip detach -p 0
```

USB/IP with native_sim The preferred method to develop with USB/IP support enabled is to use native_sim. Use on real hardware is not really tested yet. USB/IP requires a network connection, see *Networking with native_sim board* for how to set up the interface on the client side.

Building and running a sample with USB/IP requires extensive configuration, you can use usbip-native-sim snippet to configure host and USB/IP support.

```
# From the root of the zephyr repository
west build -b native_sim/native/64 samples/subsys/usb/cdc_acm -- -
↪DSNIPPET=usbip-native-sim -DEXTRA_DTC_OVERLAY_FILE=app.overlay
```

USB Power Delivery support

USB-C device stack

The USB-C device stack is a hardware independent interface between a Type-C Port Controller (TCPC) and customer applications. It is a port of the Google ChromeOS Type-C Port Manager (TCPM) stack. It provides the following functionalities:

- Uses the APIs provided by the Type-C Port Controller drivers to interact with the Type-C Port Controller.
- Provides a programming interface that's used by a customer applications. The APIs is described in include/zephyr/usb_c/usbc.h

Configuration options The USB-C device stack supports implementation of Sink only, Source only, and Dual Role Power (DRP) devices.

- `CONFIG_USBC_CSM_SINK_ONLY`: Sink USB-C Connection State Machine
- `CONFIG_USBC_CSM_SOURCE_ONLY`: Source USBC Connection State Machine
- `CONFIG_USBC_CSM_DRP`: Dual Role Power (DRP) USB-C Connection State Machine

List of samples for different purposes.

Implementing a Sink Type-C and Power Delivery USB-C device The configuration of a USB-C Device is done in the stack layer and devicetree.

The following devicetree, structures and callbacks need to be defined:

- Devicetree usb-c-connector node referencing a TCPC
- Devicetree vbus node referencing a VBUS measurement device

- User defined structure that encapsulates application specific data
- Policy callbacks

For example, for the Sample USB-C Sink application:

Each Physical Type-C port is represented in the devicetree by a usb-c-connector compatible node:

```
                port1: usbc-port@1 {
                        compatible = "usb-c-connector";
                        reg = <1>;
                        tcpc = <&ucpd1>;
                        vbus = <&vbus1>;
                        power-role = "sink";
                        sink-pdos = <PDO_FIXED(5000, 100, 0)>;
                };
```

VBUS is measured by a device that's referenced in the devicetree by a usb-c-vbus-adc compatible node:

```
        vbus1: vbus {
                compatible = "zephyr,usb-c-vbus-adc";
                io-channels = <&adc2 8>;
                output-ohms = <49900>;
                full-ohms = <(330000 + 49900)>;
        };
```

A user defined structure is defined and later registered with the subsystem and can be accessed from callback through an API:

```
/**
 * @brief A structure that encapsulates Port data.
 */
static struct port0_data_t {
        /** Sink Capabilities */
        uint32_t snk_caps[DT_PROP_LEN(USBC_PORT0_NODE, sink_pdos)];
        /** Number of Sink Capabilities */
        int snk_cap_cnt;
        /** Source Capabilities */
        uint32_t src_caps[PDO_MAX_DATA_OBJECTS];
        /** Number of Source Capabilities */
        int src_cap_cnt;
        /* Power Supply Ready flag */
        atomic_t ps_ready;
} port0_data = {.snk_caps = DT_PROP(USBC_PORT0_NODE, sink_pdos),
                .snk_cap_cnt = DT_PROP_LEN(USBC_PORT0_NODE, sink_pdos),
                .src_caps = {0},
                .src_cap_cnt = 0,
                .ps_ready = 0};

```

These callbacks are used by the subsystem to set or get application specific data:

```
static int port0_policy_cb_get_snk_cap(const struct device *dev, uint32_t **pdos,␣
↪int *num_pdos)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        *pdos = dpm_data->snk_caps;
        *num_pdos = dpm_data->snk_cap_cnt;
```

(continues on next page)

(continued from previous page)

```

        return 0;
}

static void port0_policy_cb_set_src_cap(const struct device *dev, const uint32_t *pdos,
                                        const int num_pdos)
{
        struct port0_data_t *dpm_data;
        int num;
        int i;

        dpm_data = usbc_get_dpm_data(dev);

        num = num_pdos;
        if (num > PDO_MAX_DATA_OBJECTS) {
                num = PDO_MAX_DATA_OBJECTS;
        }

        for (i = 0; i < num; i++) {
                dpm_data->src_caps[i] = *(pdos + i);
        }

        dpm_data->src_cap_cnt = num;
}

static uint32_t port0_policy_cb_get_rdo(const struct device *dev)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        return build_rdo(dpm_data);
}
```

This callback is used by the subsystem to query if a certain action can be taken:

```
bool port0_policy_check(const struct device *dev, const enum usbc_policy_check_t policy_check)
{
        switch (policy_check) {
        case CHECK_POWER_ROLE_SWAP:
                /* Reject power role swaps */
                return false;
        case CHECK_DATA_ROLE_SWAP_TO_DFP:
                /* Reject data role swap to DFP */
                return false;
        case CHECK_DATA_ROLE_SWAP_TO_UFP:
                /* Accept data role swap to UFP */
                return true;
        case CHECK_SNK_AT_DEFAULT_LEVEL:
                /* This device is always at the default power level */
                return true;
        default:
                /* Reject all other policy checks */
                return false;
        }
}
```

This callback is used by the subsystem to notify the application of an event:

```
static void port0_notify(const struct device *dev, const enum usbc_policy_notify_t␣
↪policy_notify)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        switch (policy_notify) {
        case PROTOCOL_ERROR:
                break;
        case MSG_DISCARDED:
                break;
        case MSG_ACCEPT_RECEIVED:
                break;
        case MSG_REJECTED_RECEIVED:
                break;
        case MSG_NOT_SUPPORTED_RECEIVED:
                break;
        case TRANSITION_PS:
                atomic_set_bit(&dpm_data->ps_ready, 0);
                break;
        case PD_CONNECTED:
                break;
        case NOT_PD_CONNECTED:
                break;
        case POWER_CHANGE_0A0:
                LOG_INF("PWR 0A");
                break;
        case POWER_CHANGE_DEF:
                LOG_INF("PWR DEF");
                break;
        case POWER_CHANGE_1A5:
                LOG_INF("PWR 1A5");
                break;
        case POWER_CHANGE_3A0:
                LOG_INF("PWR 3A0");
                break;
        case DATA_ROLE_IS_UFP:
                break;
        case DATA_ROLE_IS_DFP:
                break;
        case PORT_PARTNER_NOT_RESPONSIVE:
                LOG_INF("Port Partner not PD Capable");
                break;
        case SNK_TRANSITION_TO_DEFAULT:
                break;
        case HARD_RESET_RECEIVED:
                break;
        case SENDER_RESPONSE_TIMEOUT:
                break;
        case SOURCE_CAPABILITIES_RECEIVED:
                break;
        default:
        }
}
```

Registering the callbacks:

```
        /* Register USB-C Callbacks */

        /* Register Policy Check callback */
        usbc_set_policy_cb_check(usbc_port0, port0_policy_check);
        /* Register Policy Notify callback */
        usbc_set_policy_cb_notify(usbc_port0, port0_notify);
        /* Register Policy Get Sink Capabilities callback */
        usbc_set_policy_cb_get_snk_cap(usbc_port0, port0_policy_cb_get_snk_cap);
        /* Register Policy Set Source Capabilities callback */
        usbc_set_policy_cb_set_src_cap(usbc_port0, port0_policy_cb_set_src_cap);
        /* Register Policy Get Request Data Object callback */
        usbc_set_policy_cb_get_rdo(usbc_port0, port0_policy_cb_get_rdo);
```

Register the user defined structure:

```
        /* Set Application port data object. This object is passed to the policy␣
↪callbacks */
        port0_data.ps_ready = ATOMIC_INIT(0);
        usbc_set_dpm_data(usbc_port0, &port0_data);
```

Start the USB-C subsystem:

```
        /* Start the USB-C Subsystem */
        usbc_start(usbc_port0);
```

Implementing a Source Type-C and Power Delivery USB-C device The configuration of a USB-C Device is done in the stack layer and devicetree.

Define the following devicetree, structures and callbacks:

- Devicetree `usb-c-connector` node referencing a TCPC
- Devicetree `vbus` node referencing a VBUS measurement device
- Devicetree `pwrctrl` node for VBUS and VCONN power control
- User defined structure that encapsulates application specific data
- Policy callbacks

For example, for the Sample USB-C Source application:

Each Physical Type-C port is represented in the devicetree by a `usb-c-connector` compatible node:

```
                port1: usbc-port@1 {
                        compatible = "usb-c-connector";
                        reg = <1>;
                        tcpc = <&ucpd1>;
                        vbus = <&vbus1>;
                        power-role = "source";
                        typec-power-opmode = "3.0A";
                        source-pdos = <PDO_FIXED(5000, 100, 0) PDO_FIXED(9000, 100,
↪ 0)
                                       PDO_FIXED(15000, 100, 0)>;
                };
```

VBUS is measured by a device that's referenced in the devicetree by a `usb-c-vbus-adc` compatible node:

```
        vbus1: vbus {
                compatible = "zephyr,usb-c-vbus-adc";
                io-channels = <&adc1 9>;
                output-ohms = <49900>;
                full-ohms = <(330000 + 49900)>;

                /* Pin B13 is used to control VBUS Discharge for Port1 */
                discharge-gpios = <&gpiob 13 GPIO_ACTIVE_HIGH>;
        };
```

Power control for VBUS and VCONN can be managed by a device referenced in the devicetree by a `zephyr,usb-c-pwrctrl` compatible node:

```
        pwrctrl1: pwrctrl {
                compatible = "zephyr,usb-c-pwrctrl";

                /* Pin D3 is used to enable VBUS Source */
                source-en-gpios = <&gpiod 3 GPIO_ACTIVE_HIGH>;

                /* Pin A1 is used for DCDC_EN */
                dcdc-en-gpios = <&gpioa 1 GPIO_ACTIVE_HIGH>;

                /* Pin D4 enables VCONN on CC1 */
                vconn1-en-gpios = <&gpiod 4 GPIO_ACTIVE_LOW>;

                /* Pin B9 enables VCONN on CC2 */
                vconn2-en-gpios = <&gpiob 9 GPIO_ACTIVE_LOW>;

                /* PWM for voltage selection */
                pwms = <&pwm15 1 PWM_USEC(50) PWM_POLARITY_INVERTED>;
        };
```

A user defined structure is defined and later registered with the subsystem and can be accessed from callback through an API:

```
/**
 * @brief A structure that encapsulates Port data.
 */
static struct port0_data_t {
        /** Power controller device */
        const struct device *pwrctrl;
        /** Source Capabilities */
        uint32_t src_caps[DT_PROP_LEN(USBC_PORT0_NODE, source_pdos)];
        /** Number of Source Capabilities */
        int src_cap_cnt;
        /** CC Rp value */
        int rp;
        /** Sink Request RDO */
        union pd_rdo sink_request;
        /** Requested Object Pos */
        int obj_pos;
        /** VCONN CC line*/
        enum tc_cc_polarity vconn_pol;
        /** True if power supply is ready */
        bool ps_ready;
        /** True if power supply should transition to a new level */
        bool ps_tran_start;
```

(continues on next page)

(continued from previous page)

```
        /** Log Sink Requested RDO to console */
        atomic_t show_sink_request;
} port0_data = {
        .rp = DT_ENUM_IDX(USBC_PORT0_NODE, typec_power_opmode),
        .src_caps = DT_PROP(USBC_PORT0_NODE, source_pdos),
        .src_cap_cnt = DT_PROP_LEN(USBC_PORT0_NODE, source_pdos),
};

```

These callbacks are used by the subsystem to set or get application specific data:

```
/**
 * @brief PE calls this function when it needs to set the Rp on CC
 */
int port0_policy_cb_get_src_rp(const struct device *dev, enum tc_rp_value *rp)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        *rp = dpm_data->rp;

        return 0;
}

/**
 * @brief PE calls this function to Enable (5V) or Disable (0V) the
 *          Power Supply
 */
int port0_policy_cb_src_en(const struct device *dev, bool en)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        source_ctrl_set(dpm_data->pwrctrl, en ? SOURCE_5V : SOURCE_0V);

        return 0;
}

/**
 * @brief PE calls this function to Enable or Disable VCONN
 */
int port0_policy_cb_vconn_en(const struct device *tcpc_dev, const struct device␣
↪*usbc_dev,
                             enum tc_cc_polarity pol, bool en)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(usbc_dev);

        dpm_data->vconn_pol = pol;

        if (en == false) {
                /* Disable VCONN on CC1 and CC2 */
                vconn_ctrl_set(dpm_data->pwrctrl, VCONN_OFF);
        } else if (pol == TC_POLARITY_CC1) {
                /* set VCONN on CC1 */
                vconn_ctrl_set(dpm_data->pwrctrl, VCONN1_ON);
        } else {
                /* set VCONN on CC2 */
                vconn_ctrl_set(dpm_data->pwrctrl, VCONN2_ON);
```

(continues on next page)

(continued from previous page)

```
        }

        return 0;
}

/**
 * @brief PE calls this function to get the Source Caps that will be sent
 *          to the Sink
 */
int port0_policy_cb_get_src_caps(const struct device *dev, const uint32_t **pdos,
                                 uint32_t *num_pdos)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        *pdos = dpm_data->src_caps;
        *num_pdos = dpm_data->src_cap_cnt;

        return 0;
}

/**
 * @brief PE calls this function to verify that a Sink's request if valid
 */
static enum usbc_snk_req_reply_t port0_policy_cb_check_sink_request(const struct device *dev,
                                                                    const uint32_t request_msg)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);
        union pd_fixed_supply_pdo_source pdo;
        uint32_t obj_pos;
        uint32_t op_current;

        dpm_data->sink_request.raw_value = request_msg;
        obj_pos = dpm_data->sink_request.fixed.object_pos;
        op_current =
                PD_CONVERT_FIXED_PDO_CURRENT_TO_MA(dpm_data->sink_request.fixed.operating_current);

        if (obj_pos == 0 || obj_pos > dpm_data->src_cap_cnt) {
                return SNK_REQUEST_REJECT;
        }

        pdo.raw_value = dpm_data->src_caps[obj_pos - 1];

        if (dpm_data->sink_request.fixed.operating_current > pdo.max_current) {
                return SNK_REQUEST_REJECT;
        }

        dpm_data->obj_pos = obj_pos;

        atomic_set_bit(&port0_data.show_sink_request, 0);

        /*
         * Clear PS ready. This will be set to true after PS is ready after
         * it transitions to the new level.
```

(continues on next page)

(continued from previous page)

```
         */
        port0_data.ps_ready = false;

        return SNK_REQUEST_VALID;
}

/**
 * @brief PE calls this function to check if the Power Supply is at the requested
 *          level
 */
static bool port0_policy_cb_is_ps_ready(const struct device *dev)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        /* Return true to inform that the Power Supply is ready */
        return dpm_data->ps_ready;
}

/**
 * @brief PE calls this function to check if the Present Contract is still
 *          valid
 */
static bool port0_policy_cb_present_contract_is_valid(const struct device *dev,
                                                      const uint32_t present_contract)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);
        union pd_fixed_supply_pdo_source pdo;
        union pd_rdo request;
        uint32_t obj_pos;
        uint32_t op_current;

        request.raw_value = present_contract;
        obj_pos = request.fixed.object_pos;
        op_current = PD_CONVERT_FIXED_PDO_CURRENT_TO_MA(request.fixed.operating_current);

        if (obj_pos == 0 || obj_pos > dpm_data->src_cap_cnt) {
                return false;
        }

        pdo.raw_value = dpm_data->src_caps[obj_pos - 1];

        if (request.fixed.operating_current > pdo.max_current) {
                return false;
        }

        return true;
}

```

This callback is used by the subsystem to query if a certain action can be taken:

```
bool port0_policy_check(const struct device *dev, const enum usbc_policy_check_t policy_check)
{
```

(continues on next page)

(continued from previous page)

```
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        switch (policy_check) {
        case CHECK_POWER_ROLE_SWAP:
                /* Reject power role swaps */
                return false;
        case CHECK_DATA_ROLE_SWAP_TO_DFP:
                /* Accept data role swap to DFP */
                return true;
        case CHECK_DATA_ROLE_SWAP_TO_UFP:
                /* Reject data role swap to UFP */
                return false;
        case CHECK_SRC_PS_AT_DEFAULT_LEVEL:
                /*
                 * This check is sent from the PE_SRC_Transition_to_default
                 * state and requires the following:
                 *        1: Vconn should be turned ON
                 *        2: Return TRUE when Power Supply is at default level
                 */

                /* Power on VCONN */
                vconn_ctrl_set(dpm_data->pwrctrl, dpm_data->vconn_pol);

                /* PS should be at default level after receiving a Hard Reset */
                return true;
        default:
                /* Reject all other policy checks */
                return false;
        }
}
```

This callback is used by the subsystem to notify the application of an event:

```
static void port0_notify(const struct device *dev, const enum usbc_policy_notify_t␣
↪policy_notify)
{
        struct port0_data_t *dpm_data = usbc_get_dpm_data(dev);

        switch (policy_notify) {
        case PROTOCOL_ERROR:
                break;
        case MSG_DISCARDED:
                break;
        case MSG_ACCEPT_RECEIVED:
                break;
        case MSG_REJECTED_RECEIVED:
                break;
        case MSG_NOT_SUPPORTED_RECEIVED:
                break;
        case TRANSITION_PS:
                dpm_data->ps_tran_start = true;
                break;
        case PD_CONNECTED:
                break;
        case NOT_PD_CONNECTED:
                break;
```

(continues on next page)

(continued from previous page)

```
        case DATA_ROLE_IS_UFP:
                break;
        case DATA_ROLE_IS_DFP:
                break;
        case PORT_PARTNER_NOT_RESPONSIVE:
                LOG_INF("Port Partner not PD Capable");
                break;
        case HARD_RESET_RECEIVED:
                /*
                 * This notification is sent from the PE_SRC_Transition_to_default
                 * state and requires the following:
                 *        1: Vconn should be turned OFF
                 *        2: Reset of the local hardware
                 */

                /* Power off VCONN */
                vconn_ctrl_set(dpm_data->pwrctrl, VCONN_OFF);
                /* Transition PS to Default level */
                source_ctrl_set(dpm_data->pwrctrl, SOURCE_5V);
                break;
        default:
        }
}
```

Registering the callbacks:

```
        /* Register USB-C Callbacks */

        /* Register Policy Check callback */
        usbc_set_policy_cb_check(usbc_port0, port0_policy_check);
        /* Register Policy Notify callback */
        usbc_set_policy_cb_notify(usbc_port0, port0_notify);
        /* Register Policy callback to set the Rp on CC lines */
        usbc_set_policy_cb_get_src_rp(usbc_port0, port0_policy_cb_get_src_rp);
        /* Register Policy callback to enable or disable power supply */
        usbc_set_policy_cb_src_en(usbc_port0, port0_policy_cb_src_en);
        /* Register Policy callback to enable or disable vconn */
        usbc_set_vconn_control_cb(usbc_port0, port0_policy_cb_vconn_en);
        /* Register Policy callback to send the source caps to the sink */
        usbc_set_policy_cb_get_src_caps(usbc_port0, port0_policy_cb_get_src_caps);
        /* Register Policy callback to check if the sink request is valid */
        usbc_set_policy_cb_check_sink_request(usbc_port0, port0_policy_cb_check_sink_request);
        /* Register Policy callback to check if the power supply is ready */
        usbc_set_policy_cb_is_ps_ready(usbc_port0, port0_policy_cb_is_ps_ready);
        /* Register Policy callback to check if Present Contract is still valid */
        usbc_set_policy_cb_present_contract_is_valid(usbc_port0, port0_policy_cb_present_contract_is_valid);

```

Register the user defined structure:

```
        /* Set Application port data object. This object is passed to the policy callbacks */
        usbc_set_dpm_data(usbc_port0, &port0_data);
```

Start the USB-C subsystem:

```
        /* Start the USB-C Subsystem */
        usbc_start(usbc_port0);
```

Implementing a Dual Role Power (DRP) USB-C device DRP devices can operate as both Source and Sink, automatically negotiating the appropriate role with the port partner. When unattached, the device toggles between Source (Rp) and Sink (Rd) CC line advertisements to detect and connect with any partner type. Once an attach is detected, the device enters the appropriate attached state (Attached.SRC or Attached.SNK) and starts the corresponding Policy Engine state machine (PE_SRC or PE_SNK) to negotiate power delivery.

Configuration is similar to Source and Sink devices, with the following key differences:

- Set `power-role = "dual"` in the devicetree `usb-c-connector` node
- Implement callbacks for both Source and Sink operations

The DRP toggle behavior can be configured via Kconfig:

- `CONFIG_USBC_DRP_PERIOD_MS`: Toggle period (50-100ms, default 75ms)
- `CONFIG_USBC_DRP_DUTY_CYCLE`: Percentage of time as Source (30-70%, default 50%)

See usb-c-drp for a complete example.

API reference

USB-C Device API

Related code samples

- Basic USB-C DRPImplement a USB-C Power Delivery application in the form of a USB-C DRP (Dual Role Power).
- Basic USB-C SinkImplement a USB-C Power Delivery application in the form of a USB-C Sink.
- Basic USB-C SourceImplement a USB-C Power Delivery application in the form of a USB-C Source.

SINK callback reference

Sink_callbacks

SOURCE callback reference

Source_callbacks

Common sections related to USB support

Human Interface Devices (HID)

Common USB HID part that can be used outside of USB support, defined in header file include/zephyr/usb/class/hid.h.

HID types reference

USB HID common definitions

HID items reference

USB HID Item helpers

HID Mouse and Keyboard report descriptors The pre-defined Mouse and Keyboard report descriptors can be used by a HID device implementation or simply as examples.

Mouse and keyboard report descriptors

4.1.4 Controller Area Network (CAN) Bus Protocols

ISO-TP Transport Protocol

- *Overview*
- *API Reference*

Overview ISO-TP is a transport protocol defined in the ISO-Standard ISO15765-2 Road vehicles - Diagnostic communication over Controller Area Network (DoCAN). Part2: Transport protocol and network layer services. As its name already implies, it is originally designed, and still used in road vehicle diagnostic over Controller Area Networks. Nevertheless, it's not limited to applications in road vehicles or the automotive domain.

This transport protocol extends the limited payload data size for classical CAN (8 bytes) and CAN FD (64 bytes) to theoretically four gigabytes. Additionally, it adds a flow control mechanism to influence the sender's behavior. ISO-TP segments packets into small fragments depending on the payload size of the CAN frame. The header of those segments is called Protocol Control Information (PCI).

Packets smaller or equal to seven bytes on Classical CAN are called single-frames (SF). They don't need to fragment and do not have any flow-control.

Packets larger than that are segmented into a first-frame (FF) and as many consecutive-frames (CF) as required. The FF contains information about the length of the entire payload data and additionally, the first few bytes of payload data. The receiving peer sends back a flow-control-frame (FC) to either deny, postpone, or accept the following consecutive frames. The FC also defines the conditions of sending, namely the block-size (BS) and the minimum separation time between frames (STmin). The block size defines how many CF the sender is allowed to send, before he has to wait for another FC.

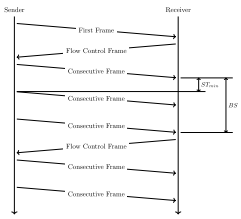


Figure 1: Example Sequence of a Segmented Packet with a BS of three

API Reference
CAN ISO-TP Protocol

Related code samples

- ISO-TP libraryUse ISO-TP library to exchange messages between two boards.

4.1.5 LoRa and LoRaWAN

Overview

LoRa (abbrev. for Long Range) is a proprietary low-power wireless communication protocol developed by the Semtech Corporation.

LoRa acts as the physical layer (PHY) based on the chirp spread spectrum (CSS) modulation technique.

LoRaWAN (for Long Range Wide Area Network) defines a networking layer on top of the LoRa PHY.

Zephyr provides APIs for LoRa to send raw data packets directly over the wireless interface as well as APIs for LoRaWAN to connect the end device to the internet through a gateway.

Zephyr provides two LoRaWAN backend implementations:

- **LoRaMac-node** (default): Based on Semtech's LoRaMac-node library, included as a Zephyr module. Supports all regions defined by the LoRaWAN specification. Selected with `CONFIG_LORA_MODULE_BACKEND_LORAMAC_NODE`.
- **Native**: A Zephyr-idiomatic LoRaWAN 1.0.x Class A implementation that talks directly to the LoRa radio driver without external dependencies. Currently supports the EU868 region. Selected with `CONFIG_LORA_MODULE_BACKEND_NATIVE`.

> **Note**
>
> `LoRaMac-node` has been deprecated by Semtech in favor of LoRa Basics Modem. Porting the Zephyr API's to use `LoRa Basics Modem` as the backend is in progress.
>
> Currently, only the base LoRa API is supported for the SX1261, SX1262, SX1272 and SX1276 chipsets through `CONFIG_LORA_MODULE_BACKEND_LORA_BASICS_MODEM`.

The LoRaWAN specification is published by the LoRa Alliance.

Configuration Options

LoRa PHY Related configuration options can be found under drivers/lora/Kconfig.

- `CONFIG_LORA`
- `CONFIG_LORA_SHELL`
- `CONFIG_LORA_INIT_PRIORITY`

LoRaWAN Related configuration options can be found under subsys/lorawan/Kconfig.

- `CONFIG_LORAWAN`
- `CONFIG_LORAWAN_SYSTEM_MAX_RX_ERROR`
- `CONFIG_LORAWAN_REGION_AS923`
- `CONFIG_LORAWAN_REGION_AU915`
- `CONFIG_LORAWAN_REGION_CN470`
- `CONFIG_LORAWAN_REGION_CN779`
- `CONFIG_LORAWAN_REGION_EU433`
- `CONFIG_LORAWAN_REGION_EU868`
- `CONFIG_LORAWAN_REGION_KR920`
- `CONFIG_LORAWAN_REGION_IN865`
- `CONFIG_LORAWAN_REGION_US915`

- CONFIG_LORAWAN_REGION_RU864

Native Backend The native backend is selected with CONFIG_LORA_MODULE_BACKEND_NATIVE and has additional options under subsys/lorawan/native/Kconfig:

- CONFIG_LORAWAN_NATIVE_ENGINE_STACK_SIZE
- CONFIG_LORAWAN_NATIVE_ENGINE_PRIORITY
- CONFIG_LORAWAN_NATIVE_PUBLIC_NETWORK
- CONFIG_LORAWAN_NATIVE_DUTY_CYCLE

API Reference

LoRa PHY
LoRa

Related code samples

- LoRa duty cycleReceive LoRa packets using hardware RX duty cycling (wake-on-radio).
- LoRa receiveReceive packets in both synchronous and asynchronous mode using the LoRa radio.
- LoRa sendTransmit a preconfigured payload every second using the LoRa radio.

LoRaWAN
LoRaWAN APIs

Related code samples

- LoRaWAN class A deviceJoin a LoRaWAN network and send a message periodically.
- LoRaWAN FUOTAPerform a LoRaWAN firmware-upgrade over the air (FUOTA) operation.

4.1.6 Modbus

Modbus is an industrial messaging protocol. The protocol is specified for different types of networks or buses. Zephyr OS implementation supports communication over serial line and may be used with different physical interfaces, like RS485 or RS232. TCP support is not implemented directly, but there are helper functions to realize TCP support according to the application's needs.

Modbus communication is based on client/server model. Only one client may be present on the bus. Client can communicate with several server devices. Server devices themselves are passive and must not send requests or unsolicited responses. Services requested by the client are specified by function codes (FCxx), and can be found in the specification or documentation of the API below.

Zephyr RTOS implementation supports both client and server roles.

More information about Modbus and Modbus RTU can be found on the website MODBUS Protocol Specifications.

Samples

- modbus-rtu-server and modbus-rtu-client samples give the possibility to try out RTU server and RTU client implementation with an evaluation board.
- modbus-tcp-server sample is a simple Modbus TCP server.

- modbus-gateway sample shows how to build a TCP to serial line gateway with Zephyr OS.

API Reference

MODBUS

Related code samples

- Modbus RTU clientCommunicate with a Modbus RTU server.
- Modbus RTU serverImplement a Modbus RTU server exposing Modbus commands to control LEDs.
- Modbus TCP serverImplement a Modbus TCP server exposing Modbus commands to control LEDs.
- Modbus TCP-to-serial gatewayImplement a gateway between an Ethernet TCP-IP network and a Modbus serial line.

4.1.7 Modem modules

This service provides modules necessary to communicate with modems.

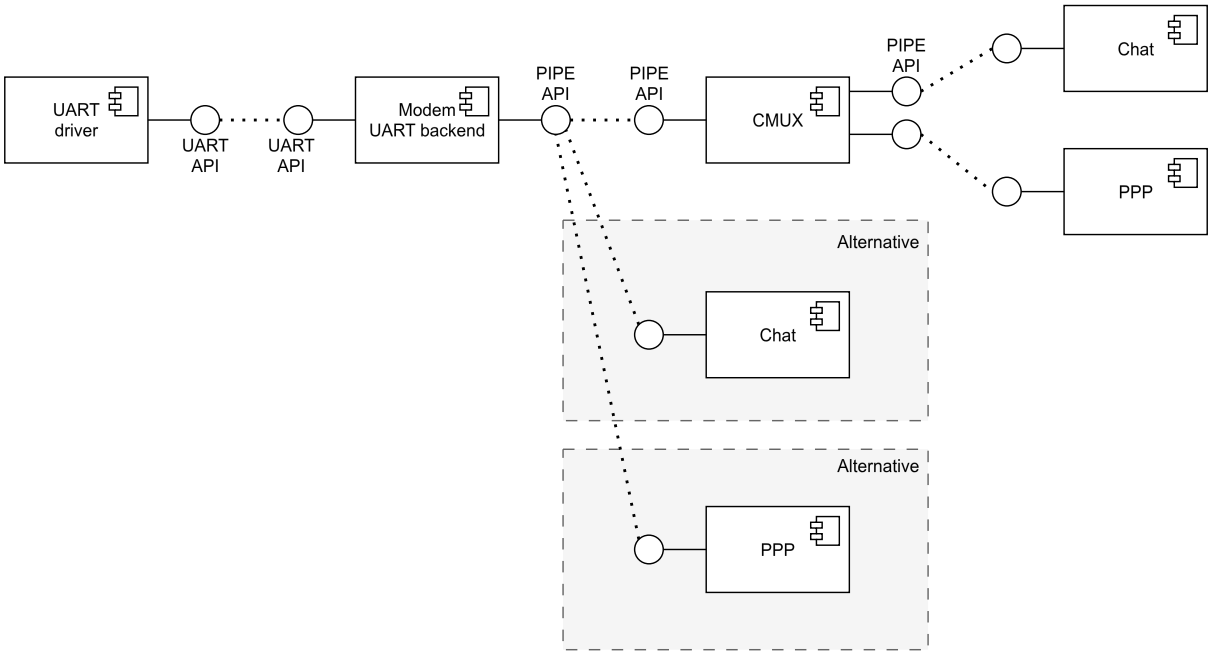
Modems are self-contained devices that implement the hardware and software necessary to perform RF (Radio-Frequency) communication, including GNSS, Cellular, WiFi etc.

The modem modules are inter-connected dynamically using data-in/data-out pipes making them independently testable and highly flexible, ensuring stability and scalability.

Modem pipe

This module is used to abstract data-in/data-out communication over a variety of mechanisms, like UART and CMUX DLCI channels, in a thread-safe manner.

A modem backend will internally contain an instance of a modem_pipe structure, alongside any buffers and additional structures required to abstract away its underlying mechanism.



The modem backend will return a pointer to its internal modem_pipe structure when initialized, which will be used to interact with the backend through the modem pipe API.

Modem Pipe

Related code samples

- AT Command ClientCreate a bidirectional UART bridge between console and modem for AT command communication.

Modem PPP

This module defines and binds a L2 PPP network interface, described in *L2 Layer Management*, to a modem backend. The L2 PPP interface sends and receives network packets. These network packets have to be wrapped in PPP frames before being transported via a modem backend. This module performs said wrapping.

Modem PPP

Modem CMUX

This module is an implementation of CMUX following the 3GPP 27.010 specification. CMUX is a multiplexing protocol, allowing for multiple bi-directional streams of data, called DLCI channels. The module attaches to a single modem backend, exposing multiple modem backends, each representing a DLCI channel.

Protocol defines simple framing for splitting each DLC into small chunks of data.

Flag	Address	Control	Length indicator	Information	FCS	Flag
1 octet	1 octet	1 octet	1 or 2 octet	Unspecified length	1 octet	1 octet

Zephyr implements the basic frame type, with build-time configurable MTU size.

The module also implements power-saving using CMUX Power Saving Command (PSC) for modems that support it. See the *CMUX Power Saving* section below for more details.

Modem CMUX

Modem pipelink

This module is used to share modem pipes globally. This module aims to decouple the creation and setup of modem pipes in device drivers from the users of said pipes. See drivers/modem/modem_at_shell.c and drivers/modem/modem_cellular.c for examples of how to use the modem pipelink between device driver and application.

Modem pipelink

Modem chat

This module implements scripted AT command communication with a modem. AT commands are organized into **scripts**, where each script is a sequence of command–response exchanges. The module sends a command string through a modem pipe and then waits for a response matching one of the expected patterns. When a match is found, an optional callback is invoked with parsed arguments.

Scripts are defined at build time using the `MODEM_CHAT_SCRIPT_DEFINE` and `MODEM_CHAT_SCRIPT_CMDS_DEFINE` macros. Each script entry pairs a request string with one or more `modem_chat_match` patterns. Match patterns support argument parsing by specifying a separator character (for example, `","`), making it easy to extract fields from responses such as `+CSQ: 20,99`.

In addition to scripted exchanges, the module continuously monitors the data stream for **unsolicited** responses — messages the modem sends without a preceding command (for example, `+CEREG:` network registration updates). These are handled by a separate set of match patterns registered when the chat instance is initialized.

The chat module attaches to any modem pipe, so it can communicate over a raw UART backend or over a CMUX DLCI channel interchangeably.

Modem Chat

Cellular Modem

The generic cellular modem driver, drivers/modem/modem_cellular.c, brings together all of the modules described above to implement a complete, hardware-agnostic cellular data connection. It exposes a standard Zephyr network interface so that applications can use the *BSD sockets API* without any modem-specific code.

Architecture overview The driver assembles the modem modules into a layered pipe architecture:

1. **UART backend** — A `modem_backend_uart` instance provides the lowest-level modem pipe. It abstracts the physical UART peripheral into the modem pipe API, performing thread-safe, interrupt-driven transfers.
2. **CMUX** — The `modem_cmux` instance attaches to the UART pipe and multiplexes it into two DLCI channels, following the 3GPP 27.010 specification.
3. **DLCI channel 1 (data)** — This pipe carries either AT command traffic during initialization, or PPP-framed IP data once the connection is established.
4. **DLCI channel 2 (commands)** — This pipe is dedicated to AT command traffic while the data connection is active, allowing the driver to query signal quality, registration status, and other parameters without interrupting the data flow.
5. **Modem chat** — A `modem_chat` instance executes AT command scripts. Before CMUX is started it attaches directly to the UART pipe; afterwards it moves to DLCI 1 for initialization scripts and then to DLCI 2 for periodic monitoring.
6. **Modem PPP** — A `modem_ppp` instance wraps Zephyr IP packets in PPP frames and sends them through DLCI 1. In the receive direction it strips the PPP framing and delivers packets to the network stack.

The simultaneous use of two DLCI channels is the key enabler: the modem can transfer IP data at full speed on DLCI 1 while the driver continues to exchange AT commands on DLCI 2 in the background.

Connection lifecycle The driver is built around an internal state machine that progresses through the following phases:

Power-on and hardware reset The driver pulses the modem reset and power GPIOs (if defined in the Device Tree) and waits for the modem to become responsive.

Initial AT configuration The modem chat module attaches directly to the UART pipe and runs the modem-specific `init_chat_script`. This script typically disables echo, queries the IMEI, model, firmware version, and configures unsolicited response reporting. All communication at this stage is plain AT over UART — CMUX is not yet active.

CMUX bring-up After the init script succeeds the driver sends the `AT+CMUX` command to switch the modem into multiplexed mode. The CMUX module attaches to the UART pipe and opens DLCI 1 and DLCI 2. From this point all communication is framed inside CMUX.

APN configuration and dialing The driver attaches the modem chat to DLCI 1 and runs a dynamically built APN script (`AT+CGDCONT`) followed by the dial script (`ATD*99#` or equivalent). On success the modem enters data mode on DLCI 1.

Data connection The modem PPP module attaches to the DLCI 1 pipe, and the modem chat moves to DLCI 2. The driver calls `net_if_carrier_on()` once a `+CEREG` (or `+CREG` / `+CGREG`) unsolicited response indicates that the modem has registered on the network.

At this point the Zephyr network stack negotiates a PPP session over DLCI 1 and the application can use sockets normally. Meanwhile, periodic chat scripts on DLCI 2 monitor signal quality and registration status.

Shutdown The driver runs a shutdown script and pulses the power-off GPIO to cleanly power down the modem. The state machine returns to the idle state.

Adding support for a new modem The generic cellular driver uses per-modem configuration supplied through Device Tree compatible bindings and associated chat scripts. Adding a new modem requires three artifacts:

1. A **Device Tree binding** that includes the common cellular modem base binding.
2. **Chat scripts** defining the AT command sequences for initialization, dialing, periodic monitoring, and (optionally) shutdown.
3. A **device instantiation macro** that ties the chat scripts and hardware configuration together.

The driver handles all pipe plumbing, CMUX management, PPP framing, and state machine logic automatically.

Supported modems The following modems are already supported in drivers/modem/modem_cellular.c:

- Fibocom LE250
- Quectel BG95, BG96
- Quectel EG25-G, EG800Q
- SIMCom SIM7080, A76xx
- u-blox SARA-R4, SARA-R5, LARA-R6
- Sierra Wireless HL7800
- Telit ME910G1, ME310G1, LE910C1 Thread-x, LEx10Q1
- Nordic Semiconductor nRF91 SLM
- Sequans GM02S

Each supported modem is defined using the same set of macros (`MODEM_CHAT_SCRIPT_CMDS_DEFINE`, `MODEM_CHAT_SCRIPT_DEFINE`, `MODEM_CELLULAR_DEFINE_INSTANCE`, etc.). Refer to drivers/modem/modem_cellular.c for complete examples of init, dial, periodic, and shutdown chat scripts. The common base Device Tree properties shared by all modems are documented in dts/bindings/modem/zephyr,cellular-modem-device.yaml.

Out-of-tree modem The driver macros and data structures are exported through include/zephyr/drivers/modem/modem_cellular.h, so a completely new modem can be defined **outside** the Zephyr tree — for example in application source — without modifying any upstream files. This is useful for fine-tuning AT command sequences for a specific use case, or for developing support for a new modem before submitting it upstream.

Start by creating a Device Tree binding that includes the common base binding, then write a driver source file using the macros from `modem_cellular.h`. The following three files are all that is needed.

Device Tree binding — `app/dts/bindings/my,modem.yaml`:

```
compatible: "my,modem"

include: zephyr,cellular-modem-device.yaml
```

Device Tree overlay — `app.overlay` (partial):

```
&uart30 {
    modem: modem {
        compatible = "my,modem";
        status = "okay";
        mdm-power-gpios = <&gpio1 13 GPIO_ACTIVE_HIGH>;
    };
};
```

Driver source — `app/src/my_modem.c`:

```
#include <zephyr/drivers/modem/modem_cellular.h>
#include <zephyr/device.h>

MODEM_CELLULAR_COMMON_CHAT_MATCHES();
MODEM_CHAT_MATCHES_DEFINE(my_modem_unsol,
    MODEM_CELLULAR_COMMON_UNSOL_MATCHES);

/* Init script - configure the modem, enable unsolicited LTE
 * registration notifications, then switch to CMUX.
 */
MODEM_CHAT_SCRIPT_CMDS_DEFINE(init_chat_script_cmds,
    MODEM_CHAT_SCRIPT_CMD_RESP("ATE0",              ok_match),
    MODEM_CHAT_SCRIPT_CMD_RESP("AT+CEREG=1",        ok_match),
    MODEM_CHAT_SCRIPT_CMD_RESP("AT+CMUX=0,0,5,127", ok_match));

MODEM_CHAT_SCRIPT_DEFINE(init_chat_script, init_chat_script_cmds,
    abort_matches, modem_cellular_chat_callback_handler, 1);

/* Dial script - enable the radio and open data mode */
MODEM_CHAT_SCRIPT_CMDS_DEFINE(dial_chat_script_cmds,
    MODEM_CHAT_SCRIPT_CMD_RESP("AT+CFUN=1", ok_match),
    MODEM_CHAT_SCRIPT_CMD_RESP("ATD*99#",   connect_match));

MODEM_CHAT_SCRIPT_DEFINE(dial_chat_script, dial_chat_script_cmds,
    dial_abort_matches, modem_cellular_chat_callback_handler, 60);

static const struct modem_cellular_vendor_config my_modem_vendor = {
    .scripts = {
        .init = &init_chat_script,
        .dial = &dial_chat_script,
    },
    .unsol_matches = {
        .matches = my_modem_unsol,
        .size = ARRAY_SIZE(my_modem_unsol),
    },
    .chat_delimiter = "\r",
    .chat_filter = "\n",
```

(continues on next page)

(continued from previous page)

```
    .power_pulse_duration_ms = 1000,
    .reset_pulse_duration_ms = 100,
    .startup_time_ms = 5000,
    .shutdown_time_ms = 5000,
};

/* Macro for defining a DT instance */
#define MY_MODEM_DEVICE(inst)                                                   \
    MODEM_DT_INST_PPP_DEFINE(inst,                                              \
        MODEM_CELLULAR_INST_NAME(ppp, inst), NULL, 1500, 64);                   \
                                                                                \
    static struct modem_cellular_data                                           \
        MODEM_CELLULAR_INST_NAME(data, inst);                                   \
                                                                                \
    MODEM_CELLULAR_DEFINE_AND_INIT_USER_PIPES(inst,                             \
        (user_pipe_0, 3), (user_pipe_1, 4))                                     \
                                                                                \
    MODEM_CELLULAR_DEFINE_INSTANCE(inst, &my_modem_vendor, NULL)

#define DT_DRV_COMPAT my_modem
DT_INST_FOREACH_STATUS_OKAY(MY_MODEM_DEVICE)
#undef DT_DRV_COMPAT
```

The example above is minimal but enough to start a PPP connection on a generic cellular modem. Refer to drivers/modem/modem_cellular.c for more complete examples.

Board-specific init script The init, network and dial scripts are vendor-scoped (keyed to the devicetree compatible), so configuration that differs between boards using the same modem, for example RF tuner band routing, would otherwise require forking the vendor driver. A board may instead register its own chat script with `MODEM_CELLULAR_BOARD_INIT_DEFINE`. The script runs over the AT control channel after CMUX is established and before APN and network configuration. Boards that register no script add no ROM or RAM footprint.

The driver overrides the script's completion callback with the one that advances the connect sequence, so any callback set on the script is ignored.

```
MODEM_CHAT_MATCH_DEFINE(board_init_ok_match, "OK", "", NULL);
MODEM_CHAT_SCRIPT_CMDS_DEFINE(board_init_cmds,
    MODEM_CHAT_SCRIPT_CMD_RESP("AT+QCFG=\"rf/tuner_cfg\",0,\"12,28\"",␣
↪board_init_ok_match));
MODEM_CHAT_SCRIPT_NO_ABORT_DEFINE(board_init, board_init_cmds, NULL, 10);

MODEM_CELLULAR_BOARD_INIT_DEFINE(DT_NODELABEL(modem), &board_init);
```

CMUX Power Saving

The 3GPP TS 27.010 specifies a power saving mechanism for CMUX which can be used in Zephyr when the modem supports it.

The power saving mechanism is covered in the following sections on the specification:

- 5.2.5 Inter-frame Fill
- 5.4.6.3.2 Power Saving Control (PSC) message
- 5.4.7 Power Control and Wake-up Mechanisms

The power saving mechanism allows runtime power management for the UART device used by the CMUX module. When there is no data to be sent or received on any DLCI channel, the CMUX

module will enter the idle state after a configurable timeout. In the idle state, the CMUX module will send a Power Saving Control message to the modem, requesting it to enter a low power state. The CMUX module may then close the pipe device, allowing the UART device to be powered down if runtime power management is enabled.

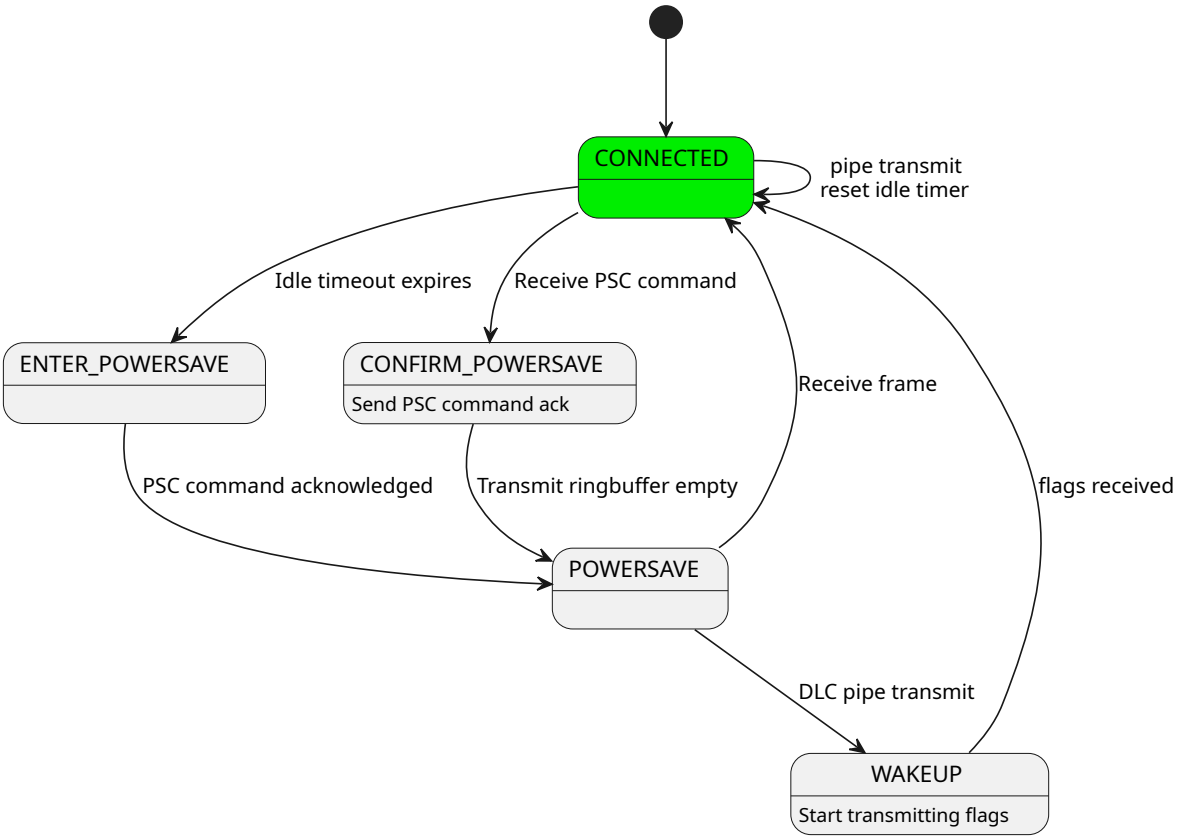
When data is to be sent or received on any DLCI channel, the CMUX module will exit the idle state and wakes the modem up by sending flag characters until it receives a flag character from the modem.

For modems that drive sleep and wake from a hardware line outside CMUX rather than via the in-band protocol, the `cmux-no-powersave-handshake` property opts out of the handshake on both halves. On entry CMUX skips the PSC frame exchange and transitions directly to power save. On exit CMUX skips the flag-character exchange and transitions directly to connected once the pipe is re-opened. The next outgoing frame re-synchronises framing on its own.

Some modems allow UART to be powered down only when the DTR (Data Terminal Ready) signal is de-asserted. In this case, a UART device with DTR support can be used with the CMUX module to control the DTR signal based on the power state of the UART.

Waking up on incoming data when UART is powered down requires a modem that supports RING signal to wake up the host. The RING signal is handled by the modem driver and it opens the pipe device when the RING signal is detected, allowing the CMUX module to wake up the modem and process incoming data.

The subsys/modem/modem_cmux.c module implements the power saving mechanism using the following state machine.



Within a connected state, `modem_cmux_process_received_byte()` is required to reply repeated flag characters as described in 5.2.5 Inter-frame Fill at the specification. Idle timer is kept running and cleared on every sent or received frame. Timer expiry will initiate transitioning to power saving modes.

Within the POWERSAVE state all DLC pipes remain open but the pipe towards UART is blocked or closed, so all data is buffered within CMUX ringbuffers to wait for waking up. Within this state,

repeated flag characters are also replied to allow remote end proceed with wake-up procedure as described in 5.4.7. If pipe is closed, it allows UART device to be powered down if runtime power management is enabled.

When idle timer expires on CONNECTED state, CMUX state machine blocks all DLC pipes and sends PSC command for remote end to initiate transitioning to POWERSAVE state. When PSC command is replied, CMUX transitions to POWERSAVE mode.

When within the CONNECTED state, the remote end may send a PSC command to initiate the transitioning to power saving mode. CMUX blocks all DLC pipes and sends a PSC response. When TX buffers are emptied, CMUX enters the POWERSAVE state.

When any of DLC pipes try to transmit data during POWERSAVE state, CMUX buffers it and moves to WAKEUP state that initiates wake-up procedure as specified in 5.4.7 by sending repeated stream of flag characters. Remote end replies the flag characters to indicate it is ready to receive data. CMUX then stops sending flag characters and moves back to CONNECTED state, resuming normal operation.

The CMUX power saving mechanism can be configured using the following Device Tree properties:

```
cmux-enable-runtime-power-save:
  type: boolean
  description: Enable runtime power saving using CMUX PSC commands.
               This requires modem to support CMUX and PSC commands while keeping␣
↪the data
               connection active.
cmux-close-pipe-on-power-save:
  type: boolean
  description: Close the modem pipe when entering power save mode.
              When runtime power management is enabled, this closes the UART.
              This requires modem to support waking up the UART using RING signal.
cmux-idle-timeout-ms:
  type: int
  description: Time in milliseconds after which CMUX will enter power save mode.
  default: 10000
```

Example Device Tree setup for CMUX with power saving:

```
&uart1 {
  status = "okay";
  zephyr,pm-device-runtime-auto;

  uart_dtr: uart-dtr {
    compatible = "zephyr,uart-dtr";
    dtr-gpios = <&interface_to_nrf9160 4 GPIO_ACTIVE_LOW>;
    status = "okay";
    zephyr,pm-device-runtime-auto;

    modem: modem {
      compatible = "nordic,nrf91-slm";
      status = "okay";
      mdm-ring-gpios = <&interface_to_nrf9160 5 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
      zephyr,pm-device-runtime-auto;
      cmux-enable-runtime-power-save;
      cmux-close-pipe-on-power-save;
      cmux-idle-timeout-ms = <5000>;
    };
  };
};
```

The above example shows a UART device with DTR support being used by a modem that supports CMUX and PSC commands. The DTR signal is used to control the power state of the UART. The RING signal from the modem is used to wake up the modem subsystem when it is powered down.

Input / Output

4.2 Input / Output

This section covers services for interacting with the outside world at runtime: reading from input devices, writing to the console, formatting output, and moving data efficiently between application code and hardware.

4.2.1 Console

Console API

Related code samples

- Console echoEcho an input character back to the output using the Console API.

4.2.2 Input

The input subsystem provides an API for dispatching input events from input devices to the application.

Input Events

The subsystem is built around the `input_event` structure. An input event represents a change in an individual event entity, for example the state of a single button, or a movement in a single axis.

The `input_event` structure describes the specific event, and includes a synchronization bit to indicate that the device reached a stable state, for example when the events corresponding to multiple axes of a multi-axis device have been reported.

Input Devices

An input device can report input events directly using `input_report()` or any related function; for example buttons or other on-off input entities would use `input_report_key()`.

Complex devices may use a combination of multiple events, and set the `sync` bit once the output is stable.

The `input_report*` functions take a `device` pointer, which is used to indicate which device reported the event and can be used by subscribers to only receive events from a specific device. If there's no actual device associated with the event, it can be set to `NULL`, in which case only subscribers with no device filter will receive the event.

Application API

An application can register a callback using the `INPUT_CALLBACK_DEFINE` macro. If a device node is specified, the callback is only invoked for events from the specific device, otherwise the callback will receive all the events in the system. This is the only type of filtering supported, any more complex filtering logic has to be implemented in the callback itself.

The subsystem can operate synchronously or by using an event queue, depending on the `CONFIG_INPUT_MODE` option. If the input thread is used, all the events are added to a queue and

executed in a common `input` thread. If the thread is not used, the callback are invoked directly in the input driver context.

The synchronous mode can be used in a simple application to keep a minimal footprint, or in a complex application with an existing event model, where the callback is just a wrapper to pipe back the event in a more complex application specific event system.

HID code mapping

A common use case for input devices is to use them to generate HID reports. For this purpose, the `input_to_hid_code()` and `input_to_hid_modifier()` functions can be used to map input codes to HID codes and modifiers.

General Purpose Drivers

- `adc-keys`: for buttons connected to a resistor ladder.
- `analog-axis`: for absolute position devices connected to an ADC input (thumbsticks, sliders...).
- `gpio-kbd-matrix`: for GPIO-connected keyboard matrices.
- `gpio-keys`: for switches directly connected to a GPIO, implements button debouncing.
- `gpio-qdec`: for GPIO-connected quadrature encoders.
- `input-keymap`: maps row/col/touch events from a keyboard matrix to key events.
- `zephyr,input-longpress`: listens for key events, emits events for short and long press.
- `zephyr,input-double-tap`: listens for key events, emits events for input double taps and, optionally, single taps
- `zephyr,lvgl-button-input` `zephyr,lvgl-encoder-input` `zephyr,lvgl-keypad-input` `zephyr,lvgl-pointer-input`: listens for input events and translates those to various types of LVGL input devices.

Detailed Driver Documentation

GPIO Keyboard Matrix The `gpio-kbd-matrix` driver supports a large variety of keyboard matrix hardware configurations and has numerous options to change its behavior. This is an overview of some common setups and how they can be supported by the driver.

The conventional configuration for all of these is that the driver reads on the row GPIOs (inputs) and selects on the columns GPIOs (output).

Base use case, no isolation diodes, interrupt capable GPIOs This is the common configuration found on consumer keyboards with membrane switches and flexible circuit boards, no isolation diodes, requires ghosting detection (which is enabled by default).

The system must support GPIO interrupts, and the interrupt can be enabled on all row GPIOs at the same time.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     row-gpios = <&gpio0 0 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 1 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 2 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
     col-gpios = <&gpio0 3 GPIO_ACTIVE_LOW>,
                 <&gpio0 4 GPIO_ACTIVE_LOW>,
                 <&gpio0 5 GPIO_ACTIVE_LOW>;
};
```

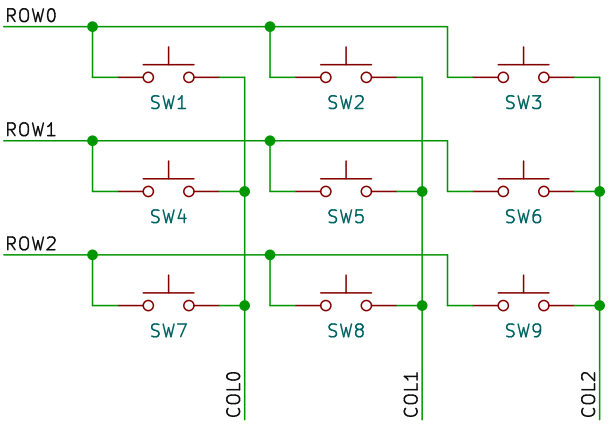


Fig. 23: A 3x3 matrix, no diodes

In this configuration the matrix scanning library enters idle mode once all keys are released, and the keyboard matrix thread only wakes up when a key has been pressed.

GPIOs for columns that are not currently selected are configured in high impedance mode. This means that the row state may need some time to settle to avoid misreading the key state from a column to the following one. The settle time can be tweaked by changing the `settle-time-us` property.

Isolation diodes If the matrix has isolation diodes for every key, then it's possible to:

- disable ghosting detection, allowing any key combination to be detected
- configuring the driver to drive unselected columns GPIO to inactive state rather than high impedance, this allows to reduce the settle time (potentially down to 0), and use the more efficient port wide GPIO read APIs (happens automatically if the GPIO pins are sequential)

Matrixes with diodes going from rows to columns must use pull-ups on rows and active low columns.

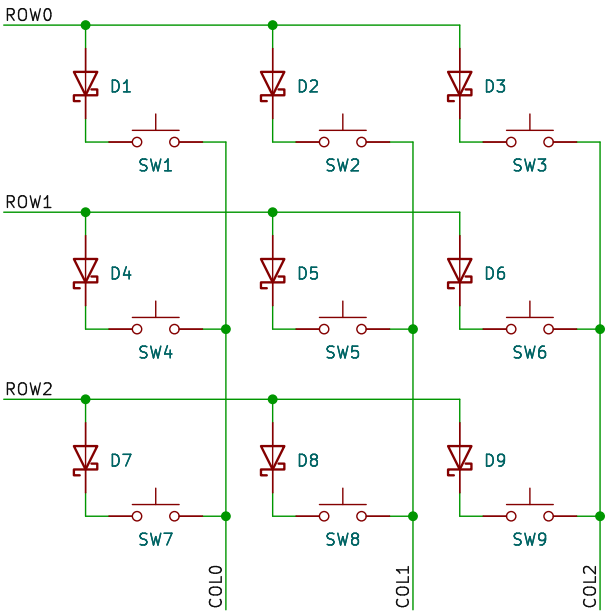


Fig. 24: A 3x3 matrix with row to column isolation diodes.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     row-gpios = <&gpio0 0 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 1 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>,
                 <&gpio0 2 (GPIO_PULL_UP | GPIO_ACTIVE_LOW)>;
     col-gpios = <&gpio0 3 GPIO_ACTIVE_LOW>,
                 <&gpio0 4 GPIO_ACTIVE_LOW>,
                 <&gpio0 5 GPIO_ACTIVE_LOW>;
     col-drive-inactive;
     settle-time-us = <0>;
     no-ghostkey-check;
};
```

Matrixes with diodes going from columns to rows must use pull-downs on rows and active high columns.

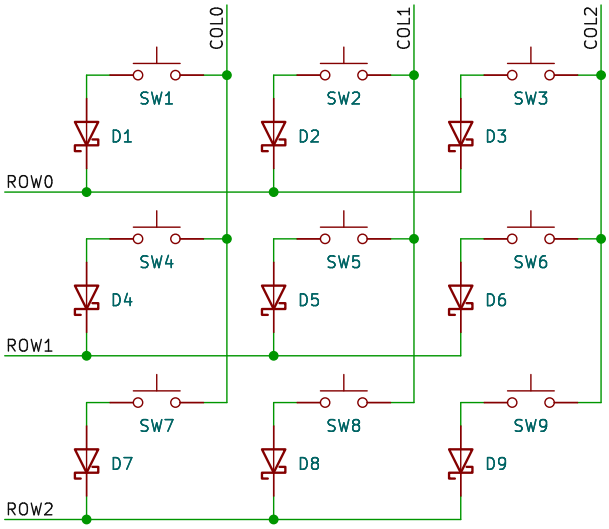


Fig. 25: A 3x3 matrix with column to row isolation diodes.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     row-gpios = <&gpio0 0 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>,
                 <&gpio0 1 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>,
                 <&gpio0 2 (GPIO_PULL_DOWN | GPIO_ACTIVE_HIGH)>;
     col-gpios = <&gpio0 3 GPIO_ACTIVE_HIGH>,
                 <&gpio0 4 GPIO_ACTIVE_HIGH>,
                 <&gpio0 5 GPIO_ACTIVE_HIGH>;
     col-drive-inactive;
     settle-time-us = <0>;
     no-ghostkey-check;
};
```

GPIO with no interrupt support Some GPIO controllers have limitations on GPIO interrupts, and may not support enabling interrupts on all row GPIOs at the same time.

In this case, the driver can be configured to not use interrupt at all, and instead idle by selecting all columns and keep polling on the row GPIOs, which is a single GPIO API operation if the pins are sequential.

This configuration can be enabled by setting the `idle-mode` property to `poll`:

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     ...
     idle-mode = "poll";
};
```

GPIO multiplexer In more extreme cases, such as if the columns are using a multiplexer and it's impossible to select all of them at the same time, the driver can be configured to scan continuously.

This can be done by setting `idle-mode` to `scan` and `poll-timeout-ms` to `0`.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     ...
     poll-timeout-ms = <0>;
     idle-mode = "scan";
};
```

Row and column GPIO selection If the row GPIOs are sequential and on the same gpio controller, the driver automatically switches API to read from the whole GPIO port rather than the individual pins. This is particularly useful if the GPIOs are not memory mapped, for example on an I2C or SPI port expander, as this significantly reduces the number of transactions on the corresponding bus.

The same is true for column GPIOs, but only if the matrix is configured for `col-drive-inactive`, so that is only usable for matrixes with isolation diodes.

16-bit row support The driver uses an 8-bit datatype to store the row state by default, which limits the matrix row size to 8. This can be increased to 16 by enabling the `CONFIG_INPUT_KBD_MATRIX_16_BIT_ROW` option.

Actual key mask configuration If the key matrix is not complete, a map of the keys that are actually populated can be specified using the `actual-key-mask` property. This allows the matrix state to be filtered to remove keys that are not present before ghosting detection, potentially allowing key combinations that would otherwise be blocked by it.

For example for a 3x3 matrix missing a key:

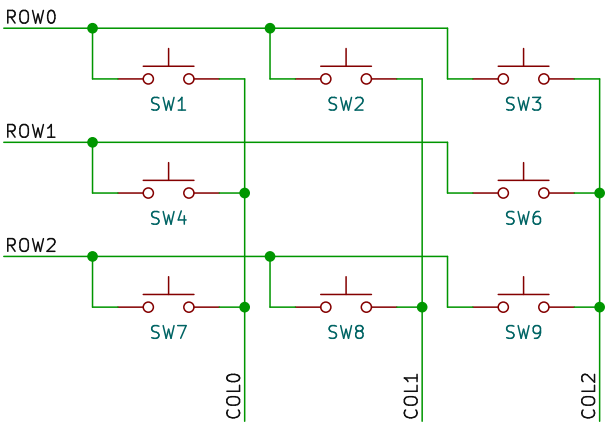


Fig. 26: A 3x3 matrix missing a key.

```
kbd-matrix {
     compatible = "gpio-kbd-matrix";
     ...
     actual-key-mask = <0x07 0x05 0x07>;
};
```

This would allow, for example, to detect pressing `Sw1`, `SW2` and `SW4` at the same time without triggering anti ghosting.

The actual key mask can be changed at runtime by enabling `CONFIG_INPUT_KBD_ACTUAL_KEY_MASK_DYNAMIC` and the using the `input_kbd_matrix_actual_key_mask_set()` API.

Keymap configuration Keyboard matrix devices report a series of x/y/touch events. These can be mapped to normal key events using the `input-keymap` driver.

For example, the following would setup a `keymap` device that take the x/y/touch events as an input and generate corresponding key events as an output:

```
kbd {
    ...
    keymap {
        compatible = "input-keymap";
        keymap = <
            MATRIX_KEY(0, 0, INPUT_KEY_1)
            MATRIX_KEY(0, 1, INPUT_KEY_2)
            MATRIX_KEY(0, 2, INPUT_KEY_3)
            MATRIX_KEY(1, 0, INPUT_KEY_4)
            MATRIX_KEY(1, 1, INPUT_KEY_5)
            MATRIX_KEY(1, 2, INPUT_KEY_6)
            MATRIX_KEY(2, 0, INPUT_KEY_7)
            MATRIX_KEY(2, 1, INPUT_KEY_8)
            MATRIX_KEY(2, 2, INPUT_KEY_9)
        >;
        row-size = <3>;
        col-size = <3>;
    };
};
```

Keymap utilities

Keyboard matrix shell commands The shell command `kbd_matrix_state_dump` can be used to test the functionality of any keyboard matrix driver implemented using the keyboard matrix library. Once enabled it logs the state of the matrix every time it changes, and once disabled it prints an or-mask of any key that has been detected, which can be used to set the `actual-key-mask` property.

The command can be enabled using the `CONFIG_INPUT_SHELL_KBD_MATRIX_STATE`.

Example usage:

```
uart:~$ device list
devices:
- kbd-matrix (READY)
uart:~$ input kbd_matrix_state_dump kbd-matrix
Keyboard state logging enabled for kbd-matrix
[00:01:41.678,466] <inf> input: kbd-matrix state [01 -- -- --] (1)
[00:01:41.784,912] <inf> input: kbd-matrix state [-- -- -- --] (0)
```

(continues on next page)

(continued from previous page)

```
...
press more buttons
...
uart:~$ input kbd_matrix_state_dump off
Keyboard state logging disabled
[00:01:47.967,651] <inf> input: kbd-matrix key-mask [07 05 07 --] (8)
```

Keyboard matrix library The GPIO keyboard matrix driver is based on a generic keyboard matrix library, which implements the core functionalities such as scanning delays, debouncing, idle mode etc. This can be reused to implement other keyboard matrix drivers, potentially application specific.

Keyboard Matrix

API Reference

Input

Related code samples

- ButtonHandle GPIO button inputs using the input subsystem.
- HT16K33 LED driver with keyscanControl up to 128 LEDs connected to an HT16K33 LED driver and log keyscan events.
- Legacy USB HID mouseImplement a basic HID mouse device.
- LVGL basic sampleDisplay a "Hello World" and react to user input using LVGL.
- RTK0EG0019B01002BJ Capacitive Touch Application ShieldInteract with the Capacitive Touch Sensor and LED matrix on the RTK0EG0019B01002BJ shield
- USB HID keyboardImplement a basic HID keyboard device.
- USB HID mouseImplement a basic HID mouse device.

Input Event Definitions

Input Event Definitions

Related code samples

- Draw touch eventsVisualize touch events on a display.
- Input dumpPrint all input events.
- USB MIDI2 deviceImplements a simple USB MIDI loopback and keyboard device.

Analog Axis API Reference

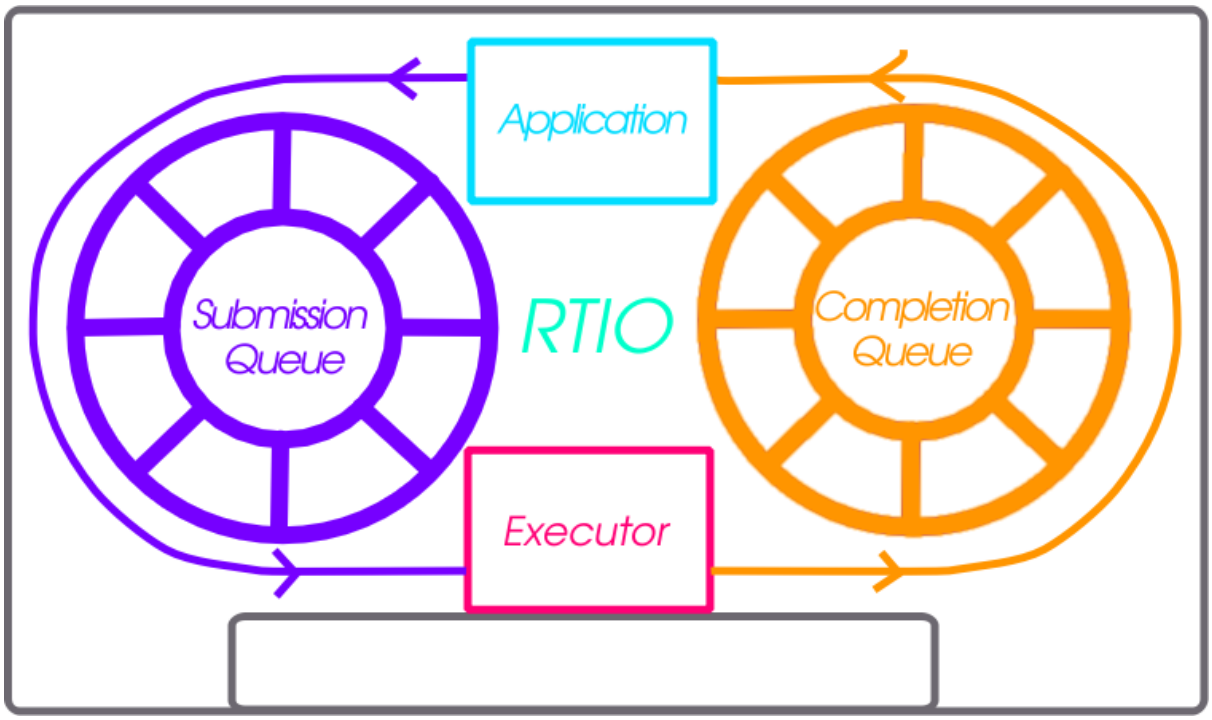
Analog axis

Touchscreen API Reference

Touchscreen Events

4.2.3 Real Time I/O (RTIO)

- *Problem*
- *Inspiration, introducing io_uring*
- *Submission Queue*
- *Completion Queue*
- *Executor*
- *IO Device*
- *Cancellation*
- *Memory pools*
- *When to Use*
- *Supported Buses*
- *API Reference*



RTIO provides a framework for doing asynchronous operation chains with event driven I/O. This section covers the RTIO API, queues, executor, iodev, and common usage patterns with peripheral devices.

RTIO takes a lot of inspiration from Linux's io_uring in its operations and API as that API matches up well with hardware transfer queues and descriptions such as DMA transfer lists.

Problem

An application wishing to do complex DMA or interrupt driven operations today in Zephyr requires direct knowledge of the hardware and how it works. There is no understanding in the DMA API of other Zephyr devices and how they relate.

This means doing complex audio, video, or sensor streaming requires direct hardware knowledge or leaky abstractions over DMA controllers. Neither is ideal.

To enable asynchronous operations, especially with DMA, a description of what to do rather than direct operations through C and callbacks is needed. Enabling DMA features such as channels with priority, and sequences of transfers requires more than a simple list of descriptions.

Using DMA and/or interrupt driven I/O shouldn't dictate whether or not the call is blocking or not.

Inspiration, introducing io_uring

It's better not to reinvent the wheel (or ring in this case) and io_uring as an API from the Linux kernel provides a winning model. In io_uring there are two lock-free ring buffers acting as queues shared between the kernel and a userland application. One queue for submission entries which may be chained and flushed to create concurrent sequential requests. A second queue for completion queue events. Only a single syscall is actually required to execute many operations, the io_uring_submit call. This call may block the caller when a number of operations to wait on is given.

This model maps well to DMA and interrupt driven transfers. A request to do a sequence of operations in an asynchronous way directly relates to the way hardware typically works with interrupt driven state machines potentially involving multiple peripheral IPs like bus and DMA controllers.

Submission Queue

The submission queue (sq), is the description of the operations to perform in concurrent chains.

For example imagine a typical SPI transfer where you wish to write a register address to then read from. So the sequence of operations might be...

1. Chip Select
2. Clock Enable
3. Write register address into SPI transmit register
4. Read from the SPI receive register into a buffer
5. Disable clock
6. Disable Chip Select

If anything in this chain of operations fails give up. Some of those operations can be embodied in a device abstraction that understands a read or write implicitly means setup the clock and chip select. The transactional nature of the request also needs to be embodied in some manner. Of the operations above perhaps the read could be done using DMA as its large enough make sense. That requires an understanding of how to setup the device's particular DMA to do so.

The above sequence of operations is embodied in RTIO as chain of submission queue entries (sqe). Chaining is done by setting a bitflag in an sqe to signify the next sqe must wait on the current one.

Because the chip select and clocking is common to a particular SPI controller and device on the bus it is embodied in what RTIO calls an iodev.

Multiple operations against the same iodev are done in the order provided as soon as possible. If two operation chains have varying points using the same device its possible one chain will have to wait for another to complete.

Completion Queue

In order to know when a sqe has completed there is a completion queue (cq) with completion queue events (cqe). A sqe once completed results in a cqe being pushed into the cq. The ordering of cqe may not be the same order of sqe. A chain of sqe will however ensure ordering and failure cascading.

Other potential schemes are possible but a completion queue is a well trod idea with io_uring and other similar operating system APIs.

Executor

The RTIO executor is a low overhead concurrent I/O task scheduler. It ensures certain request flags provide the expected behavior. It takes a list of submissions working through them in order. Various flags allow for changing the behavior of how submissions are worked through. Flags to form in order chains of submissions, transactional sets of submissions, or create multi-shot (continuously producing) requests are all possible!

IO Device

Turning submission queue entries (sqe) into completion queue events (cqe) is the job of objects implementing the iodev (IO device) API. This API accepts requests in the form of the iodev submit API call. It is the io devices job to work through its internal queue of submissions and convert them into completions. In effect every io device can be viewed as an independent, event driven actor like object, that accepts a never ending queue of I/O like requests. How the iodev does this work is up to the author of the iodev, perhaps the entire queue of operations can be converted to a set of DMA transfer descriptors, meaning the hardware does almost all of the real work.

Cancellation

Canceling an already queued operation is possible but not guaranteed. If the SQE has not yet started, it's likely that a call to `rtio_sqe_cancel()` will remove the SQE and never run it. If, however, the SQE already started running, the cancel request will be ignored.

Memory pools

In some cases requests to read may not know how much data will be produced. Alternatively, a reader might be handling data from multiple io devices where the frequency of the data is unpredictable. In these cases it may be wasteful to bind memory to in flight read requests. Instead with memory pools the memory to read into is left to the iodev to allocate from a memory pool associated with the RTIO context that the read was associated with. To create such an RTIO context the `RTIO_DEFINE_WITH_MEMPOOL` can be used. It allows creating an RTIO context with a dedicated pool of "memory blocks" which can be consumed by the iodev. Below is a snippet setting up the RTIO context with a memory pool. The memory pool has 128 blocks, each block has the size of 16 bytes, and the data is 4 byte aligned.

```
#include <zephyr/rtio/rtio.h>

#define SQ_SIZE       4
#define CQ_SIZE       4
#define MEM_BLK_COUNT 128
#define MEM_BLK_SIZE  16
#define MEM_BLK_ALIGN 4

RTIO_DEFINE_WITH_MEMPOOL(rtio_context,
    SQ_SIZE, CQ_SIZE, MEM_BLK_COUNT, MEM_BLK_SIZE, MEM_BLK_ALIGN);
```

When a read is needed, the caller simply needs to replace the call `rtio_sqe_prep_read()` (which takes a pointer to a buffer and a length) with a call to `rtio_sqe_prep_read_with_pool()`. The iodev requires only a small change which works with both pre-allocated data buffers as well as the mempool. When the read is ready, instead of getting the buffers directly from the `rtio_iodev_sqe`, the iodev should get the buffer and count by calling `rtio_sqe_rx_buf()` like so:

```
uint8_t *buf;
uint32_t buf_len;
int rc = rtio_sqe_rx_buff(iodev_sqe, MIN_BUF_LEN, DESIRED_BUF_LEN, &buf, &buf_len);

if (rc != 0) {
  LOG_ERR("Failed to get buffer of at least %u bytes", MIN_BUF_LEN);
  return;
}
```

Finally, the consumer will be able to access the allocated buffer via `rtio_cqe_get_mempool_buffer()`.

```
uint8_t *buf;
uint32_t buf_len;
int rc = rtio_cqe_get_mempool_buffer(&rtio_context, &cqe, &buf, &buf_len);

if (rc != 0) {
  LOG_ERR("Failed to get mempool buffer");
  return rc;
}

/* Release the cqe events (note that the buffer is not released yet */
rtio_cqe_release_all(&rtio_context);

/* Do something with the memory */

/* Release the mempool buffer */
rtio_release_buffer(&rtio_context, buf);
```

When to Use

RTIO is useful in cases where concurrent or batch like I/O flows are useful.

From the driver/hardware perspective the API enables batching of I/O requests, potentially in an optimal way. Many requests to the same SPI peripheral for example might be translated to hardware command queues or DMA transfer descriptors entirely. Meaning the hardware can potentially do more than ever.

There is a small cost to each RTIO context and iodev. This cost could be weighed against using a thread for each concurrent I/O operation or custom queues and threads per peripheral. RTIO is much lower cost than that.

Supported Buses

To check if your bus supports RTIO natively, you can check the driver API implementation, if the driver implements the `iodev_submit` function of the bus API, then RTIO is supported. If the driver doesn't support the RTIO APIs, it will set the submit function to `i2c_iodev_submit_fallback`.

I2C buses have a default implementation which allows apps to leverage the RTIO work queue while vendors implement the submit function. With this queue, any I2C bus driver that does not implement the `iodev_submit` function will defer to a work item which will perform a blocking I2C transaction. To change the pool size, set a different value to `CONFIG_RTIO_WORKQ_POOL_ITEMS`.

API Reference

RTIO

Related code samples

- I2C RTIO loopbackPerform I2C transfers between I2C controller and custom I2C target using RTIO.
- Producer ConsumerImplement a producer->consumer pipe using RTIO.
- Sensor batch processingImplement a sensor device using RTIO for asynchronous data processing.
- SPI RTIO loopbackPerform SPI transfers between SPI controller and SPI target using RTIO.

4.2.4 Formatted Output

Applications as well as Zephyr itself requires infrastructure to format values for user consumption. The standard C99 library `*printf()` functionality fulfills this need for streaming output devices or memory buffers, but in an embedded system devices may not accept streamed data and memory may not be available to store the formatted output.

Internal Zephyr API traditionally provided this both for `printk()` and for Zephyr's internal minimal libc, but with separate internal interfaces. Logging, tracing, shell, and other applications made use of either these APIs or standard libc routines based on build options.

The `cbprintf()` public APIs convert C99 format strings and arguments, providing output produced one character at a time through a callback mechanism, replacing the original internal functions and providing support for almost all C99 format specifications. Existing use of `s*printf()` C libraries in Zephyr can be converted to `snprintfcb()` to avoid pulling in libc implementations.

Several Kconfig options control the set of features that are enabled, allowing some control over features and memory usage:

- `CONFIG_CBPRINTF_FULL_INTEGRAL` or `CONFIG_CBPRINTF_REDUCED_INTEGRAL`
- `CONFIG_CBPRINTF_FP_SUPPORT`
- `CONFIG_CBPRINTF_FP_A_SUPPORT`
- `CONFIG_CBPRINTF_FP_ALWAYS_A`
- `CONFIG_CBPRINTF_N_SPECIFIER`

`CONFIG_CBPRINTF_LIBC_SUBSTS` can be used to provide functions that behave like standard libc functions but use the selected cbprintf formatter rather than pulling in another formatter from libc.

In addition `CONFIG_CBPRINTF_NANO` can be used to revert back to the very space-optimized but limited formatter used for `printk()` before this capability was added.

Cbprintf Packaging

Typically, strings are formatted synchronously when a function from `printf` family is called. However, there are cases when it is beneficial that formatting is deferred. In that case, a state (format string and arguments) must be captured. Such state forms a self-contained package which contains format string and arguments. Additionally, package may contain copies of strings which are part of a format string (format string or any %s argument). Package primary content resembles va_list stack frame thus standard formatting functions are used to process a package. Since package contains data which is processed as va_list frame, strict alignment must be maintained. Due to required padding, size of the package depends on alignment. When package is copied, it should be copied to a memory block with the same alignment as origin.

Package can have following variants:

- **Self-contained** - non read-only strings appended to the package. String can be formatted from such package as long as there is access to read-only string locations. Package may contain information where read-only strings are located within the package. That information can be used to convert packet to fully self-contained package.
- **Fully self-contained** - all strings are appended to the package. String can be formatted from such package without any external data.
- **Transient**- only arguments are stored. Package contain information where pointers to non read-only strings are located within the package. Optionally, it may contain read-only string location information. String can be formatted from such package as long as non read-only strings are still valid and read-only strings are accessible. Alternatively, package can be converted to **self-contained** package or **fully self-contained** if information about read-only string locations is present in the package.

Package can be created using two methods:

- runtime - using `cbprintf_package()` or `cbvprintf_package()`. This method scans format string and based on detected format specifiers builds the package.
- static - types of arguments are detected at compile time by the preprocessor and package is created as simple assignments to a provided memory. This method is significantly faster than runtime (more than 15 times) but has a significant limitation: it cannot distinguish between `%p` and `%s` if char pointer is used. It treats all (unsigned) char pointers as `%s` thus it will attempt to append string to a package. It can be handled correctly during conversion from **transient** package to **self-contained** package using `CBPRINTF_PACKAGE_CONVERT_PTR_CHECK` flag. However, it requires access to the format string and it is not always possible thus it is recommended to cast char pointers used for `%p` to `void *`. There is a logging warning generated by `cbprintf_package_convert()` called with `CBPRINTF_PACKAGE_CONVERT_PTR_CHECK` flag when char pointer is used with `%p`.

Several Kconfig options control behavior of the packaging:

- `CONFIG_CBPRINTF_PACKAGE_LONGDOUBLE`
- `CONFIG_CBPRINTF_STATIC_PACKAGE_CHECK_ALIGNMENT`

Cbprintf package conversion It is possible to convert package to a variant which contains more information, e.g **transient** package can be converted to **self-contained**. Conversion to **fully self-contained** package is possible if `CBPRINTF_PACKAGE_ADD_RO_STR_POS` flag was used when package was created.

`cbprintf_package_copy()` is used to calculate space needed for the new package and to copy and convert a package.

Cbprintf package format Format of the package contains paddings which are platform specific. Package consists of header which contains size of package (excluding appended strings) and number of appended strings. It is followed by the arguments which contains alignment paddings and resembles *va_list* stack frame. It is followed by data associated with character pointer arguments used by the string which are not appended to the string (but may be appended later by `cbprinf_package_convert()`). Finally, package, optionally, contains appended strings. Each string contains 1 byte header which contains index of the location where address argument is stored. During packaging address is set to null and before string formatting it is updated to point to the current string location within the package. Updating address argument must happen just before string formatting since address changes whenever package is copied.

Header sizeof(void *)	1 byte: Argument list size including header and *fmt* (in 32 bit words)
	1 byte: Number of strings appended to the package
	1 byte: Number of read-only string argument locations
	1 byte: Number of transient string argument locations
	platform specific padding to sizeof(void *)
Arguments	Pointer to *fmt* (or null if *fmt* is appended to the package)
	(optional padding for platform specific alignment)
	argument 0
	(optional padding for platform specific alignment)
	argument 1
	...
String location information (optional)	Indexes of words within the package where read-only strings are located
	Pairs of argument index and argument location index where transient strings are located
Appended strings (optional)	1 byte: Index within the package to the location of associated argument
	Null terminated string
	...

Warning

If `CONFIG_MINIMAL_LIBC` is selected in combination with `CONFIG_CBPRINTF_NANO` formatting with C standard library functions like `printf` or `snprintf` is limited. Among other things the `%n` specifier, most format flags, precision control, and floating point are not supported.

Limitations and recommendations

- It is recommended to cast any character pointer used with `%p` format specifier to other pointer type (e.g. `void *`). If format string is not accessible then only static packaging is possible and it will append all detected strings. Character pointer used for `%p` will be considered as string pointer. Copying from unexpected location can have serious consequences (e.g., memory fault or security violation).

API Reference

Formatted Output APIs

Inter-Process Communication

4.3 Inter-Process Communication

This section covers mechanisms for exchanging data and events between threads, between cores, and between virtualized environments running on the same system.

4.3.1 Interprocessor Communication (IPC)

IPC service

- *Overview*
- *Simple data exchange*
- *Data exchange using the no-copy API*
- *Backends*
- *API Reference*
  - *IPC service API*
  - *IPC service backend API*

The IPC service API provides an interface to exchange data between two domains or CPUs.

Overview An IPC service communication channel consists of one instance and one or several endpoints associated with the instance.

An instance is the external representation of a physical communication channel between two domains or CPUs. The actual implementation and internal representation of the instance is peculiar to each backend.

An individual instance is not used to send data between domains/CPUs. To send and receive the data, the user must create (register) an endpoint in the instance. This allows for the connection of the two domains of interest.

It is possible to have zero or multiple endpoints for one single instance, possibly with different priorities, and to use each to exchange data. Endpoint prioritization and multi-instance ability highly depend on the backend used.

The endpoint is an entity the user must use to send and receive data between two domains (connected by the instance). An endpoint is always associated to an instance.

The creation of the instances is left to the backend, usually at init time. The registration of the endpoints is left to the user, usually at run time.

The API does not mandate a way for the backend to create instances but it is strongly recommended to use the devicetree to retrieve the configuration parameters for an instance. Currently, each backend defines its own DT-compatible configuration that is used to configure the interface at boot time.

The following usage scenarios are supported:

- Simple data exchange.
- Data exchange using the no-copy API.

Simple data exchange To send data between domains or CPUs, an endpoint must be registered onto an instance.

See the following example:

> **Note**
>
> Before registering an endpoint, the instance must be opened using the `ipc_service_open_instance()` function.

```
#include <zephyr/ipc/ipc_service.h>

static void bound_cb(void *priv)
```

(continues on next page)

(continued from previous page)

```
{
   /* Endpoint bounded */
}

static void recv_cb(const void *data, size_t len, void *priv)
{
   /* Data received */
}

static struct ipc_ept_cfg ept0_cfg = {
   .name = "ept0",
   .cb = {
      .bound    = bound_cb,
      .received = recv_cb,
   },
};

int main(void)
{
   const struct device *inst0;
   struct ipc_ept ept0;
   int ret;

   inst0 = DEVICE_DT_GET(DT_NODELABEL(ipc0));
   ret = ipc_service_open_instance(inst0);
   ret = ipc_service_register_endpoint(inst0, &ept0, &ept0_cfg);

   /* Wait for endpoint bound (bound_cb called) */

   unsigned char message[] = "hello world";
   ret = ipc_service_send(&ept0, &message, sizeof(message));
}
```

Data exchange using the no-copy API If the backend supports the no-copy API you can use it to directly write and read to and from shared memory regions.

See the following example:

```
#include <zephyr/ipc/ipc_service.h>
#include <stdint.h>
#include <string.h>

static struct ipc_ept ept0;

static void bound_cb(void *priv)
{
   /* Endpoint bounded */
}

static void recv_cb_nocopy(const void *data, size_t len, void *priv)
{
   int ret;

   ret = ipc_service_hold_rx_buffer(&ept0, (void *)data);
   /* Process directly or put the buffer somewhere else and release. */
   ret = ipc_service_release_rx_buffer(&ept0, (void *)data);
```

(continues on next page)

(continued from previous page)

```
}

static struct ipc_ept_cfg ept0_cfg = {
   .name = "ept0",
   .cb = {
      .bound    = bound_cb,
      .received = recv_cb,
   },
};

int main(void)
{
   const struct device *inst0;
   int ret;

   inst0 = DEVICE_DT_GET(DT_NODELABEL(ipc0));
   ret = ipc_service_open_instance(inst0);
   ret = ipc_service_register_endpoint(inst0, &ept0, &ept0_cfg);

   /* Wait for endpoint bound (bound_cb called) */
   void *data;
   unsigned char message[] = "hello world";
   uint32_t len = sizeof(message);

   ret = ipc_service_get_tx_buffer(&ept0, &data, &len, K_FOREVER);

   memcpy(data, message, len);

   ret = ipc_service_send_nocopy(&ept0, data, sizeof(message));
}
```

Backends The requirements needed for implementing backends give flexibility to the IPC service. These allow for the addition of dedicated backends having only a subsets of features for specific use cases.

The backend must support at least the following:

- The init-time creation of instances.
- The run-time registration of an endpoint in an instance.

Additionally, the backend can also support the following:

- The run-time deregistration of an endpoint from the instance.
- The run-time closing of an instance.
- The no-copy API.

Each backend can have its own limitations and features that make the backend unique and dedicated to a specific use case. The IPC service API can be used with multiple backends simultaneously, combining the pros and cons of each backend.

ICMsg backend The inter core messaging backend (ICMsg) is a lighter alternative to the heavier RPMsg static vrings backend. It offers a minimal feature set in a small memory footprint. The ICMsg backend is build on top of *Single Producer Single Consumer Packet Buffer*.

Overview The ICMsg backend uses shared memory and MBOX devices for exchanging data. Shared memory is used to store the data, MBOX devices are used to signal that the data has been

written.

The backend supports the registration of a single endpoint on a single instance. If the application requires more than one communication channel, you must define multiple instances, each having its own dedicated endpoint.

Configuration The backend is configured via Kconfig and devicetree. When configuring the backend, do the following:

- If at least one of the cores uses data cache on shared memory, set the `dcache-alignment` value. This must be the largest value of the invalidation or the write-back size for both sides of the communication. You can skip it if none of the communication sides is using data cache on shared memory.
- Define two memory regions and assign them to `tx-region` and `rx-region` of an instance. Ensure that the memory regions used for data exchange are unique (not overlapping any other region) and accessible by both domains (or CPUs).
- Define MBOX devices which are used to send the signal that informs the other domain (or CPU) that data has been written. Ensure that the other domain (or CPU) is able to receive the signal.

Caution

Make sure that you set correct value of the `dcache-alignment`. At first, wrong value may not show any signs, which may give a false impression that everything works. Unstable behavior will appear sooner or later.

See the following configuration example for one of the instances:

```
reserved-memory {
   tx: memory@20070000 {
      reg = <0x20070000 0x0800>;
   };

   rx: memory@20078000 {
      reg = <0x20078000 0x0800>;
   };
};

   ipc {
      ipc0: ipc0 {
         compatible = "zephyr,ipc-icmsg";
         dcache-alignment = <32>;
         tx-region = <&tx>;
         rx-region = <&rx>;
         mboxes = <&mbox 0>, <&mbox 1>;
         mbox-names = "tx", "rx";
         status = "okay";
      };
   };
};
```

You must provide a similar configuration for the other side of the communication (domain or CPU) but you must swap the MBOX channels and memory regions (`tx-region` and `rx-region`).

Bonding When the endpoint is registered, the following happens on each domain (or CPU) connected through the IPC instance:

1. The domain (or CPU) writes a magic number to its `tx-region` of the shared memory.
2. It then sends a signal to the other domain or CPU, informing that the data has been written. Sending the signal to the other domain or CPU is repeated with timeout.
3. When the signal from the other domain or CPU is received, the magic number is read from `rx-region`. If it is correct, the bonding process is finished and the backend informs the application by calling `ipc_service_cb.bound` callback.

Samples

- ipc-icmsg

Detailed Protocol Specification The ICMsg uses two shared memory regions and two MBOX channels. The region and channel pair are used to transfer messages in one direction. The other pair is symmetric and transfers messages in the opposite direction. For this reason, the specification below focuses on one such pair. The other pair is identical.

The ICMsg provides just one endpoint per instance.

Shared Memory Region Organization If data caching is enabled, the shared memory region provided to ICMsg must be aligned according to the cache requirement. If cache is not enabled, the required alignment is 4 bytes.

The shared memory region is entirely used by a single FIFO. It contains read and write indexes followed by the data buffer. The detailed structure is contained in the following table:

Field name	Size (bytes)	Byte order	Description
`rd_idx`	4	little-endian	Index of the first incoming byte in the `data` field.
`padding`	depends on cache alignment	n/a	Padding added to align `wr_idx` to the cache alignment.
`wr_idx`	4	little-endian	Index of the byte after the last incoming byte in the `data` field.
`data`	everything to the end of the region	n/a	Circular buffer containing actual bytes to transfer.

This is usual FIFO with a circular buffer:

- The Indexes (`rd_idx` and `wr_idx`) are wrapped around when they reach the end of the `data` buffer.
- The FIFO is empty if `rd_idx == wr_idx`.
- The FIFO has one byte less capacity than the `data` buffer length.

Packets Packets are sent over the FIFO described in the above section. One packet can be wrapped around if it occurs at the end of the FIFO buffer.

The following is the packet structure:

Field name	Size (bytes)	Byte order	Description
`len`	2	big-endian	Length of the `data` field.
`reserved`	2	n/a	Reserved for the future use. It must be 0 for the current protocol version.
`data`	`len`	n/a	Packet data.
`padding`	0-3	n/a	Padding is added to align the total packet size to 4 bytes.

The packet send procedure is the following:

1. Check if the packet fits into the buffer.
2. Write the packet to `data` FIFO buffer starting at `wr_idx`. Wrap it if needed.
3. Write a new value of the `wr_idx`.
4. Notify the receiver over the MBOX channel.

Initialization The initialization sequence is the following:

1. Set the `wr_idx` and `rd_idx` to zero.
2. Push a single packet to FIFO containing magic data: `45 6d 31 6c 31 4b 30 72 6e 33 6c 69 34`. The MBOX is not used yet.
3. Initialize the MBOX.
4. Repeat the notification over the MBOX channel using some interval, for example, 1 ms.
5. Wait for an incoming packet containing the magic data. It will arrive over the other pair (shared memory region and MBOX).
6. Stop repeating the MBOX notification.

After this, the ICMsg is bound, and it is ready to transfer packets.

ICMsg with dynamically allocated buffers backend Data transferred over this backend travels in dynamically allocated buffers on shared memory. Allocation is thread safe and can happen from any context. The backend supports:

- Multiple endpoints.
- No-copy sending.
- Holding RX buffers.
- Sending from an interrupt context.
- Two levels of endpoint priorities.
- Statistics and optional shell command with utilization report
- Up to 32 blocks are supported.
- Data cache support.
- Low memory footprint (around 2 kB of code).

Overview For each direction a shared memory region is reserved and each region is divided into two parts. One part forms a pool of fixed size buffers and the allocator builds a variable size buffer from adjacent buffers in the pool. The other part is used for control path consisting of two message queues (for each direction). There is a producer queue which is written by the sender and read by the receiver and a consumer queue which is written by the receiver and read by the sender. The producer queue has information about location (within the pool) of the next message. The consumer queue has information about location (within the pool) of the consumed message.

The data sending process is following:

- The sender allocates one or more blocks from the pool. If there are not enough sequential blocks, a thread context waits using the timeout provided in the parameter that also includes K_FOREVER and K_NO_WAIT.
- The allocated blocks are filled with data. At the beginning of the first block there is a 32 bit message header with length, endpoint ID and own block index. For the zero-copy case, this is done by the caller, otherwise, it is copied automatically. During this time other threads

are not blocked in any way as long as there are enough free blocks for them. They can allocate, send data and receive data.

- A block index with the beginning of the message is written to the producer queue. Information about the priority of the endpoint is appended to the block index. `CONFIG_IPC_SERVICE_BACKEND_ICBMSG_MAX_ACTIVE_COUNT` defines number of slots in the queue. Mailbox notification is send to the receiver.
- The receiver reads the producer queue. Higher prioriy messages are processed first. It can hold the data as long as desired. Again, other threads are not blocked as long as there are enough free blocks for them.
- When data is no longer needed, the receiver writes to the consumer queue the block index.
- The sender is performing a garbage collection by reading the consumer queue and freeing the buffers. Garbage collection is performed after sending any message or if there is no available buffers.

Configuration The backend is configured using Kconfig and devicetree.

There are following Kconfig options:

`CONFIG_IPC_SERVICE_BACKEND_ICBMSG_NUM_EP` - maximum number of registered endpoints.

`CONFIG_IPC_SERVICE_BACKEND_ICBMSG_MAX_ACTIVE_COUNT` - number of slots in the queues.

`CONFIG_IPC_SERVICE_BACKEND_ICBMSG_DEINIT` - support for deregistration and closing.

`CONFIG_IPC_SERVICE_BACKEND_ICBMSG_SHELL` - support for the shell command.

When configuring the backend, do the following:

- If at least one of the cores uses data cache on shared memory, set the `dcache-alignment` value. This must be the largest value of the invalidation or the write-back size for both sides of the communication. You can skip it if none of the communication sides is using data cache on shared memory.
- Define two memory regions and assign them to `tx-region` and `rx-region` of an instance. Ensure that the memory regions used for data exchange are unique (not overlapping any other region) and accessible by both domains (or CPUs).
- Define the number of allocable blocks for each region with `tx-blocks` and `rx-blocks`.
- Define MBOX devices for sending a signal that informs the other domain (or CPU) of the written data. Ensure that the other domain (or CPU) can receive the signal.

Caution

Make sure that you set correct value of the `dcache-alignment`. At first, wrong value may not show any signs, which may give a false impression that everything works. Unstable behavior will appear sooner or later.

If `dcache-alignment` is used then configuration for the number of blocks should be selected carefully to avoid inefficient usage of the memory. That is because the blocks are aligned to the cache alignment and if the number of blocks is not a multiple of the cache alignment then the last block will not be used efficiently. That is because the blocks and control data are aligned to the cache alignment. For example, if `dcache-alignment` is 32 and 1024 bytes of shared memory is used for one direction. Control data will take 64 bytes and 960 bytes are left for the buffers. Using 16 blocks will result in 32 bytes per block (due to the cache alignment). Using 15 blocks will result in 64 bytes per block (due to the cache alignment) with much better memory utilization.

See the following configuration example for one of the instances:

```
reserved-memory {
   tx: memory@20070000 {
      reg = <0x20070000 0x0800>;
   };

   rx: memory@20078000 {
      reg = <0x20078000 0x0800>;
   };
};

ipc {
   ipc0: ipc0 {
      compatible = "zephyr,ipc-icbmsg";
      dcache-alignment = <32>;
      tx-region = <&tx>;
      rx-region = <&rx>;
      tx-blocks = <16>;
      rx-blocks = <16>;
      mboxes = <&mbox 0>, <&mbox 1>;
      mbox-names = "tx", "rx";
      status = "okay";
   };
};
```

You must provide a similar configuration for the other side of the communication (domain or CPU). Swap the MBOX channels, memory regions (`tx-region` and `rx-region`), and block count (`tx-blocks` and `rx-blocks`).

Limitations

- Expected same endianness on both sides of the communication.
- No support for detection of unexpected remote reset.

Samples

- ipc_multi_endpoint

Detailed Protocol Specification The ICBMsg protocol transfers messages using dynamically allocated blocks of shared memory.

Shared Memory Organization The ICBMsg uses two shared memory regions, `rx-region` for message receiving, and `tx-region` for message transmission. The regions do not need to be next to each other, placed in any specific order, or be of the same size. Those regions are interchanged on each core.

Each shared memory region is divided into following two parts:

- **Control area** - An area reserved by producer and consumer queues.
- **Blocks area** - An area containing allocatable blocks carrying the content of the messages. This area is divided into even-sized blocks aligned to cache boundaries.

The location of each area is calculated to fulfill cache boundary requirements and allow optimal region usage. It is calculated using the following algorithm:

Inputs:

- `region_begin`, `region_end` - Boundaries of the region.
- `local_blocks` - Number of blocks in this region.

- `remote_blocks` - Number of blocks in the opposite region.
- `alignment` - Memory cache alignment.

The algorithm:

1. Align region boundaries to cache:
   - `region_begin_aligned = ROUND_UP(region_begin, alignment)`
   - `region_end_aligned = ROUND_DOWN(region_end, alignment)`
   - `region_size_aligned = region_end_aligned - region_begin_aligned`
2. Calculate the minimum size required for the control area:
   - for each queue there is `IPC_SERVICE_BACKEND_ICBMSG_MAX_ACTIVE_COUNT` bytes and 8 byte queue header.
   - Typically control data takes less than 64 bytes per direction.
3. Calculate available size for block area. Note that the actual size may be smaller because of block alignment:

   `blocks_area_available_size = region_size_aligned - control_area`
4. Calculate single block size:

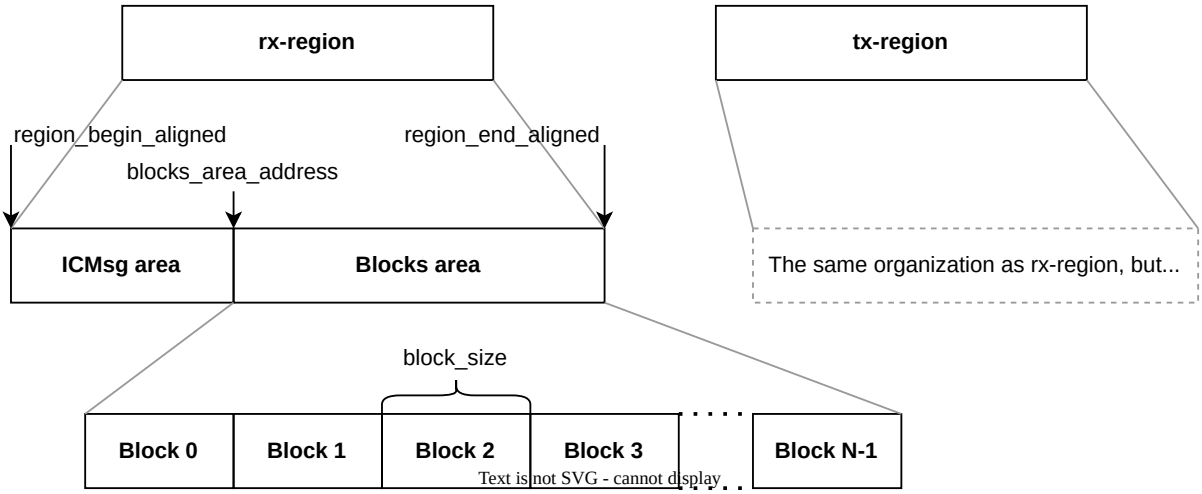
   `block_size = ROUND_DOWN(blocks_area_available_size / local_blocks, alignment)`
5. Calculate actual block area size:

   `blocks_area_size = block_size * local_blocks`
6. Calculate block area start address:

   `blocks_area_begin = region_end_aligned - blocks_area_size`

The result:

- `region_begin_aligned` - The start of ICMsg area.
- `blocks_area_begin` - End of ICMsg area and the start of block area.
- `block_size` - Single block size.
- `region_end_aligned` - End of blocks area.

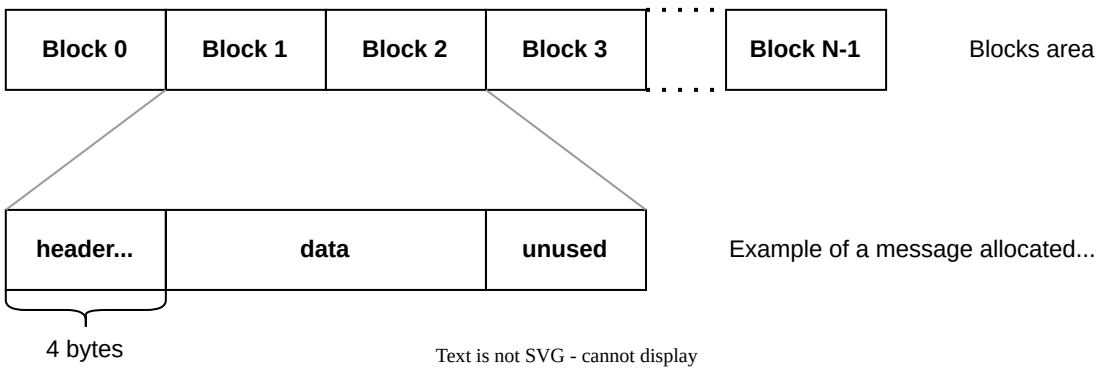


Message Transfer The ICBMsg uses following two types of messages:

- **Control message** - Messages like binding or unbinding.
- **Data message** - Message carrying actual user data.

They serve different purposes, but their lifetime and flow are the same. The following steps describe it:

1. The sender wants to send a message that contains `K` bytes.
2. The sender reserves blocks from his `tx-region` blocks area that can hold at least `K + 4` bytes. The additional `+ 4` bytes are reserved for the header. The blocks must be continuous (one after another). The sender is responsible for block allocation management. If blocks are not available then a thread context may block and an interrupt context will return error.
3. The sender fills the header.
4. The sender fills the remaining part of the blocks with his data. Unused space is ignored.
5. The sender writes the message to the producer queue and sends the mailbox signal.
6. The receiver executes mailbox callback in the mailbox interrupt context and reads the producer queue.
7. The receiver reads the block index and locates the message within his `rx-region`.
8. The receiver reads the endpoint and message length and processes the message.
9. The receiver consumes the message by writing its block index to the consumer queue. The mailbox signal is not send.
10. The sender checks consume queue after each sending or if sending fails. Messages are read from the consumer queue and released to the pool.



Binding Instances When backend instance is opened it send bound message which contains a 64 bit magic number. Mailbox callback is enabled and instance is waiting for bound message. After receiving bound message, instance is bound to the remote instance and endpoints can be registered.

Binding Endpoint Endpoint binding message contains SHA of the endpoint name and endpoint ID which is the index in the local array with endpoint's data.

There are two possible scenarios:

- The remote instance sent binding message for the endpoint before it was registered.
- The endpoint is registered before receiving the binding message from the remote instance.

When the binding message is received then SHA is compared to the stored SHA in the array of endpoint's data. If match is found then it means that the endpoint is already registered by the local instance. Endpoint ID is stored in the endpoint's data and bound callback is called. If match is not found then empty slot is found and endpoint ID and SHA is stored in the available slot.

When endpoint is registered then SHA for the name is calculated and compared to the stored SHA in the array of endpoint's data. If match is found then it means that remote binding message for that endpoint is already received. In this case binding message is sent to the remote instance and bound callback is called. If match is not found then empty slot is found and endpoint ID and SHA is stored in the available slot. Binding message is sent to the remote instance but endpoints are not bound yet.

Later on, remote endpoint ID is used for data message to identify the endpoint.

Unbinding Endpoint When endpoint is unregistered then unbinding control message is sent. Endpoint is removed from the array of endpoint's data. When unbinding message is received then endpoint slot is marked as empty and unbinding callback is called.

API Reference

IPC service API
IPC service APIs

IPC service backend API
IPC service backend

4.3.2 Zephyr bus (zbus)

The *Zephyr bus - zbus* is a lightweight and flexible software bus enabling a simple way for threads to talk to one another in a many-to-many way.

- *Concepts*
  - *Virtual Distributed Event Dispatcher*
  - *Limitations*
- *Usage*
  - *Publishing to a channel*
  - *Reading from a channel*
  - *Notifying a channel*
  - *Declaring channels and observers*
  - *Iterating over channels and observers*
  - *Advanced channel control*
- *Proxy Agent Communication (Experimental)*
  - *Concepts*
  - *Transport Backends*
  - *Usage*
- *Samples*
- *Suggested Uses*

- *Configuration Options*
  - *Proxy Agent Configuration Options*
- *API Reference*

Concepts

Threads can send messages to one or more observers using zbus. It makes the many-to-many communication possible. The bus implements message-passing and publish/subscribe communication paradigms that enable threads to communicate synchronously or asynchronously through shared memory.

The communication through zbus is channel-based. Threads (or callbacks) use channels to exchange messages. Additionally, besides other actions, threads can publish and observe channels. When a thread publishes a message on a channel, the bus will make the message available to all the published channel's observers. Based on the observer's type, it can access the message directly, receive a copy of it, or even receive only a reference of the published channel.

The figure below shows an example of a typical application using zbus in which the application logic (hardware independent) talks to other threads via software bus. Note that the threads are decoupled from each other because they only use zbus channels and do not need to know each other to talk.

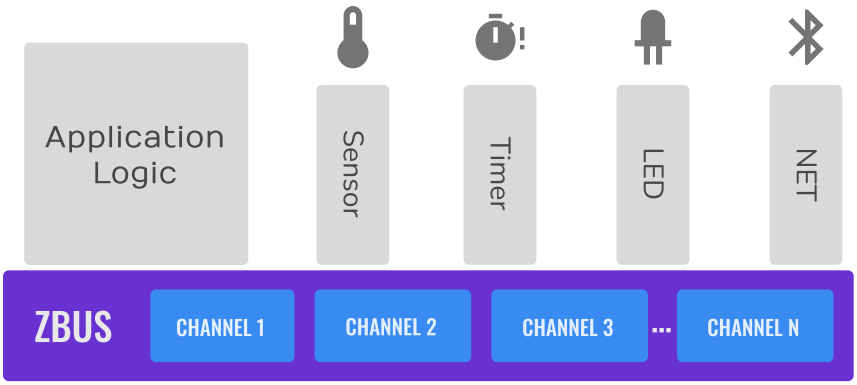


Fig. 27: A typical zbus application architecture.

The bus comprises:

- Set of channels that consists of the control metadata information, and the message itself;
- *Virtual Distributed Event Dispatcher* (VDED), the bus logic responsible for sending notifications/messages to the observers. The VDED logic runs inside the publishing action in the same thread context, giving the bus an idea of a distributed execution. When a thread publishes to a channel, it also propagates the notifications to the observers;
- Threads (subscribers and message subscribers), callbacks (listeners), and async listeners (callbacks deferred to a work queue) publishing, reading, and receiving notifications from the bus.

The bus makes the publish, read, claim, finish, notify, and subscribe actions available over channels. Publishing, reading, claiming, and finishing are available in all RTOS thread contexts and ISRs. The publish and read operations are simple and fast; the procedure is channel locking followed by a memory copy to and from a shared memory region and then a channel unlocking. Another essential aspect of zbus is the observers. There are four types of observers:

- Listeners, a callback that the event dispatcher executes every time an observed channel is published or notified;

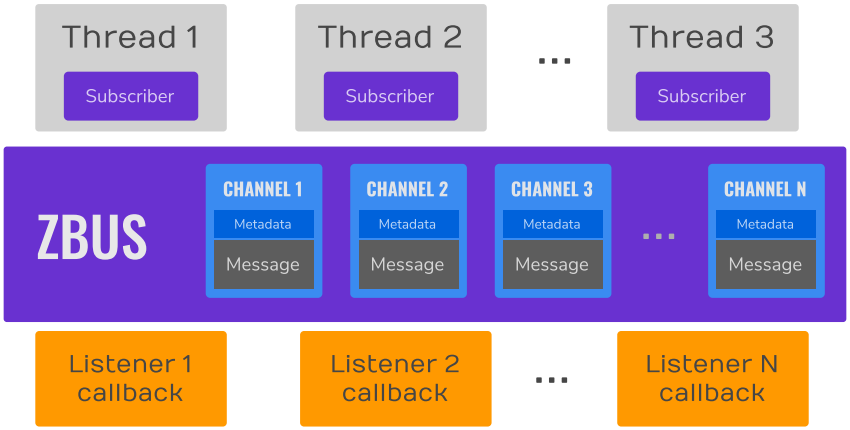


Fig. 28: ZBus anatomy.

Fig. 29: ZBus observers.

- Async Listeners, a callback that the event dispatcher schedules to execute in a work queue (system work queue by default) every time an observed channel is published or notified;
- Subscriber, a thread-based observer that relies internally on a message queue where the event dispatcher puts a changed channel's reference every time an observed channel is published or notified. Note this kind of observer does not receive the message itself. It should read the message from the channel after receiving the notification;
- Message subscribers, a thread-based observer that relies internally on a FIFO where the event dispatcher puts a copy of the message every time an observed channel is published or notified.

Channel observation structures define the relationship between a channel and its observers. For every observation, a channel/observer pair is created. Developers can statically allocate observations using the `ZBUS_CHAN_DEFINE` or `ZBUS_CHAN_ADD_OBS`. There are also runtime observers, enabling developers to create runtime observations. It is possible to disable an observer entirely or observations individually. The event dispatcher will ignore disabled observers and observations.

The above figure illustrates some states, from (a) to (d), for channels from `C1` to `C5`, `Subscriber 1`, and the observations. The last two are in orange to indicate they are dynamically allocated (runtime observation). (a) shows that the observer and all observations are enabled. (b) shows the observer is disabled, so the event dispatcher will ignore it. (c) shows the observer enabled. However, there is one static observation disabled. The event dispatcher will only stop sending notifications from channel `C3`. In (d), the event dispatcher will stop sending notifications from channels `C3` and `C5` to `Subscriber 1`.

Suppose a usual sensor-based solution is in the figure below for illustration purposes. When triggered, the timer publishes to the `Trigger` channel. As the sensor thread subscribed to the `Trigger` channel, it receives the sensor data. Notice the VDED executes the `Blink` because it also listens to the `Trigger` channel. When the sensor data is ready, the sensor thread publishes it to the `Sensor data` channel. The core thread receives the message as a `Sensor data` channel message subscriber, processes the sensor data, and stores it in an internal sample buffer. It repeats until the sample buffer is full; when it happens, the core thread aggregates the sample buffer information, prepares a package, and publishes that to the `Payload` channel. The LoRa thread receives that because it is a `Payload` channel message subscriber and sends the payload to the cloud. When it completes the transmission, the LoRa thread publishes to the `Transmission done` channel. The VDED executes the `Blink` again since it listens to the `Transmission done` channel.

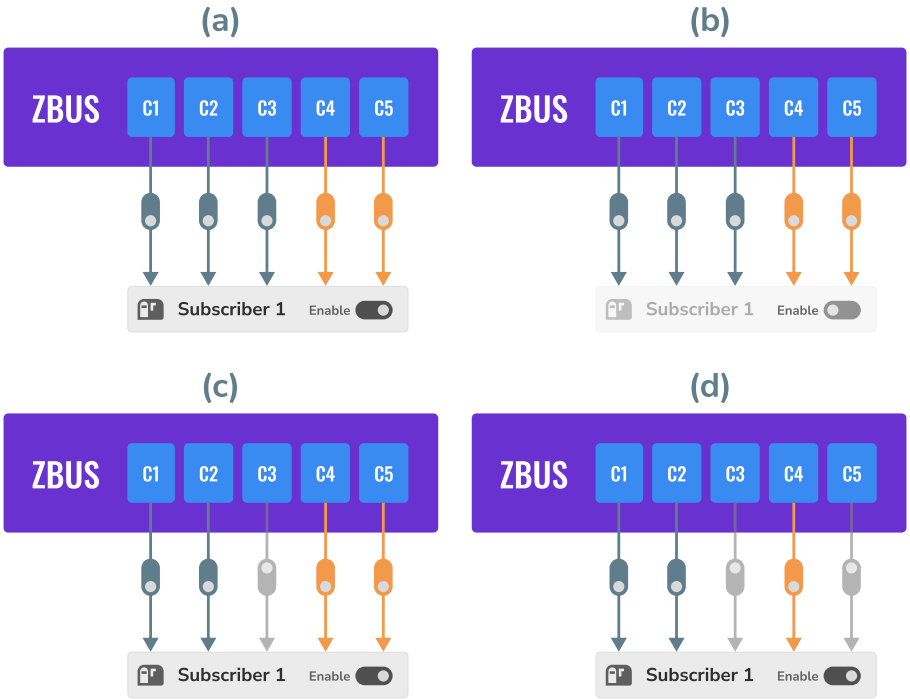


Fig. 30: ZBus observation mask.

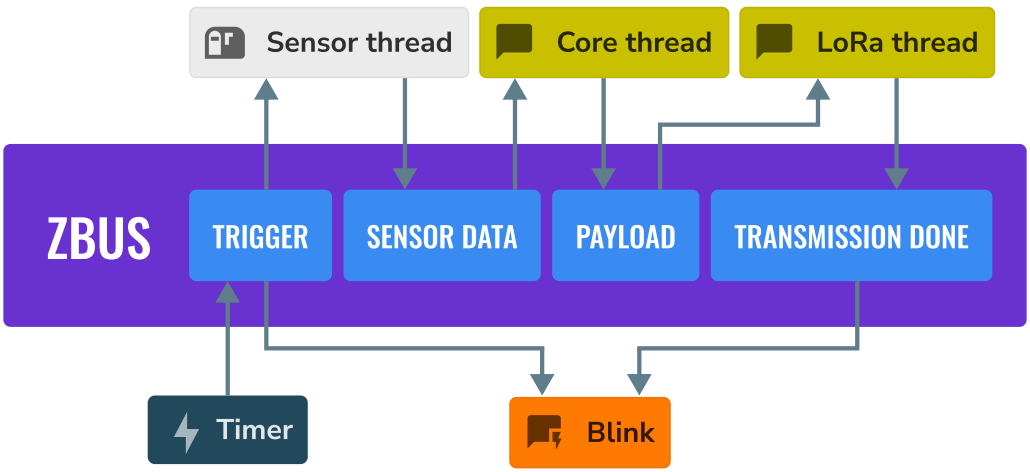


Fig. 31: ZBus sensor-based application.

This way of implementing the solution makes the application more flexible, enabling us to change things independently. For example, we want to change the trigger from a timer to a button press. We can do that, and the change does not affect other parts of the system. Likewise, we would like to change the communication interface from LoRa to Bluetooth; we only need to change the LoRa thread. No other change is required in order to make that work. Thus, the developer would do that for every block of the image. Based on that, this indicates that zbus promotes decoupling in the system architecture.

Another important aspect of using zbus is the reuse of system modules. If a code portion with well-defined behaviors (we call that module) only uses zbus channels and not hardware interfaces, it can easily be reused in other solutions. The new solution must implement the interfaces (set of channels) the module needs to work. That indicates zbus could improve the module reuse.

The last important note is the flexibility of zbus. Zbus provides many features that give developers freedom to create solutions that fit their specific needs. These features include:

- Messages can be dynamically or statically allocated
- Notifications can be synchronous or asynchronous
- Channels can be controlled in various ways by claiming them
- Custom metadata can be added to channels using the user-data field
- Optional validators can be used to enforce message format accuracy

These characteristics expand the range of solutions that can be built with zbus and make it well-suited as an open-source community tool.

Virtual Distributed Event Dispatcher The VDED execution always happens in the publisher's context. It can be a thread or an ISR. Be careful with publications inside ISR because the scheduler won't preempt the VDED. Use that wisely. The basic description of the execution is as follows:

- The channel lock is acquired;
- The channel receives the new message via direct copy (by a raw `memcpy()`);
- The event dispatcher logic executes the listeners, sends a copy of the message to the message subscribers, and pushes the channel's reference to the subscribers' notification message queue in the same sequence they appear on the channel observers' list. The listeners can perform non-copy quick access to the constant message reference directly (via the `zbus_chan_const_msg()` function) since the channel is still locked;
- At last, the publishing function unlocks the channel.

To illustrate the VDED execution, consider the example illustrated below. We have four threads in ascending priority `S1`, `MS2`, `MS1`, and `T1` (the highest priority); two listeners, `L1` and `L2`; and channel A. Supposing `L1`, `L2`, `MS1`, `MS2`, and `S1` observe channel A.

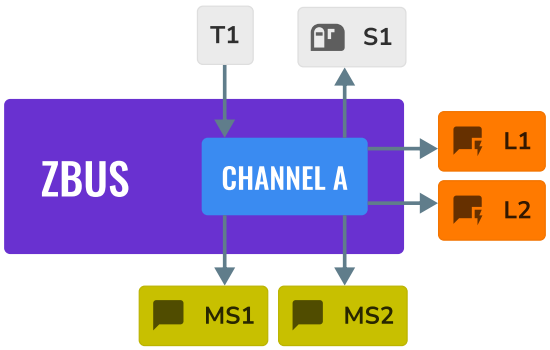


Fig. 32: ZBus VDED execution example scenario.

The following code implements channel A. Note the `struct a_msg` is illustrative only.

```
ZBUS_CHAN_DEFINE(a_chan,                        /* Name */
         struct a_msg,                          /* Message type */

         NULL,                                  /* Validator */
         NULL,                                  /* User Data */
         ZBUS_OBSERVERS(L1, L2, MS1, MS2, S1),  /* observers */
         ZBUS_MSG_INIT(0)                       /* Initial value {0} */
);
```

In the figure below, the letters indicate some action related to the VDED execution. The X-axis represents the time, and the Y-axis represents the priority of threads. Channel A's message, represented by a voice balloon, is only one memory portion (shared memory). It appears several times only as an illustration of the message at that point in time.

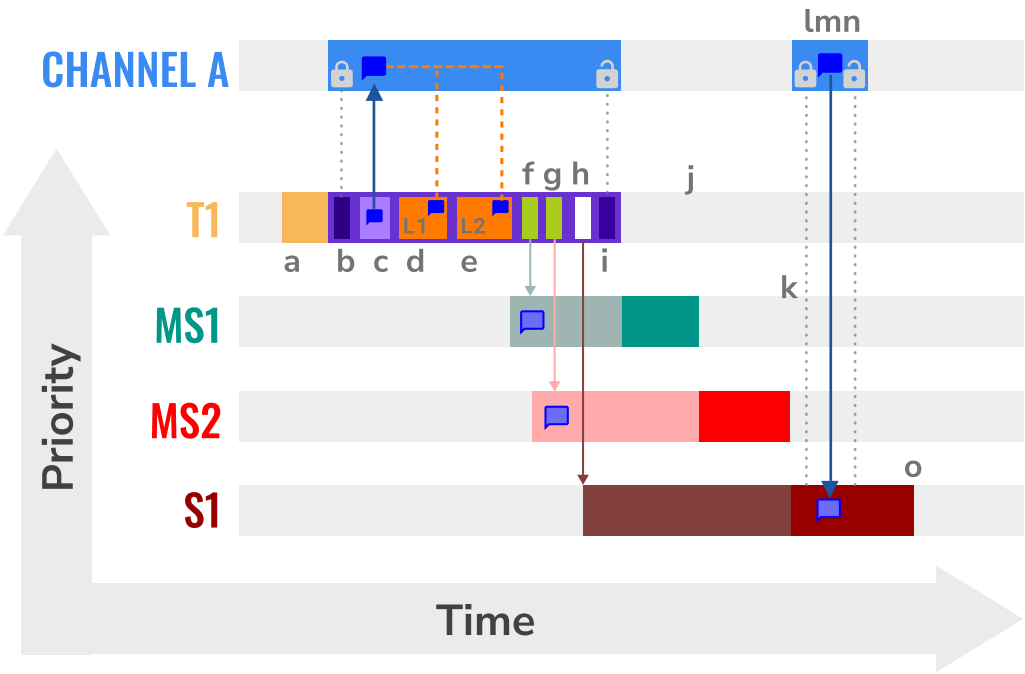


Fig. 33: ZBus VDED execution detail for priority T1 > MS1 > MS2 > S1.

The figure above illustrates the actions performed during the VDED execution when T1 publishes to channel A. Thus, the table below describes the activities (represented by a letter) of the VDED execution. The scenario considers the following priorities: T1 > MS1 > MS2 > S1. T1 has the highest priority.

Table 44: VDED execution steps in detail for priority T1 > MS1 > MS2 > S1.

Actions	Description
a	T1 starts and, at some point, publishes to channel A.
b	The publishing (VDED) process starts. The VDED locks the channel A.
c	The VDED copies the T1 message to the channel A message.
d, e	The VDED executes L1 and L2 in the respective sequence. Inside the listeners, usually, there is a call to the `zbus_chan_const_msg()` function, which provides a direct constant reference to channel A's message. It is quick, and no copy is needed here.
f, g	The VDED copies the message and sends that to MS1 and MS2 sequentially. Notice the threads get ready to execute right after receiving the notification. However, they go to a pending state because they have less priority than T1.
h	The VDED pushes the notification message to the queue of S1. Notice the thread gets ready to execute right after receiving the notification. However, it goes to a pending state because it cannot access the channel since it is still locked.
i	VDED finishes the publishing by unlocking channel A. The MS1 leaves the pending state and starts executing.
j	MS1 finishes execution. The MS2 leaves the pending state and starts executing.
k	MS2 finishes execution. The S1 leaves the pending state and starts executing.
l, m, n	The S1 leaves the pending state since channel A is not locked. It gets in the CPU again and starts executing. As it did receive a notification from channel A, it performed a channel read (as simple as lock, memory copy, unlock), continues its execution and goes out of the CPU.
o	S1 finishes its workload.

The figure below illustrates the actions performed during the VDED execution when T1 publishes to channel A. The scenario considers the following priorities: T1 < MS1 < MS2 < S1.

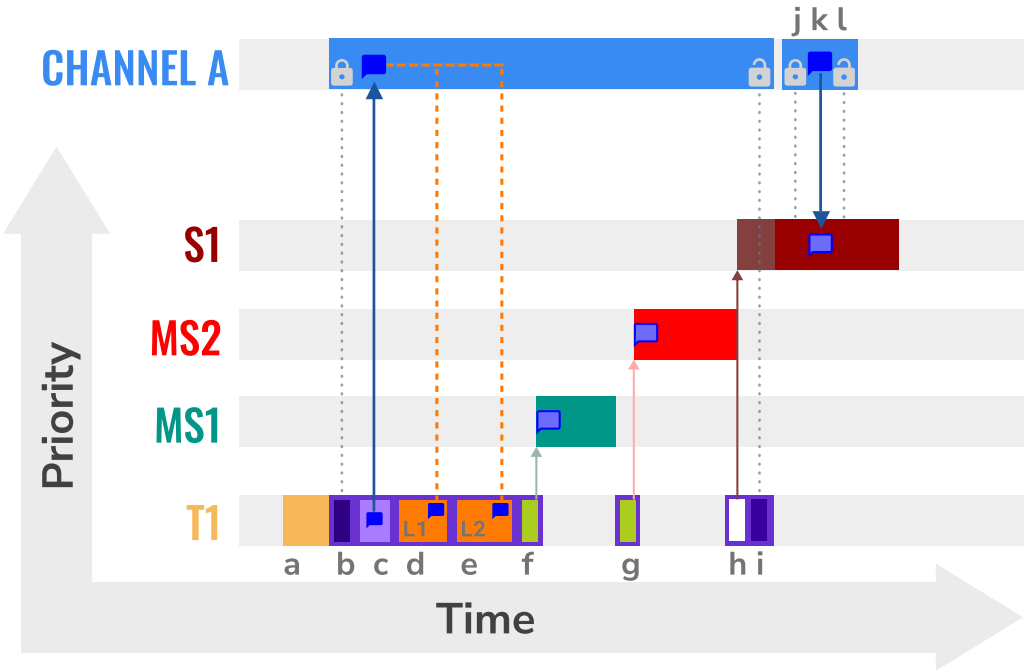


Fig. 34: ZBus VDED execution detail for priority T1 < MS1 < MS2 < S1.

Thus, the table below describes the activities (represented by a letter) of the VDED execution.

Table 45: VDED execution steps in detail for priority T1 < MS1 < MS2 < S1.

Actions	Description
a	T1 starts and, at some point, publishes to channel A.
b	The publishing (VDED) process starts. The VDED locks the channel A.
c	The VDED copies the T1 message to the channel A message.
d, e	The VDED executes L1 and L2 in the respective sequence. Inside the listeners, usually, there is a call to the `zbus_chan_const_msg()` function, which provides a direct constant reference to channel A's message. It is quick, and no copy is needed here.
f	The VDED copies the message and sends that to MS1. MS1 preempts T1 and starts working. After that, the T1 regains the CPU.
g	The VDED copies the message and sends that to MS2. MS2 preempts T1 and starts working. After that, the T1 regains the CPU.
h	The VDED pushes the notification message to the queue of S1.
i	VDED finishes the publishing by unlocking channel A.
j, k, l	The S1 leaves the pending state since channel A is not locked. It gets in the CPU again and starts executing. As it did receive a notification from channel A, it performs a channel read (as simple as lock, memory copy, unlock), continues its execution, and yields the CPU.

HLP priority boost ZBus implements the Highest Locker Protocol that relies on the observers' thread priority to determine a temporary publisher priority. The protocol considers the channel's Highest Observer Priority (HOP); even if the observer is not waiting for a message on the channel, it is considered in the calculation. The VDED will elevate the publisher's priority based on the HOP to ensure small latency and as few preemptions as possible.

Note

The priority boost is enabled by default. To deactivate it, disable the `CONFIG_ZBUS_PRIORITY_BOOST` configuration option.

Warning

ZBus priority boost does not consider runtime observers on the HOP calculations.

The figure below illustrates the actions performed during the VDED execution when T1 publishes to channel A. The scenario considers the priority boost feature and the following priorities: T1 < MS1 < MS2 < S1.

To properly use the priority boost, attaching the observer to a thread is necessary. When the subscriber is attached to a thread, it assumes its priority, and the priority boost algorithm will consider the observer's priority. The following code illustrates the thread-attaching function.

```
ZBUS_SUBSCRIBER_DEFINE(s1, 4);
void s1_thread(void *ptr1, void *ptr2, void *ptr3)
{
        ARG_UNUSED(ptr1);
        ARG_UNUSED(ptr2);
        ARG_UNUSED(ptr3);

        const struct zbus_channel *chan;
```

(continues on next page)

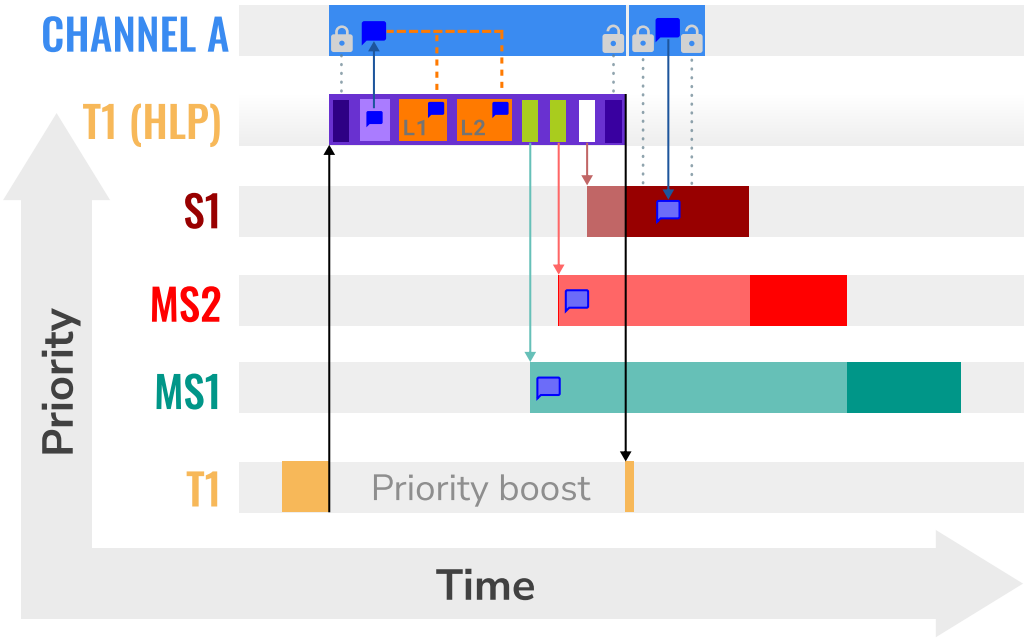


Fig. 35: ZBus VDED execution detail with priority boost enabled and for priority T1 < MS1 < MS2 < S1.

(continued from previous page)

```
        zbus_obs_attach_to_thread(&s1);

        while (1) {
                zbus_sub_wait(&s1, &chan, K_FOREVER);

                /* Subscriber implementation */

        }
}
K_THREAD_DEFINE(s1_id, CONFIG_MAIN_STACK_SIZE, s1_thread, NULL, NULL, NULL, 2, 0,␣
↪0);
```

On the above code, the `zbus_obs_attach_to_thread()` will set the s1 observer with priority two as the thread has that priority. It is possible to reverse that by detaching the observer using the `zbus_obs_detach_from_thread()`. Only enabled observers and observations will be considered on the channel HOP calculation. Masking a specific observation of a channel will affect the channel HOP.

In summary, the benefits of the feature are:

- The HLP is more effective for zbus than the mutexes priority inheritance;
- No bounded priority inversion will happen among the publisher and the observers;
- No other threads (that are not involved in the communication) with priority between T1 and S1 can preempt T1, avoiding unbounded priority inversion;
- Message subscribers will wait for the VDED to finish the message delivery process. So the VDED execution will be faster and more consistent;
- The HLP priority is dynamic and can change in execution;
- ZBus operations can be used inside ISRs;
- The priority boosting feature can be turned off, and plain semaphores can be used as the

channel lock mechanism;

- The Highest Locker Protocol's major disadvantage, the Inheritance-related Priority Inversion, is acceptable in the zbus scenario since it will ensure a small bus latency.

Limitations Based on the fact that developers can use zbus to solve many different problems, some challenges arise. ZBus will not solve every problem, so it is necessary to analyze the situation to be sure zbus is applicable. For instance, based on the zbus benchmark, it would not be well suited to a high-speed stream of bytes between threads. The *Pipe* kernel object solves this kind of need.

Delivery guarantees ZBus always delivers the messages to the listeners, message subscribers, and async listeners. However, there are no message delivery guarantees for subscribers because zbus only sends the notification, but the message reading depends on the subscriber's implementation. It is possible to increase the delivery rate by following design tips:

- Keep the listeners as quick as possible (deal with them as ISRs). If time-consuming processing is required, consider offloading some or all of it to a work queue using async listeners.
- Try to give consumers a high priority to avoid losses.
- Leave spare CPU for observers to consume data produced.
- Consider using message queues or pipes for intensive byte transfers.

Warning

ZBus uses include/zephyr/net_buf.h (network buffers) to exchange data with message subscribers. Thus, choose carefully the configurations `CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_SIZE` and `CONFIG_HEAP_MEM_POOL_ADD_SIZE_ZBUS` are crucial to a proper VDED execution (delivery guarantee) considering message subscribers and async listeners. If you want to keep an isolated pool for a specific set of channels, you can use `CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_ISOLATION` with a dedicated pool. Look at the zbus-msg-subscriber and zbus-async-listeners to see the isolation in action.

Warning

Subscribers will receive only the reference of the changing channel. A data loss may be perceived if the channel is published twice before the subscriber reads it. The second publication overwrites the value from the first. Thus, the subscriber will receive two notifications, but only the last data is there.

Message delivery sequence The message delivery will follow the precedence:

1. Observers defined in a channel using the `ZBUS_CHAN_DEFINE` (following the definition sequence);
2. Observers defined using the `ZBUS_CHAN_ADD_OBS` based on the sequence priority (parameter of the macro);
3. The latest is the runtime observers in the addition sequence using the `zbus_chan_add_obs()`.

Note

The VDED will ignore all disabled observers or observations.

Usage

ZBus operation depends on channels and observers. Therefore, it is necessary to determine the channel's message during the channel definition and its list of observers, which can be either statically (in the channel definition), using the `ZBUS_CHAN_ADD_OBS` or at runtime (see *runtime observers*). A message is a regular C struct; the observer can be a listener (synchronous), an async listener (asynchronous), a subscriber (asynchronous), or a message subscriber (asynchronous).

The following code defines and initializes a regular channel and its dependencies. This channel exchanges accelerometer data, for example.

```
struct acc_msg {
        int x;
        int y;
        int z;
};

ZBUS_CHAN_DEFINE(acc_chan,                          /* Name */
         struct acc_msg,                            /* Message type */

         NULL,                                      /* Validator */
         NULL,                                      /* User Data */
         ZBUS_OBSERVERS(my_listener, my_subscriber,
                        my_msg_subscriber),         /* observers */
         ZBUS_MSG_INIT(.x = 0, .y = 0, .z = 0)      /* Initial value */
);

void listener_callback_example(const struct zbus_channel *chan)
{
        const struct acc_msg *acc;
        if (&acc_chan == chan) {
                acc = zbus_chan_const_msg(chan); // Direct message access
                LOG_DBG("From listener -> Acc x=%d, y=%d, z=%d", acc->x, acc->y,␣
↪acc->z);
        }
}

ZBUS_LISTENER_DEFINE(my_listener, listener_callback_example);

ZBUS_LISTENER_DEFINE(my_listener2, listener_callback_example);

ZBUS_CHAN_ADD_OBS(acc_chan, my_listener2, 3);

ZBUS_SUBSCRIBER_DEFINE(my_subscriber, 4);
void subscriber_task(void)
{
        const struct zbus_channel *chan;

        while (!zbus_sub_wait(&my_subscriber, &chan, K_FOREVER)) {
                struct acc_msg acc = {0};

                if (&acc_chan == chan) {
                        // Indirect message access
                        zbus_chan_read(&acc_chan, &acc, K_NO_WAIT);
                        LOG_DBG("From subscriber -> Acc x=%d, y=%d, z=%d", acc.x,␣
↪acc.y, acc.z);
                }
        }
```

(continues on next page)

(continued from previous page)

```
}
K_THREAD_DEFINE(subscriber_task_id, 512, subscriber_task, NULL, NULL, NULL, 3, 0,␣
↪0);

ZBUS_MSG_SUBSCRIBER_DEFINE(my_msg_subscriber);
static void msg_subscriber_task(void *ptr1, void *ptr2, void *ptr3)
{
        ARG_UNUSED(ptr1);
        ARG_UNUSED(ptr2);
        ARG_UNUSED(ptr3);
        const struct zbus_channel *chan;

        struct acc_msg acc = {0};

        while (!zbus_sub_wait_msg(&my_msg_subscriber, &chan, &acc, K_FOREVER)) {
                if (&acc_chan == chan) {
                        LOG_INF("From msg subscriber -> Acc x=%d, y=%d, z=%d",␣
↪acc.x, acc.y, acc.z);
                }
        }
}
K_THREAD_DEFINE(msg_subscriber_task_id, 1024, msg_subscriber_task, NULL, NULL,␣
↪NULL, 3, 0, 0);
```

It is possible to add static observers to a channel using the `ZBUS_CHAN_ADD_OBS`. We call that a post-definition static observer. The command enables us to indicate an initialization priority that affects the observers' initialization order. The sequence priority param only affects the post-definition static observers. There is no possibility to overwrite the message delivery sequence of the static observers.

> **Note**
>
> It is unnecessary to claim/lock a channel before accessing the message inside the listener since the event dispatcher calls listeners with the notifying channel already locked. Subscribers, however, must claim/lock that or use regular read operations to access the message after being notified.

Channels can have a *validator function* that enables a channel to accept only valid messages. Publish attempts invalidated by hard channels will return immediately with an error code. This allows original creators of a channel to exert some authority over other developers/publishers who may want to piggy-back on their channels. The following code defines and initializes a *hard channel* and its dependencies. Only valid messages can be published to a *hard channel*. It is possible because a *validator function* was passed to the channel's definition. In this example, only messages with `move` equal to 0, -1, and 1 are valid. Publish function will discard all other values to `move`.

```
struct control_msg {
        int move;
};

bool control_validator(const void* msg, size_t msg_size) {
        const struct control_msg* cm = msg;
        bool is_valid = (cm->move == -1) || (cm->move == 0) || (cm->move == 1);
        return is_valid;
}
```

(continues on next page)

(continued from previous page)

```
static int message_count = 0;

ZBUS_CHAN_DEFINE(control_chan,     /* Name */
         struct control_msg,       /* Message type */

         control_validator,        /* Validator */
         &message_count,           /* User data */
         ZBUS_OBSERVERS_EMPTY,     /* observers */
         ZBUS_MSG_INIT(.move = 0) /* Initial value */
);
```

The following sections describe in detail how to use zbus features.

Publishing to a channel Messages are published to a channel in zbus by calling `zbus_chan_pub()`. For example, the following code builds on the examples above and publishes to channel `acc_chan`. The code is trying to publish the message `acc1` to channel `acc_chan`, and it will wait up to one second for the message to be published. Otherwise, the operation fails. As can be inferred from the code sample, it's OK to use stack allocated messages since VDED copies the data internally.

```
struct acc_msg acc1 = {.x = 1, .y = 1, .z = 1};
zbus_chan_pub(&acc_chan, &acc1, K_SECONDS(1));
```

Warning

Only use this function inside an ISR with a `K_NO_WAIT` timeout.

Reading from a channel Messages are read from a channel in zbus by calling `zbus_chan_read()`. So, for example, the following code tries to read the channel `acc_chan`, which will wait up to 500 milliseconds to read the message. Otherwise, the operation fails.

```
struct acc_msg acc = {0};
zbus_chan_read(&acc_chan, &acc, K_MSEC(500));
```

Warning

Only use this function inside an ISR with a `K_NO_WAIT` timeout.

Warning

Choose the timeout of `zbus_chan_read()` after receiving a notification from `zbus_sub_wait()` carefully because the channel will always be unavailable during the VDED execution. Using `K_NO_WAIT` for reading is highly likely to return a timeout error if there are more than one subscriber. For example, consider the VDED illustration again and notice how `S1` read attempts would definitely fail with K_NO_WAIT. For more details, check the *Virtual Distributed Event Dispatcher* section.

Notifying a channel It is possible to force zbus to notify a channel's observers by calling `zbus_chan_notify()`. For example, the following code builds on the examples above and forces a notification for the channel `acc_chan`. Note this can send events with no message, which does

not require any data exchange. See the code example under *Claim and finish a channel* where this may become useful.

```
zbus_chan_notify(&acc_chan, K_NO_WAIT);
```

Warning

Only use this function inside an ISR with a K_NO_WAIT timeout.

Declaring channels and observers For accessing channels or observers from files other than its defining files, it is necessary to declare them by calling ZBUS_CHAN_DECLARE and ZBUS_OBS_DECLARE. In other words, zbus channel definitions and declarations with the same channel names in different files would point to the same (global) channel. Thus, developers should be careful about existing channels, and naming new channels or linking will fail. It is possible to declare more than one channel or observer on the same call. The following code builds on the examples above and displays the defined channels and observers.

```
ZBUS_OBS_DECLARE(my_listener, my_subscriber);
ZBUS_CHAN_DECLARE(acc_chan, version_chan);
```

Unique channel identifiers To simplify integrations with external entities, it is possible to assign a unique numeric identifier to a channel. Users can then retrieve the channel reference by using the identifier with zbus_chan_from_id(), rather than needing to obtain the reference at compile time with ZBUS_CHAN_DECLARE. Channels using this feature are declared with ZBUS_CHAN_DEFINE_WITH_ID.

```
ZBUS_CHAN_DEFINE_WITH_ID(control_chan,    /* Name */
    0x12345678,              /* Unique channel identifier */
    struct control_msg,      /* Message type */
    control_validator,       /* Validator */
    &message_count,          /* User data */
    ZBUS_OBSERVERS_EMPTY,    /* observers */
    ZBUS_MSG_INIT(.move = 0) /* Initial value */
);

static void channel_retrieve(void)
{
    const struct zbus_channel *chan = zbus_chan_from_id(0x12345678);

    ...
}
```

Runtime channel creation ZBus allows for the creation of channels at runtime when CONFIG_ZBUS_RUNTIME_CHANNEL_REGISTRATION is enabled. This is useful in scenarios where the number of channels or their configuration is not known at compile time. The following code demonstrates how to create a channel at runtime. Note that the zbus_runtime_channel, zbus_channel_data and message structures passed to zbus_runtime_channel_register() must remain valid in memory until zbus_runtime_channel_unregister() is called.

```
struct runtime_msg {
    int data;
};

void create_runtime_channel(void)
```

(continues on next page)

(continued from previous page)

```
{
    struct zbus_runtime_channel runtime_chan;
    struct zbus_channel_data data;
    struct runtime_msg msg = {0};
    int ret;

    /* Initialise and register the runtime channel */
    zbus_runtime_channel_init(&runtime_chan, &data, "chan", 0x12345678, NULL, &msg,
                              sizeof(msg), NULL);
    ret = zbus_runtime_channel_register(&runtime_chan);
    if (ret < 0) {
        LOG_ERR("Failed to register runtime channel (%d)", ret);
        return;
    }

    /* Use the runtime channel here */
    struct runtime_msg pub_data = {42};

    zbus_chan_pub(&runtime_chan.channel, &pub_data, K_MSEC(100));
    ...

    /* Unregister the runtime channel when done */
    zbus_runtime_channel_unregister(&runtime_chan);
}
```

Iterating over channels and observers ZBus subsystem also implements *Iterable Sections* for channels and observers, for which there are supporting APIs like `zbus_iterate_over_channels()`, `zbus_iterate_over_channels_with_user_data()`, `zbus_iterate_over_observers()` and `zbus_iterate_over_observers_with_user_data()`. This feature enables developers to call a procedure over all declared channels, where the procedure parameter is a `zbus_channel`. The execution sequence is in the alphabetical name order of the channels (see *Iterable Sections* documentation for details). ZBus also implements this feature for `zbus_observer`.

```
static bool print_channel_data_iterator(const struct zbus_channel *chan, void␣
↪*user_data)
{
      int *count = user_data;

      LOG_INF("%d - Channel %s:", *count, zbus_chan_name(chan));
      LOG_INF("      Message size: %d", zbus_chan_msg_size(chan));
      LOG_INF("      Observers:");

      ++(*count);

      struct zbus_channel_observation *observation;

      for (int16_t i = *chan->observers_start_idx, limit = *chan->
↪observers_end_idx; i < limit;
            ++i) {
            STRUCT_SECTION_GET(zbus_channel_observation, i, &observation);

            LOG_INF("      - %s", observation->obs->name);
      }
```

(continues on next page)

(continued from previous page)

```
        struct zbus_observer_node *obs_nd, *tmp;

        SYS_SLIST_FOR_EACH_CONTAINER_SAFE(chan->observers, obs_nd, tmp, node) {
                LOG_INF("       - %s", obs_nd->obs->name);
        }

        return true;
}

static bool print_observer_data_iterator(const struct zbus_observer *obs, void␣
↪*user_data)
{
        int *count = user_data;

        LOG_INF("%d - %s %s", *count, obs->queue ? "Subscriber" : "Listener",␣
↪zbus_obs_name(obs));

        ++(*count);

        return true;
}

int main(void)
{
        int count = 0;

        LOG_INF("Channel list:");

        zbus_iterate_over_channels_with_user_data(print_channel_data_iterator, &
↪count);

        count = 0;

        LOG_INF("Observers list:");

        zbus_iterate_over_observers_with_user_data(print_observer_data_iterator, &
↪count);

        return 0;
}
```

The code will log the following output:

```
D: Channel list:
D: 0 - Channel acc_chan:
D:       Message size: 12
D:       Observers:
D:       - my_listener
D:       - my_subscriber
D: 1 - Channel version_chan:
D:       Message size: 4
D:       Observers:
D: Observers list:
D: 0 - Listener my_listener
D: 1 - Subscriber my_subscriber
```

Advanced channel control ZBus was designed to be as flexible and extensible as possible. Thus, there are some features designed to provide some control and extensibility to the bus.

Listeners message access For performance purposes, listeners can access the receiving channel message directly since they already have the channel locked for it. To access the channel's message, the listener should use the `zbus_chan_const_msg()` because the channel passed as an argument to the listener function is a constant pointer to the channel. The const pointer return type tells developers not to modify the message.

```
void listener_callback_example(const struct zbus_channel *chan)
{
        const struct acc_msg *acc;
        if (&acc_chan == chan) {
                acc = zbus_chan_const_msg(chan); // Use this
                // instead of zbus_chan_read(chan, &acc, K_MSEC(200))
                // or zbus_chan_msg(chan)

                LOG_DBG("From listener -> Acc x=%d, y=%d, z=%d", acc->x, acc->y,␣
↪acc->z);
        }
}
```

Async listeners message access Async listeners are implemented by utilizing the established message subscriber infrastructure. They are executed in a work queue context rather than in the publisher's context. When using the system work queue, the user will experience similar behavior to the regular listeners. Since the system work queue is, by default, a cooperative thread with a priority of -1, async listeners would run before all other application threads (usually preemptive) after the VDED execution.

Async listeners can access the receiving channel's message directly via the message copy reference passed to the callback. Be aware that the message copy is freed just after the async listener execution. To access the channel's message, the async listener should only cast to the proper constant message format. The following example demonstrates how to access the message from an async listener.

```
static void async_listener_callback(const struct zbus_channel *chan, const void␣
↪*message)
{
        if (chan != &chan_event) {
                LOG_ERR("Unexpected channel");
                return;
        }

        const struct msg_event *msg = message;

        LOG_INF("From async listener -> Evt=%d | %s", msg->type,
                k_thread_name_get(k_current_get()));
}
```

User Data It is possible to pass custom data into the channel's `user_data` for various purposes, such as writing channel metadata. That can be achieved by passing a pointer to the channel definition macro's `user_data` field, which will then be accessible by others. Note that `user_data` is individual for each channel. Also, note that `user_data` access is not thread-safe. For thread-safe access to `user_data`, see the next section.

Claim and finish a channel To take more control over channels, two functions were added `zbus_chan_claim()` and `zbus_chan_finish()`. With these functions, it is possible to access the

channel's metadata safely. When a channel is claimed, no actions are available to that channel. After finishing the channel, all the actions are available again.

Warning

Never change the fields of the channel struct directly. It may cause zbus behavior inconsistencies and scheduling issues.

Warning

Only use this function inside an ISR with a `K_NO_WAIT` timeout.

The following code builds on the examples above and claims the `acc_chan` to set the `user_data` to the channel. Suppose we would like to count how many times the channels exchange messages. We defined the `user_data` to have the 32 bits integer. This code could be added to the listener code described above.

```
if (!zbus_chan_claim(&acc_chan, K_MSEC(200))) {
        int *message_counting = (int *) zbus_chan_user_data(&acc_chan);
        *message_counting += 1;
        zbus_chan_finish(&acc_chan);
}
```

The following code has the exact behavior of the code in *Publishing to a channel*.

```
if (!zbus_chan_claim(&acc_chan, K_MSEC(200))) {
        struct acc_msg *acc1 = (struct acc_msg *) zbus_chan_msg(&acc_chan);
        acc1->x = 1;
        acc1->y = 1;
        acc1->z = 1;
        zbus_chan_finish(&acc_chan);
        zbus_chan_notify(&acc_chan, K_SECONDS(1));
}
```

The following code has the exact behavior of the code in *Reading from a channel*.

```
if (!zbus_chan_claim(&acc_chan, K_MSEC(200))) {
        const struct acc_msg *acc1 = (const struct acc_msg *) zbus_chan_const_msg(&
↪acc_chan);
        // access the acc_msg fields directly.
        zbus_chan_finish(&acc_chan);
}
```

Runtime observer registration It is possible to add observers to channels in runtime. Set the `CONFIG_ZBUS_RUNTIME_OBSERVERS` to enable the feature. This feature uses the heap to allocate the nodes dynamically, a memory slab to allocate the nodes statically, or user-provided nodes. It depends on the `CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC`, which can be `CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_DYNAMIC`, `CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_STATIC`, and `CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_NONE`. The dynamic is the default. When `CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_STATIC` is enabled, you need to set the number of runtime observers you are going to use by setting the `CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_POOL_SIZE` configuration. The following example illustrates the runtime registration usage.

```
ZBUS_LISTENER_DEFINE(my_listener, callback);
// ...

void thread_entry(void) {
        // ...
        /* Adding the observer to channel chan1 */
        zbus_chan_add_obs(&chan1, &my_listener, K_NO_WAIT);
        /* Removing the observer from channel chan1 */
        zbus_chan_rm_obs(&chan1, &my_listener, K_NO_WAIT);
}
```

Warning

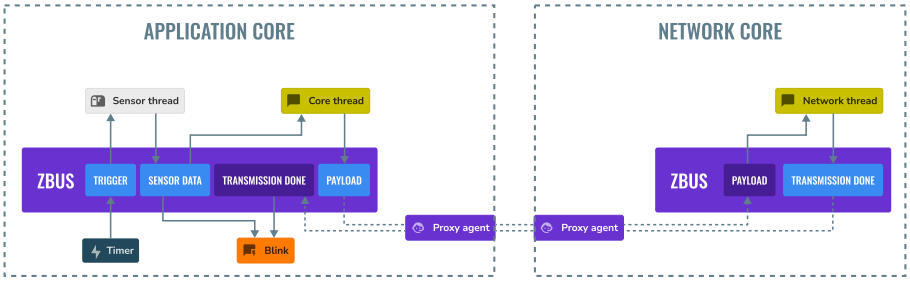
The `zbus_observer_node` can only be reused in `zbus_chan_add_obs_with_node()` after removing the channel observer it was first associated with through `zbus_chan_rm_obs()`.

Proxy Agent Communication (Experimental)

Warning

Proxy agent communication is experimental and may change without deprecation.

ZBus supports proxy agent forwarding, enabling message passing between different execution domains such as CPU cores or separate devices.



Concepts Proxy agent communication introduces several key concepts:

- **Shadow channels**: Read-only channels that mirror channels from other domains
- **Proxy agents**: Background services that synchronize channel data between domains
- **Transport backends**: Communication mechanisms used by proxy agents

Proxy agents are set up in code using `ZBUS_PROXY_AGENT_DEFINE`, specifying the transport backend and configuration parameters. Channels are defined using standard zbus macros (`ZBUS_CHAN_DEFINE` or `ZBUS_CHAN_DEFINE_WITH_ID`) and shadow channels use `ZBUS_SHADOW_CHAN_DEFINE` to create read-only mirrors linked to specific proxy agents.

Transport Backends ZBus proxy agent communication relies on transport backends to forward messages between different execution domains.

IPC Backend The IPC backend forwards messages between CPU cores within the same system using Inter-Process Communication mechanisms.

See the zbus-proxy-agent-ipc sample for a complete implementation.

Usage

1. Enable CONFIG_ZBUS_PROXY_AGENT and the desired backend option.
2. Provide the backend device in devicetree.
3. Instantiate the proxy agent:

```
#include <zephyr/zbus/proxy_agent/zbus_proxy_agent.h>
#include <zephyr/zbus/proxy_agent/zbus_proxy_agent_ipc.h>

#define IPC_DEV_NODE DT_NODELABEL(ipc0)

ZBUS_PROXY_AGENT_DEFINE(proxy_agent,                    /* Proxy agent name */
                        ZBUS_PROXY_AGENT_BACKEND_IPC,   /* Proxy agent type */
                        IPC_DEV_NODE                    /* Backend node */
);
```

Where:

- "proxy_agent": Name of the proxy agent instance
- "ZBUS_PROXY_AGENT_BACKEND_IPC": Transport backend type (IPC in this case)
- "IPC_DEV_NODE": Device tree node for the backend device

4. Forward local channels through the agent:

```
ZBUS_CHAN_DEFINE(my_channel, struct my_msg, NULL, NULL,
                 ZBUS_OBSERVERS_EMPTY, ZBUS_MSG_INIT(0));
ZBUS_PROXY_ADD_CHAN(proxy_agent, my_channel);

zbus_chan_pub(&my_channel, &msg, K_MSEC(100));
```

Any message published to "my_channel" will be forwarded by "proxy_agent" to remote domains.

5. Mirror remote channels with shadows:

```
ZBUS_SHADOW_CHAN_DEFINE(my_channel_shadow, struct my_msg,
                        proxy_agent, NULL, ZBUS_OBSERVERS_EMPTY,
                        ZBUS_MSG_INIT(0));
```

Where the shadow channel can be used like any regular channel, but is read-only. For example, adding a listener:

```
void my_listener_cb(const struct zbus_channel *chan)
{
    const struct my_msg *data = zbus_chan_const_msg(chan);
    printk("Received: %d\n", data->data);
}

ZBUS_LISTENER_DEFINE(my_listener, my_listener_cb);
ZBUS_CHAN_ADD_OBS(my_channel_shadow, my_listener, 0);
```

Samples

For a complete overview of zbus usage, take a look at the samples. There are the following samples available:

- zbus-hello-world illustrates the code used above in action;

- zbus-work-queue shows how to define and use different kinds of observers. Note there is an example of using a work queue instead of executing the listener as an execution option;
- zbus-msg-subscriber illustrates how to use message subscribers;
- zbus-async-listeners illustrates how to use async listeners;
- zbus-dyn-channel demonstrates how to use dynamically allocated exchanging data in zbus;
- zbus-uart-bridge shows an example of sending the operation of the channel to a host via serial;
- zbus-remote-mock illustrates how to implement an external mock (on the host) to send and receive messages to and from the bus;
- zbus-priority-boost illustrates zbus priority boost feature with a priority inversion scenario;
- zbus-runtime-obs-registration illustrates a way of using the runtime observer registration feature;
- zbus-confirmed-channel implements a way to implement a confirmed channel using only subscribers;
- zbus-benchmark implements a benchmark with different combinations of inputs.
- zbus-proxy-agent-ipc demonstrates multi-core communication using IPC proxy agents;

Suggested Uses

Use zbus to transfer data (messages) between threads in one-to-one, one-to-many, and many-to-many synchronously or asynchronously. Choosing the proper observer type is crucial. Use subscribers for scenarios that can tolerate message losses and duplications; when they cannot, use message subscribers (if you need a thread) or listeners (if you need to be lean and fast). In addition to the listener, another asynchronous message processing mechanism (like *message queues*) may be necessary to retain the pending message until it gets processed.

For proxy agent scenarios, use zbus to enable communication across execution boundaries:

- **Multi-core systems**: Use IPC backend proxy agents to coordinate between application and network processors, or distribute workloads across multiple CPU cores.

> **Note**
>
> ZBus can be used to transfer streams from the producer to the consumer. However, this can increase zbus' communication latency. So maybe consider a Pipe a good alternative for this communication topology.

Configuration Options

For enabling zbus, it is necessary to enable the `CONFIG_ZBUS` option.

Related configuration options:

- `CONFIG_ZBUS_PRIORITY_BOOST` zbus Highest Locker Protocol implementation;
- `CONFIG_ZBUS_CHANNELS_SYS_INIT_PRIORITY` determine the `SYS_INIT` priority used by zbus to organize the channels observations by channel;
- `CONFIG_ZBUS_CHANNEL_NAME` enables the name of channels to be available inside the channels metadata. The log uses this information to show the channels' names;
- `CONFIG_ZBUS_OBSERVER_NAME` enables the name of observers to be available inside the channels metadata;
- `CONFIG_ZBUS_PREFER_DYNAMIC_ALLOCATION` instructs zbus to use dynamic allocation for its internals. That can be disabled by the user and tuned later;

- CONFIG_ZBUS_MSG_SUBSCRIBER enables the message subscriber observer type;
- CONFIG_ZBUS_MSG_SUBSCRIBER_BUF_ALLOC_DYNAMIC uses the heap to allocate message buffers;
- CONFIG_ZBUS_MSG_SUBSCRIBER_BUF_ALLOC_STATIC uses the stack to allocate message buffers;
- CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_SIZE the available number of message buffers to be used simultaneously;
- CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_POOL_ISOLATION enables the developer to isolate a pool for the message subscriber for a set of channels; a channel only switches to a dedicated pool once zbus_chan_set_msg_sub_pool() has been called for it;
- CONFIG_ZBUS_MSG_SUBSCRIBER_NET_BUF_STATIC_DATA_SIZE the biggest message of zbus channels to be transported into a message buffer;
- CONFIG_HEAP_MEM_POOL_ADD_SIZE_ZBUS the reserved heap size for ZBus in a whole including message buffer allocation;
- CONFIG_ZBUS_ASYNC_LISTENER enables the async listener observer type;
- CONFIG_ZBUS_RUNTIME_OBSERVERS enables the runtime observer registration;
- CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_DYNAMIC allocate the runtime observers dynamically using the heap;
- CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_STATIC allocate the runtime observers statically using a memory slab;
- CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_POOL_SIZE the amount of enabled runtime observers to statically allocate;
- CONFIG_ZBUS_RUNTIME_OBSERVERS_NODE_ALLOC_NONE use user-provided runtime observers nodes;
- CONFIG_ZBUS_PROXY_AGENT enable proxy agent communication support.

Proxy Agent Configuration Options

- CONFIG_ZBUS_PROXY_AGENT_LOG_LEVEL log level for proxy agent communication;
- CONFIG_ZBUS_PROXY_AGENT_IPC enable IPC backend for proxy agent communication;
- CONFIG_ZBUS_PROXY_AGENT_IPC_LOG_LEVEL log level for IPC backend proxy agent communication;
- CONFIG_ZBUS_PROXY_AGENT_MAX_MESSAGE_SIZE maximum message size for proxy agent channels;
- CONFIG_ZBUS_PROXY_AGENT_MAX_CHANNEL_NAME_SIZE maximum size of channel names in proxy agent communication;
- CONFIG_ZBUS_PROXY_AGENT_INIT_PRIORITY initialization priority for proxy agent setup.
- CONFIG_ZBUS_PROXY_AGENT_WORK_QUEUE_STACK_SIZE stack size for the proxy agent receive work queue thread;
- CONFIG_ZBUS_PROXY_AGENT_WORK_QUEUE_PRIORITY priority for the proxy agent receive work queue thread.
- CONFIG_ZBUS_PROXY_AGENT_RX_QUEUE_DEPTH depth of the proxy agent receive queue for incoming messages from remote domains.

API Reference

Zbus APIs

Related code samples

- BenchmarkingMeasure the time for sending 256KB from a producer to N consumers.
- Confirmed channelUse confirmed zbus channels to ensure all subscribers consume a message.
- Dynamic channelUse zbus channels with dynamically allocated messages.
- Message subscriberUse zbus message subscribers to listen to messages published to channels.
- Remote mock samplePublish to a zbus instance using UART as a bridge.
- Runtime observer registrationUse zbus' runtime observer registration to filter data generated by a producer.
- UART bridgeRedirect channel events to the host over UART.
- Work queueUse a work queue to process zbus messages in various ways.
- zbus Async ListenersDemonstrate the usage of zbus async listeners.
- zbus Hello WorldMake three threads talk to each other using zbus.
- zbus Priority BoostIllustrates zbus priority boost feature with a priority inversion scenario.
- zbus Proxy agent - IPCForward zbus messages between different domains using IPC proxy agents.

4.3.3 Asynchronous Notifications

Zephyr APIs often include *async* functions where an operation is initiated and the application needs to be informed when it completes, and whether it succeeded. Using `k_poll()` is often a good method, but some application architectures may be more suited to a callback notification, and operations like enabling clocks and power rails may need to be invoked before kernel functions are available so a busy-wait for completion may be needed.

This API is intended to be embedded within specific subsystems such as *On-Off Manager* and other APIs that support async transactions. The subsystem wrappers are responsible for extracting operation-specific data from requests that include a notification element, and for invoking callbacks with the parameters required by the API.

A limitation is that this API is not suitable for *System Calls* because:

- `sys_notify` is not a kernel object;
- copying the notification content from userspace will break use of `CONTAINER_OF` in the implementing function;
- neither the spin-wait nor callback notification methods can be accepted from userspace callers.

Where a notification is required for an asynchronous operation invoked from a user mode thread the subsystem or driver should provide a syscall API that uses `k_poll_signal` for notification.

API Reference

Asynchronous Notification APIs

4.3.4 Virtualization

Inter-VM Shared Memory

- *Overview*
- *Support*
- *ivshmem-v2*
- *API Reference*

Overview As Zephyr is enabled to run as a guest OS on Qemu and ACRN it might be necessary to make VMs aware of each other, or aware of the host. This is made possible by exposing a shared memory among parties via a feature called ivshmem, which stands for inter-VM Shared Memory.

The two types are supported: a plain shared memory (ivshmem-plain) or a shared memory with the ability for a VM to generate an interruption on another, and thus to be interrupted as well itself (ivshmem-doorbell).

Please refer to the official Qemu ivshmem documentation for more information.

Support Zephyr supports both versions: plain and doorbell. Ivshmem driver can be built by enabling `CONFIG_IVSHMEM`. By default, this will expose the plain version. `CONFIG_IVSHMEM_DOORBELL` needs to be enabled to get the doorbell version.

Because the doorbell version uses MSI-X vectors to support notification vectors, the `CONFIG_IVSHMEM_MSI_X_VECTORS` has to be tweaked to the number of vectors that will be needed.

Note that a tiny shell module can be exposed to test the ivshmem feature by enabling `CONFIG_IVSHMEM_SHELL`.

ivshmem-v2 Zephyr also supports ivshmem-v2:

https://github.com/siemens/jailhouse/blob/master/Documentation/ivshmem-v2-specification.md

This is primarily used for IPC in the Jailhouse hypervisor (e.g. eth-ivshmem). It is also possible to use ivshmem-v2 without Jailhouse by building the Siemens fork of QEMU, and modifying the QEMU launch flags:

https://github.com/siemens/qemu/tree/wip/ivshmem2

API Reference
Inter-VM Shared Memory

Related code samples

- Inter-VM Shared Memory (ivshmem) EthernetCommunicate with another "cell" in the Jailhouse hypervisor using IVSHMEM Ethernet.
- IVSHMEM doorbellUse Inter-VM Shared Memory to exchange messages between two processes running on different operating systems.

Logging, Tracing & Debugging

4.4 Logging, Tracing & Debugging

This section covers services that help you observe, diagnose, and debug a running system: emitting log messages, tracing execution, inspecting the system through a shell, and monitoring runtime health.

4.4.1 Logging

- *Global Kconfig Options*
- *Usage*
  - *Logging in a module*
  - *Logging in a module instance*
  - *Controlling the logging*
- *Rate-limited logging*
  - *Configuration*
- *Logging panic*
- *Printk*
- *Architecture*
  - *Default Frontend*
  - *Custom Frontend*
  - *Logging strings*
  - *Multi-domain support*
  - *Logging backends*
  - *Dictionary-based Logging*
- *Recommendations and limitations*
- *Benchmark*
- *Stack usage*
- *Logging using ARM Coresight STM*
- *API Reference*

The logging API provides a common interface to process messages issued by developers. Messages are passed through a frontend and then processed by active backends. The frontend is responsible for immediate filtering and queueing of log messages, so it needs to be fast. Backends may run later and take more time. They format the log message and send it to a destination, for example UART, RTT or BLE. A custom frontend and custom backends can be used if needed.

Summary of the logging features:

- Deferred logging reduces the time needed to log a message by shifting time consuming operations to a known context, instead of processing and sending the log message when called.
- Multiple backends supported (up to 9 backends).
- Custom frontend support, which can work together with standard backends.
- Compile time filtering on module level.

- Independent runtime filtering for each backend.
- Additional runtime filtering on module instance level.
- Timestamping with a user provided function. Timestamps can be 32 or 64 bits large.
- Dedicated API for dumping data.
- Dedicated API for handling transient strings.
- Panic support - in panic mode, logging switches to blocking, synchronous processing.
- Printk support - printk messages can be redirected to the logging system.
- Design ready for multi-domain/multi-processor systems.
- Support for logging floating point and `long long` values.
- Built-in copying of transient strings used as arguments.
- Support for multi-domain logging.
- Rate-limited logging macros to prevent log flooding when messages are generated frequently.

The logging API is highly configurable at compile time as well as at runtime. Using Kconfig options (see *Global Kconfig Options*), logs can be gradually removed from compilation to reduce image size and execution time when logs are not needed. During compilation, logs can be filtered out by module and severity level.

Logs can also be compiled in but filtered at runtime using a dedicate API. Runtime filtering is independent for each backend and each source of log messages. The source of a log message can be a module or specific instance of a module.

There are four severity levels available in the system: error, warning, info and debug. For each severity level, the logging API (include/zephyr/logging/log.h) has a set of dedicated macros. There are also macros for logging data.

For each severity level, the following set of macros is available:

- `LOG_X` for standard printf-like messages, e.g. `LOG_ERR`.
- `LOG_HEXDUMP_X` for dumping data, e.g. `LOG_HEXDUMP_WRN`.
- `LOG_INST_X` for standard printf-like messages associated with a particular instance, e.g. `LOG_INST_INF`.
- `LOG_INST_HEXDUMP_X` for dumping data associated with a particular instance, e.g. `LOG_INST_HEXDUMP_DBG`

The warning level also exposes the following additional macro:

- `LOG_WRN_ONCE` for warnings where only the first occurrence is of interest.

Rate-limited logging macros are also available for all severity levels to prevent log flooding:

- `LOG_X_RATELIMIT` for rate-limited standard printf-like messages using the default rate, e.g. `LOG_ERR_RATELIMIT`.
- `LOG_X_RATELIMIT_RATE` for rate-limited standard printf-like messages with a custom rate, e.g. `LOG_ERR_RATELIMIT_RATE`.
- `LOG_HEXDUMP_X_RATELIMIT` for rate-limited data dumping using the default rate, e.g. `LOG_HEXDUMP_WRN_RATELIMIT`.
- `LOG_HEXDUMP_X_RATELIMIT_RATE` for rate-limited data dumping with a custom rate, e.g. `LOG_HEXDUMP_WRN_RATELIMIT_RATE`.

The convenience macros use the default rate specified by `CONFIG_LOG_RATELIMIT_INTERVAL_MS`, while the explicit rate macros take a rate parameter (in milliseconds) that specifies the minimum interval between log messages.

There are two configuration categories: configurations per module and global configuration. When logging is enabled globally, it is enabled for all modules by default. However, modules can disable logging locally. Every module can specify its own logging level. The module must define the `LOG_LEVEL` macro before using the API. Unless a global override is set, the module logging level will be honored. The global override can only increase the logging level. It cannot be used to lower module logging levels that were previously set higher. It is also possible to globally limit logs by providing a maximal severity level present in the system, where maximal means lowest severity (e.g. if maximal level in the system is set to info, it means that errors, warnings and info levels are present, but debug messages are excluded).

Each module which is using the logging must specify its unique name and register itself to the logging. If a module consists of more than one file, then registration is performed in only one file, but each file must specify the module name.

The logger's default frontend is designed to be thread safe and minimizes the time needed to log the message. Time consuming operations like string formatting or access to the transport are not performed by default when the logging API is called. When the logging API is called, a message is created and added to the list. A dedicated, configurable buffer for a pool of log messages is used. There are 2 types of messages: standard and hexdump. Each message contains a source ID (module or instance ID and domain ID, which might be used for multiprocessor systems), timestamp and severity level. A standard message contains pointers to the string and arguments. A hexdump message contains the copied data and string.

Global Kconfig Options

These options can be found in the following file: subsys/logging/Kconfig.

`CONFIG_LOG`: Global switch, turns on/off the logging.

Mode of operations:

`CONFIG_LOG_MODE_DEFERRED`: Deferred mode.

`CONFIG_LOG_MODE_IMMEDIATE`: Immediate (synchronous) mode.

`CONFIG_LOG_MODE_MINIMAL`: Minimal footprint mode.

Filtering options:

`CONFIG_LOG_RUNTIME_FILTERING`: Enables runtime reconfiguration of the filtering.

`CONFIG_LOG_DEFAULT_LEVEL`: Default level, sets the logging level used by modules that are not setting their own logging level.

`CONFIG_LOG_OVERRIDE_LEVEL`: It overrides module logging level when it is not set or set lower than the override value.

`CONFIG_LOG_MAX_LEVEL`: Maximal (lowest severity) level which is compiled in.

Processing options:

`CONFIG_LOG_MODE_OVERFLOW`: When new message cannot be allocated, oldest one are discarded.

`CONFIG_LOG_BLOCK_IN_THREAD`: If enabled and new log message cannot be allocated thread context will block for up to `CONFIG_LOG_BLOCK_IN_THREAD_TIMEOUT_MS` or until log message is allocated.

`CONFIG_LOG_PRINTK`: Redirect printk calls to the logging.

`CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD`: When the number of buffered log messages reaches the threshold, the dedicated thread (see `log_thread_set()`) is woken up. If `CONFIG_LOG_PROCESS_THREAD` is enabled then this threshold is used by the internal thread.

`CONFIG_LOG_PROCESS_THREAD`: When enabled, logging thread is created which handles log processing.

CONFIG_LOG_PROCESS_THREAD_STARTUP_DELAY_MS: Delay in milliseconds after which logging thread is started.

CONFIG_LOG_BUFFER_SIZE: Number of bytes dedicated for the circular packet buffer.

CONFIG_LOG_FRONTEND: Direct logs to a custom frontend.

CONFIG_LOG_FRONTEND_ONLY: No backends are used when messages goes to frontend.

CONFIG_LOG_FRONTEND_OPT_API: Optional API optimized for the most common simple messages.

CONFIG_LOG_CUSTOM_HEADER: Injects an application provided header into log.h

CONFIG_LOG_TIMESTAMP_64BIT: 64 bit timestamp.

CONFIG_LOG_SIMPLE_MSG_OPTIMIZE: Optimizes simple log messages for size and performance. Option available only for 32 bit architectures.

Formatting options:

CONFIG_LOG_FUNC_NAME_PREFIX_ERR: Prepend standard ERROR log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_WRN: Prepend standard WARNING log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_INF: Prepend standard INFO log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_FUNC_NAME_PREFIX_DBG: Prepend standard DEBUG log messages with function name. Hexdump messages are not prepended.

CONFIG_LOG_BACKEND_SHOW_TIMESTAMP: Enables backend to print timestamps with log.

CONFIG_LOG_BACKEND_SHOW_LEVEL: Enables backend to print levels with log.

CONFIG_LOG_BACKEND_SHOW_COLOR: Enables coloring of errors (red) and warnings (yellow).

CONFIG_LOG_BACKEND_FORMAT_TIMESTAMP: If enabled timestamp is formatted to *hh:mm:ss:mmm,uuu*. Otherwise is printed in raw format.

Backend options:

CONFIG_LOG_BACKEND_UART: Enable built-in UART backend.

CONFIG_LOG_BACKEND_NET: Enable built-in Networking backend to send syslog messages to a network server.

Usage

Logging in a module In order to use logging in the module, a unique name for the module must be specified and the module must be registered using LOG_MODULE_REGISTER. Optionally, a compile time log level for the module can be specified as the second parameter. The default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if a custom log level is not provided.

```
#include <zephyr/logging/log.h>
LOG_MODULE_REGISTER(foo, CONFIG_FOO_LOG_LEVEL);
```

If the module consists of multiple files, then LOG_MODULE_REGISTER() should appear in exactly one of them. Each other file should use LOG_MODULE_DECLARE to declare its membership in the module. Optionally, a compile time log level for the module can be specified as the second parameter. The default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if a custom log level is not provided.

```
#include <zephyr/logging/log.h>
/* In all files comprising the module but one */
LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL);
```

In order to use the logging API in a function implemented in a header file, LOG_MODULE_DECLARE macro must be used in the function body before the logging API is called. Optionally, a compile time log level for the module can be specified as the second parameter. The default log level (CONFIG_LOG_DEFAULT_LEVEL) is used if a custom log level is not provided.

```
#include <zephyr/logging/log.h>

static inline void foo(void)
{
     LOG_MODULE_DECLARE(foo, CONFIG_FOO_LOG_LEVEL);

     LOG_INF("foo");
}
```

The dedicated Kconfig template (subsys/logging/Kconfig.template.log_config) can be used to create a local log level configuration.

The example below shows the template being used. As a result, CONFIG_FOO_LOG_LEVEL will be generated:

```
module = FOO
module-str = foo
source "subsys/logging/Kconfig.template.log_config"
```

Logging in a module instance In case there are modules which are multi-instance and instances are widely used across the system, enabling logs will lead to flooding. The logging API provides tools which can be used to filter logs on the instance level rather than the module level. For example, logging can be enabled for a particular instance only.

In order to use instance level filtering, the following steps must be performed:

- A pointer to a specific logging structure is declared in the instance structure. LOG_INSTANCE_PTR_DECLARE is used for that.

```
#include <zephyr/logging/log_instance.h>

struct foo_object {
     LOG_INSTANCE_PTR_DECLARE(log);
     uint32_t id;
}
```

- The module must provide a macro for its instantiation. In that macro, the logging instance is registered and the log instance pointer is initialized in the object structure.

```
#define FOO_OBJECT_DEFINE(_name)                              \
     LOG_INSTANCE_REGISTER(foo, _name, CONFIG_FOO_LOG_LEVEL) \
     struct foo_object _name = {                              \
             LOG_INSTANCE_PTR_INIT(log, foo, _name)           \
     }
```

Note that when logging is disabled, the logging instance and pointer to that instance are not created.

In order to use the instance logging API in a source file, a compile-time log level must be set using LOG_LEVEL_SET.

```
LOG_LEVEL_SET(CONFIG_FOO_LOG_LEVEL);

void foo_init(foo_object *f)
{
```

(continues on next page)

(continued from previous page)

```
    LOG_INST_INF(f->log, "Initialized.");
}
```

In order to use the instance logging API in a header file, a compile-time log level must be set using `LOG_LEVEL_SET`.

```
static inline void foo_init(foo_object *f)
{
    LOG_LEVEL_SET(CONFIG_FOO_LOG_LEVEL);

    LOG_INST_INF(f->log, "Initialized.");
}
```

Controlling the logging By default, log processing in deferred mode is handled internally by a dedicated task which starts automatically. However, it might not be available if multithreading is disabled. It can also be disabled by unsetting `CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD`. In that case, logging can be controlled using the API defined in include/zephyr/logging/log_ctrl.h. Logging must be initialized before it can be used. Optionally, the user can provide a function which returns a timestamp value. If not provided, `k_cycle_get` or `k_cycle_get_32` is used for timestamping. The `log_process()` function is used to process one log message (if pending), and returns true if there are more messages pending. However, it is recommended to use macro wrappers (`LOG_INIT` and `LOG_PROCESS`), which handle the case where logging is disabled.

The following snippet shows how logs can be processed in a simple infinite loop.

```
#include <zephyr/logging/log_ctrl.h>

int main(void)
{
    LOG_INIT();
    /* If multithreading is enabled provide thread id to the logging. */
    log_thread_set(k_current_get());

    while (1) {
            if (LOG_PROCESS() == false) {
                    /* sleep */
            }
    }
}
```

If logs are processed from a thread (user or internal), then it is possible to enable a feature which will wake up the processing thread when a certain number of log messages is buffered (see `CONFIG_LOG_PROCESS_TRIGGER_THRESHOLD`).

Rate-limited logging

Rate-limited logging macros provide a way to prevent log flooding when messages are generated frequently. These macros ensure that log messages are not output more frequently than a specified interval, similar to Linux's `printk_ratelimited` functionality.

The rate-limited logging system provides two types of macros:

Convenience macros (using default rate): - `LOG_ERR_RATELIMIT` - Rate-limited error messages - `LOG_WRN_RATELIMIT` - Rate-limited warning messages - `LOG_INF_RATELIMIT` - Rate-limited info messages - `LOG_DBG_RATELIMIT` - Rate-limited debug messages - `LOG_HEXDUMP_ERR_RATELIMIT` - Rate-limited error hexdump - `LOG_HEXDUMP_WRN_RATELIMIT` - Rate-limited warning hexdump - `LOG_HEXDUMP_INF_RATELIMIT` - Rate-limited info hexdump - `LOG_HEXDUMP_DBG_RATELIMIT` - Rate-limited debug hexdump

Explicit rate macros (with custom rate): - LOG_ERR_RATELIMIT_RATE - Rate-limited error messages with custom rate - LOG_WRN_RATELIMIT_RATE - Rate-limited warning messages with custom rate - LOG_INF_RATELIMIT_RATE - Rate-limited info messages with custom rate - LOG_DBG_RATELIMIT_RATE - Rate-limited debug messages with custom rate - LOG_HEXDUMP_ERR_RATELIMIT_RATE - Rate-limited error hexdump with custom rate - LOG_HEXDUMP_WRN_RATELIMIT_RATE - Rate-limited warning hexdump with custom rate - LOG_HEXDUMP_INF_RATELIMIT_RATE - Rate-limited info hexdump with custom rate - LOG_HEXDUMP_DBG_RATELIMIT_RATE - Rate-limited debug hexdump with custom rate

The convenience macros use the default rate specified by CONFIG_LOG_RATELIMIT_INTERVAL_MS (5000ms by default). The explicit rate macros take a rate parameter (in milliseconds) that specifies the minimum interval between log messages. The rate limiting is per-macro-call-site, meaning that each unique call to a rate-limited macro has its own independent rate limit.

Example usage:

```
#include <zephyr/logging/log.h>
#include <zephyr/kernel.h>

LOG_MODULE_REGISTER(my_module, CONFIG_LOG_DEFAULT_LEVEL);

void process_data(void)
{
    /* Convenience macros using default rate (CONFIG_LOG_RATELIMIT_INTERVAL_MS) */
    LOG_WRN_RATELIMIT("Data processing warning: %d", error_code);
    LOG_ERR_RATELIMIT("Critical error occurred: %s", error_msg);
    LOG_INF_RATELIMIT("Processing status: %d items", item_count);
    LOG_HEXDUMP_WRN_RATELIMIT(data_buffer, data_len, "Data buffer:");

    /* Explicit rate macros with custom intervals */
    LOG_WRN_RATELIMIT_RATE(1000, "Fast rate warning: %d", error_code);
    LOG_ERR_RATELIMIT_RATE(30000, "Slow rate error: %s", error_msg);
    LOG_INF_RATELIMIT_RATE(2000, "Custom rate status: %d items", item_count);
    LOG_HEXDUMP_ERR_RATELIMIT_RATE(5000, data_buffer, data_len, "Error data:");
}
```

Rate-limited logging is particularly useful for:

- Error conditions that might occur frequently but don't need to flood the logs
- Status updates in tight loops or high-frequency callbacks
- Debug information that could overwhelm the logging system
- Network or I/O operations that might fail repeatedly

Configuration Rate-limited logging can be configured using the following Kconfig options:

- CONFIG_LOG_RATELIMIT - Master switch to enable/disable rate-limited logging
- CONFIG_LOG_RATELIMIT_INTERVAL_MS - Default interval for convenience macros (5000ms)

When CONFIG_LOG_RATELIMIT is disabled, the behavior of rate-limited macros is controlled by the CONFIG_LOG_RATELIMIT_FALLBACK choice:

- CONFIG_LOG_RATELIMIT_FALLBACK_LOG - All rate-limited macros behave as regular logging macros
- CONFIG_LOG_RATELIMIT_FALLBACK_DROP - All rate-limited macros expand to no-ops (default)

This allows you to control whether rate-limited log macros should always print or be completely suppressed when rate limiting is not available.

The rate limiting is implemented using static variables and k_uptime_get_32() to track the last log time for each call site.

Logging panic

In case of an error condition, the system usually can no longer rely on the scheduler or interrupts. In that situation, deferred log message processing is not an option. The logger controlling API provides a function for entering into panic mode (`log_panic()`), which should be called in such a situation.

When `log_panic()` is called, a *panic* notification is sent to all active backends. Once all backends are notified, all buffered messages are flushed. From that moment forward, all logs are processed in a blocking way.

Printk

Typically, logging and `printk()` use the same output, which they compete for. This can lead to issues if the output does not support preemption, but it may also result in corrupted output because logging data is interleaved with printk data. However, it is possible to redirect printk messages to the logging subsystem by enabling `CONFIG_LOG_PRINTK`. In that case, printk entries are treated as log messages with level 0 (they cannot be disabled). When enabled, logging manages the output, so there is no interleaving. However, in deferred mode, the printk behaviour is changed since the output is delayed until the logging thread processes the data. `CONFIG_LOG_PRINTK` is enabled by default.

Architecture

Logging consists of 3 main parts:

- Frontend
- Core
- Backends

A log message is generated by a source of logging, which can be a module or an instance of a module.

Default Frontend The default frontend is engaged when the logging API is called by a source of logging (e.g. `LOG_INF`) and is responsible for filtering a message (compile and runtime), allocating a buffer for the message, creating the message and committing that message. Since the logging API can be called in an interrupt, the frontend is optimized to log the message as fast as possible.

Log message A log message contains a message descriptor (source, domain and level), timestamp, formatted string details (see *Cbprintf Packaging*) and optional data. Log messages are stored in a continuous block of memory. Memory is allocated from a circular packet buffer (*Multi Producer Single Consumer Packet Buffer*), which has a few consequences:

- Each message is a self-contained, continuous block of memory. Thus, it is suited for copying the message (e.g. for offline processing).
- Messages must be freed sequentially. Backend processing is synchronous. Backends can make a copy for deferred processing.

A log message has the following format:

Message Header	2 bits: MPSC packet buffer header
	1 bit: Trace/Log message flag
	3 bits: Domain ID
	3 bits: Level
	11 bits: Cbprintf Package Length
	12 bits: Data length
	pointer: Pointer to the source descriptor[1]
	32 or 64 bits: Timestamp[1]
	pointer: Pointer to thread ID (optional)[2]
	8 bits: core ID (optional)[3]
	Optional padding[4]
Cbprintf package (optional)	Header
	Arguments
	Appended strings
Hexdump data (optional)	
Alignment padding (optional)	

Log message allocation It may happen that the frontend cannot allocate a message. This happens if the system is generating more log messages than it can process in a certain time frame. There are two strategies to handle that case:

- No overflow - the new log is dropped if space for a message cannot be allocated.
- Overflow - the oldest pending messages are freed, until the new message can be allocated. Enabled by `CONFIG_LOG_MODE_OVERFLOW`. Note that it degrades performance thus it is recommended to adjust buffer size and amount of enabled logs to limit dropping.

Runtime filtering If runtime filtering is enabled, then for each source of logging a filter structure in RAM is declared. Such a filter is using 32 bits divided into ten 3 bit slots. Except for *slot 0*, each slot stores the current filter for one backend in the system. *Slot 0* (bits 0-2) is used to aggregate the maximal filter setting for a given source of logging. The aggregate slot determines if a log message is created for a given entry, since it indicates if there is at least one backend expecting that log entry. Backend slots are examined when a message is processed by the core to determine if the message is accepted by the given backend. Compared to compile time filtering, the binary footprint is increased, because discarded logs are still compiled in.

In the example below, backend 1 is set to receive errors (*slot 1*) and backend 2 up to info level (*slot 2*). Slots 3-9 are not used. The aggregated filter (*slot 0*) is set to info level, meaning that up to this level, message from that particular source will be buffered.

slot 0	slot 1	slot 2	slot 3	...	slot 9
INF	ERR	INF	OFF	...	OFF

Custom Frontend Custom frontend is enabled using `CONFIG_LOG_FRONTEND`. Logs are directed to functions declared in include/zephyr/logging/log_frontend.h. If option `CONFIG_LOG_FRONTEND_ONLY` is enabled then log message is not created and no backend is handled. Otherwise, custom frontend can coexist with backends.

[1] Depending on their size, the source descriptor and timestamp fields may be swapped, in order to reduce the amount of padding.
[2] Only present if CONFIG_LOG_THREAD_ID_PREFIX is enabled.
[3] Only present if CONFIG_LOG_CORE_ID_PREFIX is enabled.
[4] It may be required for cbprintf package alignment.

In some cases, logs need to be redirected at the macro level. For these cases, `CONFIG_LOG_CUSTOM_HEADER` can be used to inject an application provided header named `zephyr_custom_log.h` at the end of include/zephyr/logging/log.h.

Frontend using ARM Coresight STM (System Trace Macrocell) For more details about logging using ARM Coresight STM see *Multi-domain logging using ARM Coresight STM*.

Logging strings String arguments are handled by *Cbprintf Packaging*. See *Limitations and recommendations* for limitations and recommendations.

Multi-domain support More complex systems can consist of multiple domains where each domain is an independent binary. Examples of domains are a core in a multicore SoC or one of the binaries (Secure or Nonsecure) on an ARM TrustZone core.

Tracing and debugging on a multi-domain system is more complex and requires an efficient logging system. Two approaches can be used to structure this logging system:

- Log inside each domain independently. This option is not always possible as it requires that each domain has an available backend (for example, UART). This approach can also be troublesome to use and not scalable, as logs are presented on independent outputs.
- Use a multi-domain logging system where log messages from each domain end up in one root domain, where they are processed exactly as in a single domain case. In this approach, log messages are passed between domains using a connection between domains created from the backend on one side and linked to the other.

  The Log link is an interface introduced in this multi-domain approach. The Log link is responsible for receiving any log message from another domain, creating a copy, and putting that local log message copy (including remote data) into the message queue. This specific log link implementation matches the complementary backend implementation to allow log messages exchange and logger control like configuring filtering, getting log source names, and so on.

There are three types of domains in a multi-domain system:

- The *end domain* has the logging core implementation and a cross-domain backend. It can also have other backends in parallel.
- The *relay domain* has one or more links to other domains but does not have backends that output logs to the user. It has a cross-domain backend either to another relay or to the root domain.
- The *root domain* has one or multiple links and a backend that outputs logs to the user.

See the following image for an example of a multi-domain setup:

In this architecture, a link can handle multiple domains. For example, let's consider an SoC with two ARM Cortex-M33 cores with TrustZone: cores A and B (see the example illustrated above). There are four domains in the system, as each core has both a Secure and a Nonsecure domain. If *core A nonsecure* (A_NS) is the root domain, it has two links: one to *core A secure* (A_NS-A_S) and one to *core B nonsecure* (A_NS-B_NS). *B_NS* domain has one link, to *core B secure B_NS-B_S*), and a backend to *A_NS*.

Since in all instances there is a standard logging subsystem, it is always possible to have multiple backends and simultaneously output messages to them. An example of this is shown in the illustration above as a dotted UART backend on the *B_NS* domain.

Domain ID The source of each log message can be identified by the following fields in the header: `source_id` and `domain_id`.

The value assigned to the `domain_id` is relative. Whenever a domain creates a log message, it sets its `domain_id` to `0`. When a message crosses the domain, `domain_id` changes as it is increased

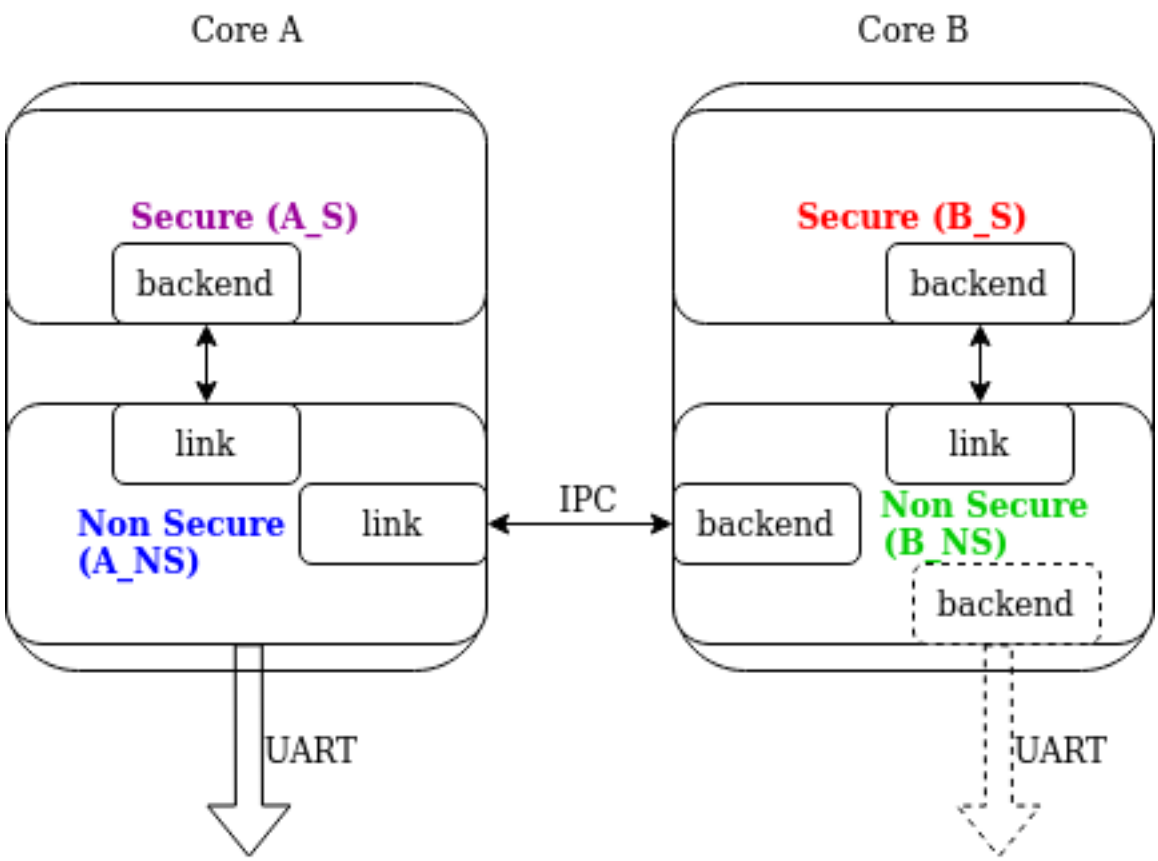


Fig. 36: Multi-domain example

by the link offset. The link offset is assigned during the initialization, where the logger core is iterating over all the registered links and assigned offsets.

The first link has the offset set to 1. The following offset equals the previous link offset plus the number of domains in the previous link.

The following example is shown below, where the assigned `domain_ids` are shown for each domain:

Let's consider a log message created on the *B_S* domain:

1. Initially, it has its `domain_id` set to `0`.
2. When the *B_NS-B_S* link receives the message, it increases the `domain_id` to 1 by adding the *B_NS-B_S* offset.
3. The message is passed to *A_NS*.
4. When the *A_NS-B_NS* link receives the message, it adds the offset (2) to the `domain_id`. The message ends up with the `domain_id` set to 3, which uniquely identifies the message originator.

Cross-domain log message In most cases, the address space of each domain is unique, and one domain cannot access directly the data in another domain. For this reason, the backend can partially process the message before it is passed to another domain. Partial processing can include converting a string package to a *fully self-contained* version (copying read-only strings to the package body).

Each domain can have a different timestamp source in terms of frequency and offset. Logging does not perform any timestamp conversion.

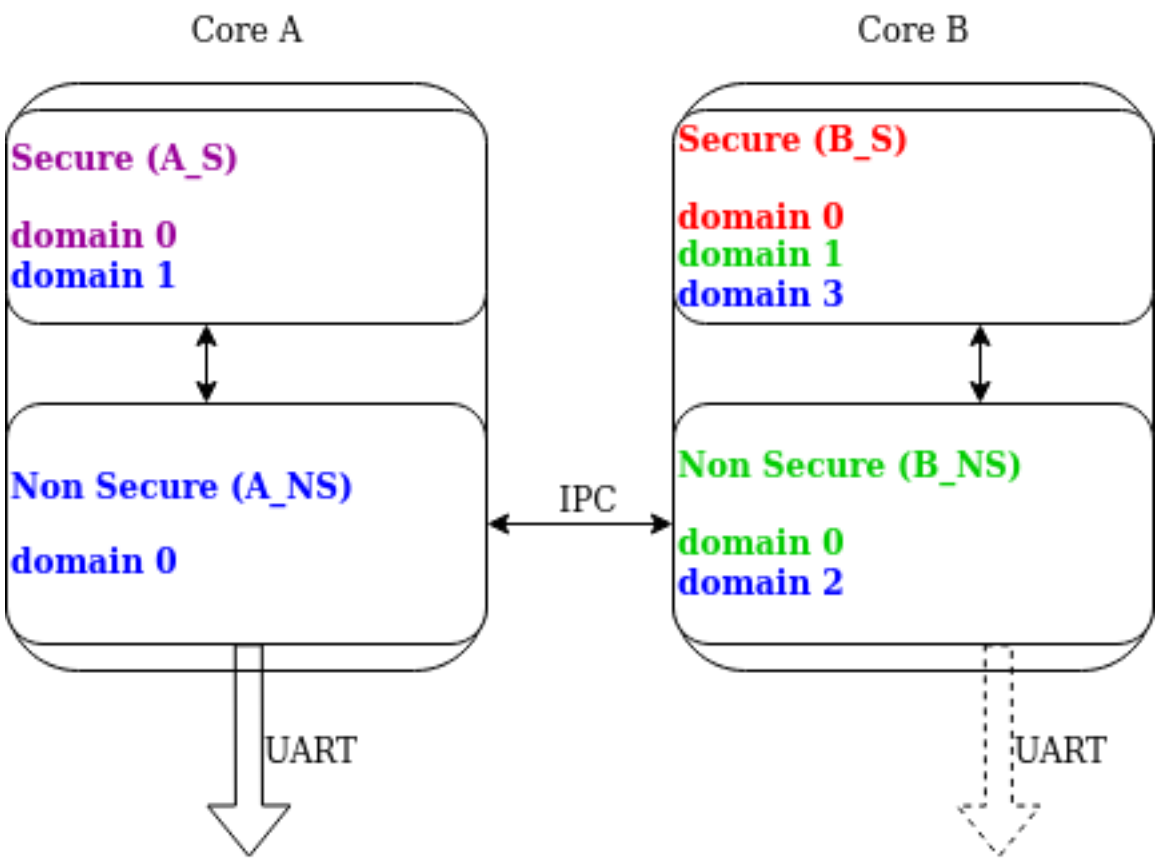


Fig. 37: Domain IDs assigning example

Runtime filtering In the single-domain case, each log source has a dedicated variable with runtime filtering for each backend in the system. In the multi-domain case, the originator of the log message is not aware of the number of backends in the root domain.

As such, to filter logs in multiple domains, each source requires a runtime filtering setting in each domain on the way to the root domain. As the number of sources in other domains is not known during the compilation, the runtime filtering of remote sources must use dynamically allocated memory (one word per source). When a backend in the root domain changes the filtering of the module from a remote domain, the local filter is updated. After the update, the aggregated filter (the maximum from all the local backends) is checked and, if changed, the remote domain is informed about this change. With this approach, the runtime filtering works identically in both multi-domain and single-domain scenarios.

Message ordering Logging does not provide any mechanism for synchronizing timestamps across multiple domains:

- If domains have different timestamp sources, messages will be processed in the order of arrival to the buffer in the root domain.
- If domains have the same timestamp source or if there is an out-of-bound mechanism that recalculates timestamps, there are 2 options:
  - Messages are processed as they arrive in the buffer in the root domain. Messages are unordered but they can be sorted by the host as the timestamp indicates the time of the message generation.
  - Links have dedicated buffers. During processing, the head of each buffer is checked and the oldest message is processed first.

    With this approach, it is possible to maintain the order of the messages at the cost of a

suboptimal memory utilization (since the buffer is not shared) and increased processing latency (see CONFIG_LOG_PROCESSING_LATENCY_US).

Logging backends Logging backends are registered using LOG_BACKEND_DEFINE. The macro creates an instance in the dedicated memory section. Backends can be dynamically enabled (log_backend_enable()) and disabled. When *Runtime filtering* is enabled, log_filter_set() can be used to dynamically change filtering of a module's logs for a given backend. A module is identified by source ID and domain ID. The source ID can be retrieved if the source name is known by iterating through all registered sources.

Logging supports up to 9 concurrent backends. A log message is passed to each backend in the processing phase. Additionally, a backend is notified when logging enters panic mode with log_backend_panic(). When that happens, the backend should switch to synchronous, interrupt-less operation or shut itself down if that is not supported. Occasionally, logging may inform the backend about the number of dropped messages with log_backend_dropped(). The message processing API is version specific.

log_backend_msg_process() is used for processing a message. It is common for standard and hexdump messages, because a standard message holds its formatting arguments in the same place where a hexdump message holds its data. It is also common for deferred and immediate logging.

Message formatting Logging provides a set of functions that can be used by the backend to format a message. Helper functions are available in include/zephyr/logging/log_output.h.

Example message formatted using log_output_msg_process():

```
[00:00:00.000,274] <info> sample_instance.inst1: logging message
```

Dictionary-based Logging Dictionary-based logging, instead of human readable text, outputs the log messages in binary format. This binary format encodes arguments to formatted strings in their native storage formats which can be more compact than their text equivalents. For statically defined strings (including the format strings and any string arguments), references to the ELF file are encoded instead of the whole strings. A dictionary created at build time contains the mappings between these references and the actual strings. This allows the offline parser to obtain the strings from the dictionary to parse the log messages. This binary format allows a more compact representation of log messages in certain scenarios. However, this requires the use of an offline parser and is not as intuitive to use as text-based log messages.

Note that long double is not supported by Python's struct module. Therefore, log messages with long double will not display the correct values.

Configuration Here are kconfig options related to dictionary-based logging:

- CONFIG_LOG_DICTIONARY_SUPPORT enables dictionary-based logging support. This should be selected by the backends which require it.
- The UART backend can be used for dictionary-based logging. These are additional config for the UART backend:
  - CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_HEX tells the UART backend to output hexadecimal characters for dictionary based logging. This is useful when the log data needs to be captured manually via terminals and consoles.
  - CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_BIN tells the UART backend to output binary data.
- The RTT backend can also be used for dictionary-based logging:
  - CONFIG_LOG_BACKEND_RTT enables the RTT backend.

- CONFIG_LOG_BACKEND_RTT_OUTPUT_DICTIONARY enables dictionary-based output for the RTT backend. Use together with CONFIG_USE_SEGGER_RTT.
- CONFIG_LOG_BACKEND_RTT_OUTPUT_DICTIONARY_HEX tells the RTT backend to output hexadecimal characters for dictionary based logging.

Usage When dictionary-based logging is enabled via enabling related logging backends, a JSON database file, named log_dictionary.json, will be created in the build directory. This database file contains information for the parser to correctly parse the log data. Note that this database file only works with the same build, and cannot be used for any other builds.

Offline Parsing To parse a previously captured log file:

```
./scripts/logging/dictionary/log_parser.py <build dir>/log_dictionary.json <log␣
↪data file>
```

The parser takes two required arguments, where the first one is the full path to the JSON database file, and the second part is the file containing log data. Add an optional argument --hex to the end if the log data file contains hexadecimal characters (e.g. when CONFIG_LOG_BACKEND_UART_OUTPUT_DICTIONARY_HEX=y). This tells the parser to convert the hexadecimal characters to binary before parsing.

Live Parsing For real-time decoding of dictionary-based log output, use the live log parser. It connects to a running device and continuously decodes binary log data as it arrives. Note that the live parser only supports binary dictionary output (not hex-encoded). The live parser supports three input modes:

Serial (UART):

```
./scripts/logging/dictionary/live_log_parser.py <build dir>/log_dictionary.json␣
↪serial <port> <baudrate>
```

For example, to read from /dev/ttyACM0 at 115200 baud:

```
./scripts/logging/dictionary/live_log_parser.py build/zephyr/log_dictionary.json␣
↪serial /dev/ttyACM0 115200
```

JLink RTT:

```
./scripts/logging/dictionary/live_log_parser.py <build dir>/log_dictionary.json␣
↪jlink-rtt <device_name>
```

For example, to read RTT output from an nRF5340:

```
./scripts/logging/dictionary/live_log_parser.py build/zephyr/log_dictionary.json␣
↪jlink-rtt nrf5340_xxaa_app
```

The JLink RTT mode requires the pylink-square Python package (pip install pylink-square). Optional arguments include --channel to select the RTT channel (default: 0), --speed to set the connection speed, and --block-address to specify the RTT control block address in hex.

File / stdin:

```
./scripts/logging/dictionary/live_log_parser.py <build dir>/log_dictionary.json␣
↪file <filepath>
```

When <filepath> is omitted, the parser reads from stdin, which allows piping binary data directly into it.

Please refer to the logging-dictionary sample for more examples on using the log parsers.

Recommendations and limitations

Consider the following recommendations:

- Enable `CONFIG_LOG_SPEED` to slightly speed up deferred logging at the cost of a slight increase in memory footprint.
- It is recommended to cast a pointer to `const char *` when it is used with a `%s` format specifier and it points to a constant string.
- It is recommended to cast a pointer to `char *` when it is used with a `%s` format specifier and it points to a transient string.
- It is required to cast a character pointer to non character pointer (e.g., `void *`) when it is used with a `%p` format specifier.

```
LOG_WRN("%s", str);
LOG_WRN("%p", (void *)str);
```

Consider the following limitations:

- Logging does not support string format specifiers with width (e.g., `%.*s` or `%8s`). That is because format string content is not used to build a log message, only argument types.
- If deferred logging is used and log messages are prefixed with the thread name (Kconfig option `CONFIG_LOG_THREAD_ID_PREFIX=y` and `CONFIG_THREAD_NAME=y`), it is assumed that the corresponding `k_thread` structure is still valid when the log message is formatted. This can be an issue when that structure is allocated dynamically, using `k_malloc()` or `malloc()` for instance. In this case, if the thread logs some messages and then gets stopped and its `struct k_thread` is freed, the log system will still try to access that structure when handling the message later. This creates a use-after-free scenario. To avoid this, a solution consists in calling `log_flush()` before freeing the structure.

```
struct k_thread *thread = k_malloc(sizeof(*thread)); /* struct allocated␣
↪dynamically */
k_thread_create(thread, ...);
k_thread_name_set(thread, "foobar");

/* Thread calls LOG_*(...) */

k_thread_join(thread, K_FOREVER);
log_flush();  /* flush log buffer before freeing the struct k_thread */
k_free(thread); /* avoid a potential use-after-free scenario if deferred logging␣
↪is used */
```

Benchmark

Benchmark numbers from tests/subsys/logging/log_benchmark performed on `qemu_x86`. It is a rough comparison to give a general overview.

Feature	
Kernel logging	7us[5]/11us
User logging	13us
kernel logging with overwrite	10us[Page 1221, 5]/15us
Logging transient string	42us
Logging transient string from user	50us
Memory utilization[6]	518
Memory footprint (test)[7]	2k
Memory footprint (application)[8]	3.5k
Message footprint[9]	47[Page 1221, 5]/32 bytes

Benchmark details

Stack usage

When logging is enabled, it impacts the stack usage of the context that uses the logging API. If the stack is optimized, it may lead to stack overflow. Stack usage depends on mode and optimization. It also significantly varies between platforms. In general, when `CONFIG_LOG_MODE_DEFERRED` is used, stack usage is smaller, since logging is limited to creating and storing log messages. If `CONFIG_LOG_MODE_IMMEDIATE` is used, the log message is processed by the backends, which includes string formatting, directly in the context that uses the logging API. In case of that mode, the stack usage will depend on which backends are used.

Some of the platforms characterization for log message with two `integer` arguments listed below:

Platform	Deferred	Deferred (no optimization)	Immediate	Immediate (no optimization)
ARM Cortex-M3	40	152	412	783
x86	12	224	388	796
riscv32	24	208	456	844
xtensa	72	336	504	944
x86_64	32	528	1088	1440

Logging using ARM Coresight STM

For logging on NRF54H20 using ARM Coresight STM see *Multi-domain logging using ARM Coresight STM*.

API Reference

Logging API

Related code samples

- Bluetooth logging backendSend log messages over Bluetooth using the Bluetooth logging backend.
- Coresight STM benchmark
- Coresight STM with remote shell
- Dictionary-based loggingOutput binary log data using the dictionary-based logging API.
- LoggingOutput log messages to the console using the logging subsystem.

Logger control API

Related code samples

- LoggingOutput log messages to the console using the logging subsystem.

[5] `CONFIG_LOG_SPEED` enabled.
[6] Number of log messages with various number of arguments that fits in 2048 bytes dedicated for logging.
[7] Logging subsystem memory footprint in tests/subsys/logging/log_benchmark where filtering and formatting features are not used.
[8] Logging subsystem memory footprint in samples/subsys/logging/logger.
[9] Average size of a log message (excluding string) with 2 arguments on `Cortex M3`

- Logging in multi-domain environmentUse the multidomain logging feature to aggregate log messages from multiple cores into a single console output via the IPC service.
- Remote syslogEnable a remote syslog service that sends syslog messages to a remote server

Log messages

Logger backends

Related code samples

- Bluetooth logging backendSend log messages over Bluetooth using the Bluetooth logging backend.
- Remote syslogEnable a remote syslog service that sends syslog messages to a remote server

Log output

Multi-domain logging using ARM Coresight STM The Arm CoreSight SoC-400 is a comprehensive library of components for the creation of debug and trace functionality within a system. The STM (System Trace Macrocell) is a trace source that is integrated into a CoreSight system, designed primarily for high-bandwidth trace of instrumentation embedded into software. This instrumentation is made up of memory-mapped writes to the STMESP (STM Extended Stimulus Port) peripheral registers. Multiple cores can share and directly access the STM without being aware of each other. Each core has 65536 stimulus ports (identical register sets) which can be independently accessed thus there is no need for locking when various contexts want to log the data.

STM Extended Stimulus Port (STMESP) Local domain (each core) has access to its own set of STMESP peripherals. Each set has registers for writing data (with or without timestamp and with or without marking) and writing a flag. Each write to STMESP register results in the user data being encoded into MIPI System Trace Protocol v2 (STPv2). Hardware manages multiplexing of data from different STMESP register sets and from different cores by adding Major and Channel opcodes into the stream when necessary. Timestamp (from common source for all cores) and synchronization opcodes are autonomously added by the STM.

Data stream generated by STM can end up in various sinks. For example it can be directly send to TPIU (Trace Port Interface Unit) or saved in the RAM Circular Buffer which is called ETR (Embedded Trace Router). TPIU is a 5 pin interface (4 data pins and clock) and external tool is required to capture the data (e.g. J-Trace PRO). When ETR is used then one core in the system is responsible for processing that data, for example dumping it over UART to the host.

ARM Coresight Embedded Trace Router (ETR) ETR is a circular RAM buffer to which tracing data is saved. Since it may contain data from various sources additional wrapping protocol is used to multiplex them. `Coresight Trace Formatter` is used for that. Data is encoded into 16 bytes frames which holds up to 15 byte of data. Circular RAM buffer size is limited to 4k thus there is a risk of data overflow. If overflow occurs then data is lost but due to presence of synchronization opcodes in the STPv2 data stream synchronization can be restored.

Data from ETR is handled on the device. It can be forwarded as-is to the host (e.g. using UART), where a host tool is decoding the data or data can be decoded on-chip to output the data in human readable format.

Nordic Semiconductor NRF54H20 implementation instead of using ETR peripheral for getting buffer busyness is using own wrapper: Trace Buffer Monitor (TBM).

STMESP Logging frontend *Custom Frontend* API is implemented. Custom frontend assumes that all required operations are executed in the context of the logging macro call. It is also assumed that frontend provides the timestamp. Frontend utilizes a fact that there are multiple STMESP register sets and use multiple sets to avoid locking context. The atomically incremented channel counter is used to pick unique STMESP set and whole message is written to that STMESP. Limited pool of sets is used leaving most of channels for other purposes.

Early logging STMESP cannot be used before STM or ETR/TPIU is setup. In order to support early logging, before infrastructure is ready a dedicated RAM buffer is used to which data is written. When STM infrastructure readiness is notified (`log_frontend_stmesp_etr_ready()`) then RAM buffer content is written to STMESP. Early logging is applicable only to the core which owns and configures Coresight infrastructure (e.g. in case of NRF54H20 it is the Secure core).

Trace points There may be cases when logging is too slow (even though it is very fast). For cases like that a dedicated `trace point` API is added. It is extremely fast because it is just a single write to one STMESP register. There are 2 functions in the API:

- `log_frontend_stmesp_tp()` - It accepts single argument - index. Index is between 0 and 65280.
- `log_frontend_stmesp_tp_d32()` - It accepts two arguments - index and user data. Index is between 0 and 65280. User data is 32 bit word.

It takes less than 100 ns to log a single trace point on NRF54H20 and it is approx. 7 times faster than the fastest log message.

Logging using STM STM logging has 2 modes of operations:

- Dictionary-based - assisted mode which is using dictionary-based logging. In this mode logging strings can be removed from the binary because they are only needed for decoding and decoding is performed by the host tool. This mode takes less memory and is faster (2-3 times faster). There is less data written so there is less chance of data overflow in ETR buffer.
- Standalone - data is decoded on chip and human-readable strings are printed. This mode requires more device resources and processing power.

Following figure presents multi-domain logging using ARM Coresight STM.

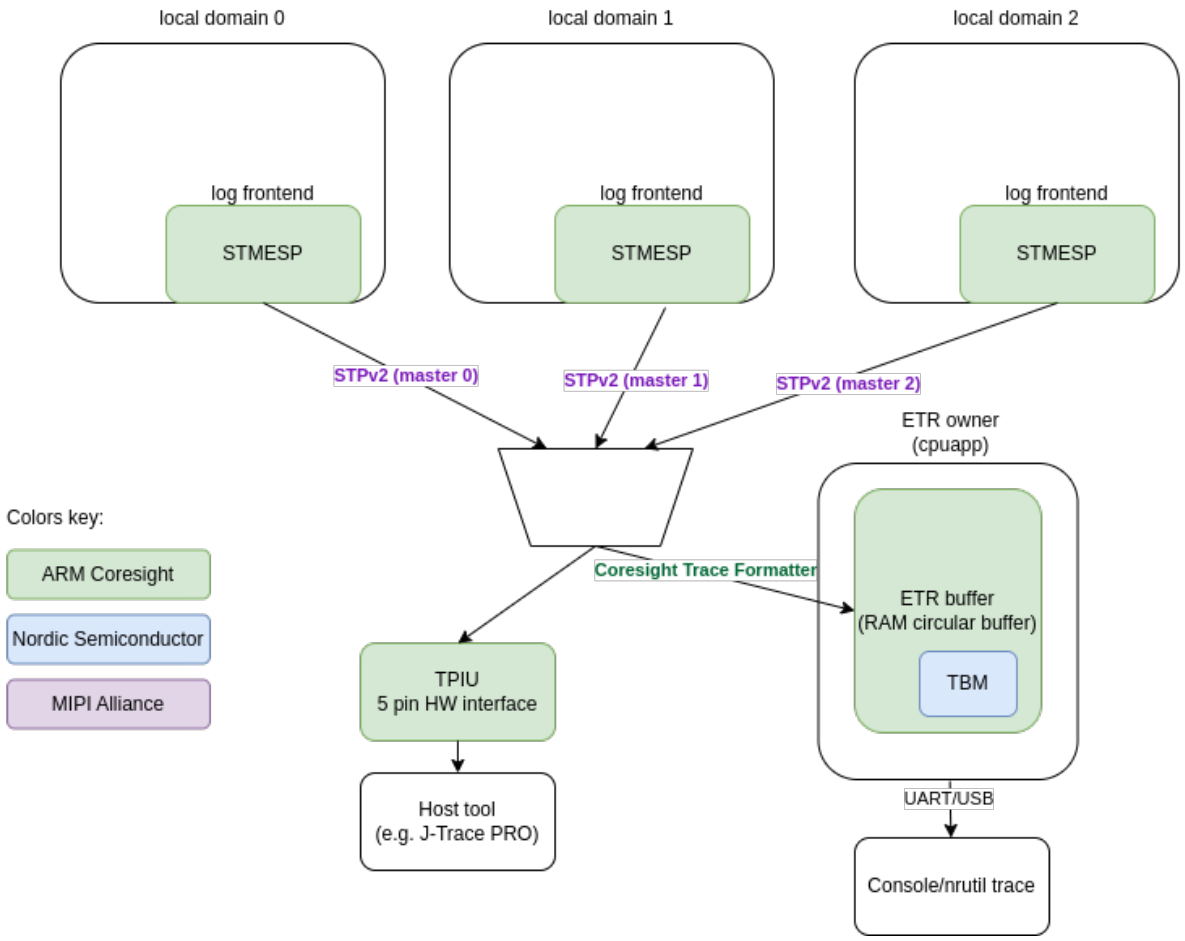
Each core (local domain) is using logging frontend which writes logging data to STMESP registers.

If TPIU is used as the output then there is no more software components needed. STPv2 stream of data is composed by STM and send over TPIU.

If ETR RAM buffer is used then buffer owner core (`proxy`) is responsible for handling that data. If dictionary-based logging is used then proxy just sends data as-is over UART. If standalone logging is used then proxy is using *ARM Coresight Trace Deformatter* and *MIPI STP Decoder* to decode data and demultiplex messages. Messages are formatted to human readable strings using logging *Message formatting*.

Dictionary-based logging Assisted multicore logging uses dictionary-based logging to send messages without redundant strings to STM, and is based on the *Dictionary-based Logging* feature of the logging API provided by Zephyr. Instead of including the format strings in the log messages, it logs the addresses (message IDs) where the strings are stored, which reduces the size of the logging subsystem. If the data goes to the ETR buffer, the proxy core's responsibility is to dump this data. The host PC, equipped with a decoder tool, translates these addresses back into human-readable text using a JSON database generated during the build process.

When using logs, this method has the following advantages:


local domain 0
local domain 1
local domain 2
log frontend
STMESP
log frontend
STMESP
log frontend
STMESP
STPv2 (master 0)
STPv2 (master 1)
STPv2 (master 2)
ETR owner
(cpuapp)
Colors key:
ARM Coresight
Nordic Semiconductor
MIPI Alliance
Coresight Trace Formatter
ETR buffer
(RAM circular buffer)
TBM
TPIU
5 pin HW interface
Host tool
(e.g. J-Trace PRO)
UART/USB
Console/nrutil trace

- It reduces the size of the binary, as the strings used in the log messages are not stored in the binary itself. Logging infrastructure is also very limited. On local domains it is just the frontend. There is even no need to have string formatting function.
- It reduces the amount of data that needs to be sent to and processed by the application core, as the string formatting is offloaded to the host side.
- Logging is fast. It takes less than 1 us to log a simple message (up to 2 arguments) on NRF54H20.

Proxy core is using Nordic specific peripheral (TBM) to get ETR buffer busyness and send data over UART. Nordic specific driver for ETR buffer is located in drivers/debug/debug_nrf_etr.c.

Configuration For Nordic SoC a dedicated snippet (*Nordic Dictionary-based STM logging snippet (nordic-log-stm-dict)*) shall be used to enable logging. Each core shall be build with that snippet. If any core wants to use it then application core is required to have it enabled too since it acts as the proxy (ETR buffer processing). All cores must use the same configuration of the logging.

Reading the logs To read the dictionary-based STM log output, do the following:

1. Set up the log capture.

   Use the `nrfutil trace stm` command to start capturing logs from the device, specifying the database configuration for each domain ID, as well as the serial port, the baud rate, and the output file name:

   ```
   nrfutil trace stm --database-config␣
   ↪34:build/zephyr/log_dictionary.json,35:build_rad/zephyr/log_dictionary.json␣
   ↪--input-serialport /dev/ttyACM1 --baudrate 115200 --output-ascii out.txt
   ```

2. Capture and decode the logs.

   nrfutil will capture the log data from the specified UART port and use the provided dictionary databases to decode the logs into a human-readable format. The decoded logs will be saved in the specified output file (the `out.txt` file in the previous example).

3. Open the output file to review the decoded log messages.

   The file will contain timestamps and the log messages in a format that is human-readable.

If the log capture fails to find a sync, rerun the capture process.

> **Note**
>
> Decoding artifacts or incorrect timestamps might occur when rerunning the process.

Each log line contains a domain-related or core-related prefix between the log level and the module name, indicating the core that generated the log entry. The following are the prefixes used to indicate the cores:

Table 46: nRF54H20 log prefixes

Core	Prefix	ID
Secure Domain	`sec`	0x21
Application core	`app`	0x22
Radio core	`rad`	0x23
System Controller (SysCtrl)	`sys`	0x2c
Fast Lightweight Processor (FLPR)	`flpr`	0x2d
Peripheral Processor (PPR)	`ppr`	0x2e
	`mod`	0x24

Stand-alone logging Frontend is writing to STMESP registers. Message format is aligned with the on chip decoder in subsys/logging/frontends/log_frontend_stmesp_demux.c.

`Proxy` is using Nordic specific peripheral (TBM) to get ETR buffer busyness and read and decode data and send human-readable data over UART. Nordic specific driver for ETR buffer is located in drivers/debug/debug_nrf_etr.c. It is using *ARM Coresight Trace Deformatter* and *MIPI STP Decoder* and above-mentioned demultiplexer to decode messages.

Logging messages contains read-only format string used in the logging macros thus they cannot be removed from the binary. This mode uses more read-only memory and has lower throughput because messages are longer and processing is time consuming. Compared to the dictionary-based mode logging is 2-3 slower.

Configuration For Nordic SoC a dedicated snippet (*Nordic Standalone STM logging snippet (nordic-log-stm)*) shall be used to enable logging. Each core shall be build with that snippet. If any core wants to use it then application core is required to have it enabled too since it acts as the proxy (ETR buffer processing). All cores must use the same configuration of the logging.

Reading the logs Logs are printed using Console UART.

> **Note**
>
> To use UART in your application, the UART's node must be described in devicetree. For more details, see *Introduction to devicetree*.

The following is an example log output:

```
[00:00:00.154,790] <inf> app/spsc_pbuf: alloc in 0x2f0df800
[00:00:00.163,319] <inf> app/spsc_pbuf: alloc 0x2f0df800 wr_idx:20
[00:00:00.181,112] <inf> app/spsc_pbuf: commit in 0x2f0df800
[00:00:00.189,090] <inf> app/spsc_pbuf: commit 0x2f0df800, len:20 wr_idx: 44
[00:00:00.202,577] <inf> rad/icmsg: mbox_callback
[00:00:00.214,750] <inf> rad/spsc_pbuf: claim 0x2f0df800 rd_idx:20
[00:00:00.235,823] <inf> rad/spsc_pbuf: free 0x2f0df800 len:20 rd_idx: 44
[00:00:00.244,507] <inf> rad/spsc_pbuf: read done 0x2f0df800 len:20
[00:00:00.272,444] <inf> rad/host: ep recv 0x330021f0, len:20
[00:00:00.283,939] <inf> rad/host: rx:00 exp:00
[00:00:00.292,200] <inf> rad/icmsg: read 0
[00:00:05.077,026] <inf> rad/spsc_pbuf: alloc in 0x2f0df000
[00:00:05.077,068] <inf> rad/spsc_pbuf: alloc 0x2f0df000 wr_idx:44
[00:00:05.077,098] <inf> rad/spsc_pbuf: commit in 0x2f0df000
[00:00:05.077,134] <inf> rad/spsc_pbuf: commit 0x2f0df000, len:20 wr_idx
```

Each log line contains a domain-related or core-related prefix between the log level and the module name, indicating the core that generated the log entry. The following are the prefixes used to indicate the cores:

Table 47: nRF54H20 log prefixes

Core	Prefix
Secure Domain	`sec`
Application core	`app`
Radio core	`rad`
System Controller (SysCtrl)	`sys`
Fast Lightweight Processor (FLPR)	`flpr`
Peripheral Processor (PPR)	`ppr`

Additional considerations When using STM logging, consider the following:

- Use optimized log macros (having up to 2 word size numeric arguments, like `LOG_INF("%d %c", (int)x, (char)y)`) to improve the size and speed of logging.
- For memory constrained applications (for example, when running on the PPR core), disable the `printk()` function by setting both the `CONFIG_PRINTK` and `CONFIG_BOOT_BANNER` Kconfig options to `n` in your project configuration.
- When working with multiple domains, such as the Radio core and Application core, ensure that each database is prefixed with the correct domain ID.
- Some log messages might be dropped due to the limited size of the RAM buffer that stores STM logs.

4.4.2 Tracing

Overview

The tracing feature provides hooks that permits you to collect data from your application and allows *Tracing Tools* running on a host to visualize the inner-working of the kernel and various subsystems.

Every system has application-specific events to trace out. Historically, that has implied:

1. Determining the application-specific payload,
2. Choosing suitable serialization-format,
3. Writing the on-target serialization code,
4. Deciding on and writing the I/O transport mechanics,
5. Writing the PC-side deserializer/parser,
6. Writing custom ad-hoc tools for filtering and presentation.

An application can use one of the existing formats or define a custom format by overriding the macros declared in include/zephyr/tracing/tracing.h.

Different formats, transports and host tools are available and supported in Zephyr.

In fact, I/O varies greatly from system to system. Therefore, it is instructive to create a taxonomy for I/O types when we must ensure the interface between payload/format (Top Layer) and the transport mechanics (bottom Layer) is generic and efficient enough to model these. See the *I/O taxonomy* section below.

Named Trace Events

Although the user can extend any of the supported serialization formats to enable additional tracing functions (or provide their own backend), zephyr also provides one generic named tracing function for convenience purposes, as well as to demonstrate how tracing frameworks could be extended.

Users can generate a custom trace event by calling `sys_trace_named_event()`, which takes an event name as well as two arbitrary 4 byte arguments. Tracing backends may truncate the provided event name if it is too long for the serialization format they support.

Serialization Formats

Common Trace Format (CTF) Support Common Trace Format, CTF, is an open format and language to describe trace formats. This enables tool reuse, of which line-textual (babeltrace) and graphical (TraceCompass) variants already exist.

CTF should look familiar to C programmers but adds stronger typing. See CTF - A Flexible, High-performance Binary Trace Format.

CTF allows us to formally describe application specific payload and the serialization format, which enables common infrastructure for host tools and parsers and tools for filtering and presentation.

A Generic Interface In CTF, an event is serialized to a packet containing one or more fields. As seen from *I/O taxonomy* section below, a bottom layer may:

- perform actions at transaction-start (e.g. mutex-lock),
- process each field in some way (e.g. sync-push emit, concat, enqueue to thread-bound FIFO),
- perform actions at transaction-stop (e.g. mutex-release, emit of concat buffer).

CTF Top-Layer Example The CTF_EVENT macro will serialize each argument to a field:

```
/* Example for illustration */
static inline void ctf_top_foo(uint32_t thread_id, ctf_bounded_string_t name)
{
  CTF_EVENT(
    CTF_LITERAL(uint8_t, 42),
    thread_id,
    name,
    "hello, I was emitted from function: ",
    __func__  /* __func__ is standard since C99 */
  );
}
```

How to serialize and emit fields as well as handling alignment, can be done internally and statically at compile-time in the bottom-layer.

The CTF top layer is enabled using the configuration option `CONFIG_TRACING_CTF` and can be used with the different transport backends both in synchronous and asynchronous modes.

Tracing Tools

Zephyr includes support for several popular tracing tools, presented below in alphabetical order.

Percepio Tracealyzer Support Zephyr includes support for Percepio Tracealyzer that offers trace visualization for simplified analysis, report generation and other analysis features. Tracealyzer allows for trace streaming over various interfaces and also snapshot tracing, where the events are kept in a RAM buffer.

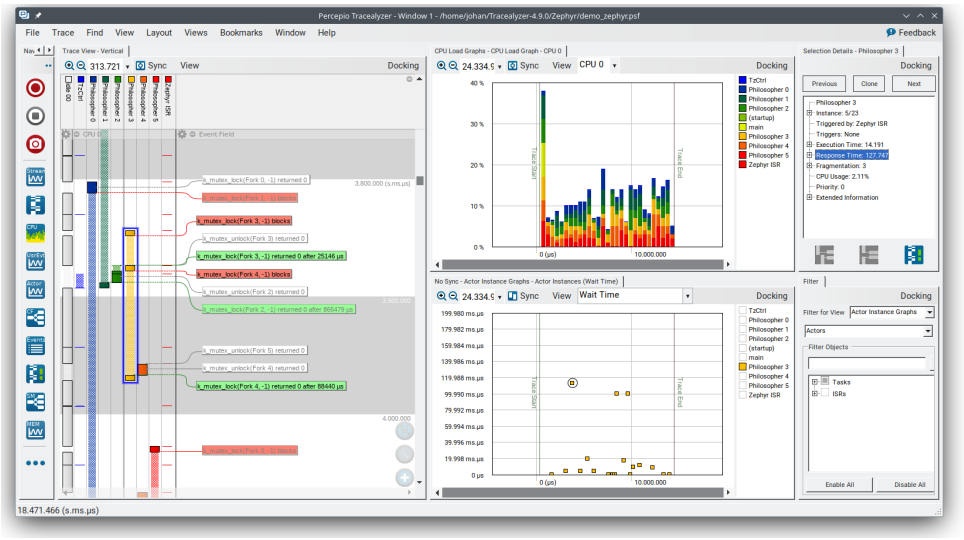
Zephyr kernel events are captured automatically when Tracealyzer tracing is enabled. Tracealyzer also provides extensive support for application logging, where you call the tracing library from your application code. This lets you visualize kernel events and application events together, for example as data plots or state diagrams on logged variables. Learn more in the Tracealyzer User Manual provided with the application.

Percepio TraceRecorder and Stream Ports The tracing library for Tracealyzer (TraceRecorder) is included in the Zephyr manifest and provided under the same license (Apache 2.0). This is enabled by adding the following configuration options in your prj.conf:

```
CONFIG_TRACING=y
CONFIG_PERCEPIO_TRACERECORDER=y
```

Or using menuconfig:

- Enable *Subsystems and OS Services* ▸ *Tracing Support*



- Under *Subsystems and OS Services* ▸ *Tracing Support* ▸ *Tracing Format*, select *Percepio Tracealyzer*

Some additional settings are needed to configure TraceRecorder. The most important configuration is to select the right "stream port". This specifies how to output the trace data. As of July 2024, the following stream ports are available in the Zephyr configuration system:

- **Ring Buffer**: The trace data is kept in a circular RAM buffer.
- **RTT**: Trace streaming via Segger RTT on J-Link debug probes.
- **ITM**: Trace streaming via the ITM function on Arm Cortex-M devices.
- **Semihost**: For tracing on QEMU. Streams the trace data to a host file.

Select the stream port in menuconfig under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Stream Port*.

Or simply add one of the following options in your prj.conf:

```
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RINGBUFFER=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RTT=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_ITM=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_ZEPHYR_SEMIHOST=y
```

Make sure to only include ONE of these configuration options.

The stream port modules have individual configuration options. In menuconfig these are found under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *(Stream Port) Config*. The most important options for each stream port are described below.

Tracealyzer Snapshot Tracing (Ring Buffer) The "Ring Buffer" stream port keeps the trace data in a RAM buffer on the device. By default, this is a circular buffer, meaning that it always contains the most recent data. This is used to dump "snapshots" of the trace data, e.g. by using the debugger. This usually only allows for short traces, unless you have megabytes of RAM to spare, so it is not suitable for profiling. However, it can be quite useful for debugging in combination with breakpoints. For example, if you set a breakpoint in an error handler, a snapshot trace can show the sequence of events leading up to the error. Snapshot tracing is also easy to begin with, since it doesn't depend on any particular debug probe or other development tool.

To use the Ring Buffer option, make sure to have the following configuration options in your prj.cnf:

```
CONFIG_TRACING=y
CONFIG_PERCEPIO_TRACERECORDER=y
CONFIG_PERCEPIO_TRC_START_MODE_START=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RINGBUFFER=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RINGBUFFER_SIZE=<size in bytes>
```

Or if using menuconfig:

- Enable *Subsystems and OS Services* ▸ *Tracing Support*
- Under *Subsystems and OS Services* ▸ *Tracing Support* ▸ *Tracing Format*, select *Percepio Tracealyzer*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Recorder Start Mode*, select *Start*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Stream Port*, select *Ring Buffer*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Ring Buffer Config* ▸ *Buffer Size*, set the buffer size in bytes.

The default buffer size can be reduced if you are tight on RAM, or increased if you have RAM to spare and want longer traces. You may also optimize the Tracing Configuration settings to get longer traces by filtering out less important events. In menuconfig, see *Subsystems and OS Services* ▸ *Tracing Support* ▸ *Tracing Configuration*.

To view the trace data, the easiest way is to start your debugger (west debug) and run the following GDB command:

```
dump binary value trace.bin *RecorderDataPtr
```

The resulting file is typically found in the root of the build folder, unless a different path is specified. Open this file in Tracealyzer by selecting *File* ▸ *Open* ▸ *Open File*.

Tracealyzer Streaming with SEGGER RTT Tracealyzer has built-in support for SEGGER RTT to receive trace data using a J-Link probe. This allows for recording very long traces. To configure Zephyr for RTT streaming to Tracealyzer, add the following configuration options in your prj.cnf:

```
CONFIG_TRACING=y
CONFIG_PERCEPIO_TRACERECORDER=y
CONFIG_PERCEPIO_TRC_START_MODE_START_FROM_HOST=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RTT=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_RTT_UP_BUFFER_SIZE=<size in bytes>
```

Or if using menuconfig:

- Enable *Subsystems and OS Services* ▸ *Tracing Support*
- Under *Subsystems and OS Services* ▸ *Tracing Support* ▸ *Tracing Format*, select *Percepio Tracealyzer*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Recorder Start Mode*, select *Start From Host*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Stream Port*, select *RTT*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *RTT Config*, set the size of the RTT "up" buffer in bytes.

The setting *RTT buffer size up* sets the size of the RTT transmission buffer. This is important for throughput. By default this buffer is quite large, 5000 bytes, to give decent performance also on onboard J-Link debuggers (they are not as fast as the stand-alone probes). If you are tight on RAM, you may consider reducing this setting. If using a regular J-Link probe it is often sufficient with a much smaller buffer, e.g. 1 KB or less.

Learn more about RTT streaming in the Tracealyzer User Manual. See Creating and Loading Traces -> Percepio TraceRecorder -> Using TraceRecorder v4.6 or later -> Stream ports (or search for RTT).

Tracealyzer Streaming with Arm ITM This stream port is for Arm Cortex-M devices featuring the ITM unit. It is recommended to use a fast debug probe that allows for SWO speeds of 10 MHz or higher. To use this stream port, apply the following configuration options:

```
CONFIG_TRACING=y
CONFIG_PERCEPIO_TRACERECORDER=y
CONFIG_PERCEPIO_TRC_START_MODE_START=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_ITM=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_ITM_PORT=1
```

Or if using menuconfig:

- Enable *Subsystems and OS Services* ▸ *Tracing Support*
- Under *Subsystems and OS Services* ▸ *Tracing Support* ▸ *Tracing Format*, select *Percepio Tracealyzer*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Recorder Start Mode*, select *Start*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Stream Port*, select *ITM*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *ITM Config*, set the ITM port to 1.

The main setting for the ITM stream port is the ITM port (0-31). A dedicated channel is needed for Tracealyzer. Port 0 is usually reserved for printf logging, so channel 1 is used by default.

The option *Use internal buffer* should typically remain disabled. It buffers the data in RAM before transmission and defers the data transmission to the periodic TzCtrl thread.

The host-side setup depends on what debug probe you are using. Learn more in the Tracealyzer User Manual. See *Creating and Loading Traces* ▸ *Percepio TraceRecorder* ▸ *Using TraceRecorder v4.6 or later* ▸ *Stream ports (or search for ITM)*.

Tracealyzer Streaming from QEMU (Semihost) This stream port is designed for Zephyr tracing in QEMU. This can be an easy way to get started with tracing and try out streaming trace without needing a fast debug probe. The data is streamed to a host file using semihosting. To use this option, apply the following configuration options:

```
CONFIG_SEMIHOST=y
CONFIG_TRACING=y
CONFIG_PERCEPIO_TRACERECORDER=y
CONFIG_PERCEPIO_TRC_START_MODE_START=y
CONFIG_PERCEPIO_TRC_CFG_STREAM_PORT_ZEPHYR_SEMIHOST=y
```

Using menuconfig

- Enable *General Architecture Options* ▸ *Semihosting support for Arm and RISC-V targets*
- Enable *Subsystems and OS Services* ▸ *Tracing Support*
- Under *Subsystems and OS Services* ▸ *Tracing Support* ▸ *Tracing Format*, select *Percepio Tracealyzer*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Recorder Start Mode*, select *Start*
- Under *Modules* ▸ *percepio* ▸ *TraceRecorder* ▸ *Stream Port*, select *Semihost*

By default, the resulting trace file is found in `./trace.psf` in the root of the build folder, unless a different path is specified. Open this file in Percepio Tracealyzer by selecting *File* ▸ *Open* ▸ *Open File*.

Recorder Start Mode You may have noticed the *Recorder Start Mode* option in the Tracealyzer examples above. This decides when the tracing starts. With the option *Start*, the tracing begins directly at startup, once the TraceRecorder library has been initialized. This is recommended when using the Ring Buffer and Semihost stream ports.

For streaming via RTT or ITM you may also use *Start From Host* or *Start Await Host*. Both listens for start commands from the Tracealyzer application. The latter option, *Start Await Host*, causes the TraceRecorder initialization to block until the start command is received from the Tracealyzer application.

Custom Stream Ports for Tracealyzer The stream ports are small modules within TraceRecorder that define what functions to call to output the trace data and (optionally) how to read start/stop commands from Tracealyzer. It is fairly easy to make custom stream ports to implement your own data transport and Tracealyzer can receive trace streams over various interfaces, including files, sockets, COM ports, named pipes and more. Note that additional stream port modules are available in the TraceRecorder repo (e.g. lwIP), although they might require modifications to work with Zephyr.

Learning More Learn more about how to get started in the Tracealyzer Getting Started Guides.

Percepio View for Zephyr Percepio View is a free-of-charge tracing tool based on Percepio Tracealyzer, intended for debugging and verification of Zephyr applications.

Percepio View can be used side-by-side with a traditional debugger and complements your debugger by visualising the real-time execution of threads, ISRs, syscalls and your own "User Events".

To learn more about Percepio View, how to get started and upgrade options, check out Percepio's product page.

Percepio View provides snapshot tracing, meaning the data is stored to a ring-buffer in target RAM and is saved to host using the regular debugger connection. For trace streaming support, Percepio offers (paid-for) upgrades to Percepio Profile or Percepio Tracealyzer. No modifications of the Zephyr source code are needed, only enabling the TraceRecorder library in Kconfig. Percepio View runs on Windows and Linux hosts.

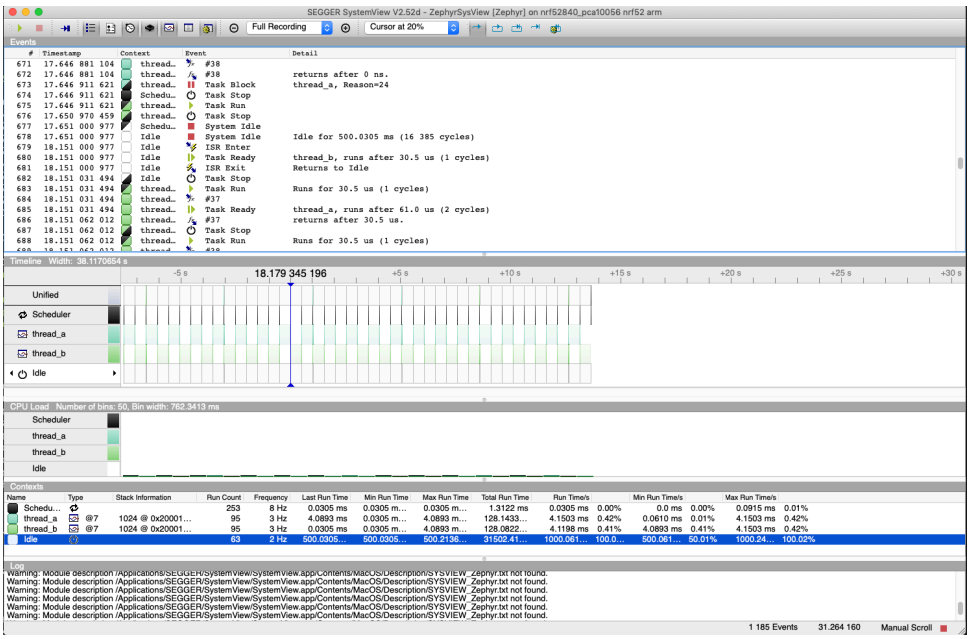
SEGGER SystemView Support Zephyr provides built-in support for SEGGER SystemView that can be enabled in any application for platforms that have the required hardware support.

The payload and format used with SystemView is custom to the application and relies on RTT as a transport. Newer versions of SystemView support other transports, for example UART or using snapshot mode (both still not supported in Zephyr).

To enable tracing support with SEGGER SystemView add the *SystemView RTT Tracing Snippet (rtt-tracing)* to your build command:

```
west build -b <board> -S rtt-tracing samples/synchronization
```

SystemView can also be used for post-mortem tracing, which can be enabled with `CONFIG_SEGGER_SYSVIEW_POST_MORTEM_MODE`. In this mode, a debugger can be attached after the system has crashed using `west attach` after which the latest data from the internal RAM buffer can be loaded into SystemView.



Recent versions of SEGGER SystemView come with an API translation table for Zephyr which is incomplete and does not match the current level of support available in Zephyr. To use the latest Zephyr API description table, copy the file available in the tree to your local configuration directory to override the builtin table:

```
# On Linux and MacOS
cp $ZEPHYR_BASE/subsys/tracing/sysview/SYSVIEW_Zephyr.txt ~/.config/SEGGER/
```

TraceCompass TraceCompass is an open source tool that visualizes CTF events such as thread scheduling and interrupts, and is helpful to find unintended interactions and resource conflicts on complex systems.

See also the presentation by Ericsson, Advanced Trouble-shooting Of Real-time Systems.

Perfetto Perfetto is a web trace viewer for the Chrome Trace Event Format. It shows the same scheduling and interrupt activity as TraceCompass without installing any tool: scripts/tracing/ctf2perfetto.py converts a CTF trace to that format with Python 3 alone:

```
$ZEPHYR_BASE/scripts/tracing/ctf2perfetto.py <trace file> -o trace.json
```

Loading `trace.json` at https://ui.perfetto.dev gives a timeline with one lane per thread, plus lanes for ISRs and for kernel objects such as semaphores, mutexes and timers.

The *Instrumentation* subsystem writes the same format from function call traces, so a build can be looked at either way in the same viewer.

The TSDL `metadata` is read from next to the trace, as babeltrace2 expects, unless `--metadata` points elsewhere. Use the metadata generated by the build that produced the trace: CTF records carry no length, so a record the metadata does not describe cannot be skipped, and the converter stops and names its event id rather than writing a partial trace.

User-Defined Tracing This tracing format lets the application define plain C functions that run on each hook, with no need to author or fork a tracing header. Each `sys_port_trace_*` hook is routed to a `__weak` callback named `sys_trace_<event>_user()`; the application overrides only the callbacks it cares about, and any callback it leaves undefined is a no-op.

Examples include:

- simple toggling of GPIO for external scope tracing while minimizing extra cpu load
- generating/outputting trace data in a non-standard or proprietary format that can not be supported by the other tracing systems

The callbacks cover thread/scheduler, ISR, idle, system init and sleep events, GPIO, timers, RTIO, and the kernel objects (semaphores, mutexes, condition variables, message queues, mailboxes, events, polling, memory slabs and work queues). The authoritative list, with exact signatures, is the set of `sys_trace_*_user` declarations in subsys/tracing/user/tracing_user.h. A representative subset:

```
void sys_trace_thread_create_user(struct k_thread *thread);
void sys_trace_thread_switched_in_user(void);
void sys_trace_thread_switched_out_user(void);
void sys_trace_isr_enter_user(void);
void sys_trace_isr_exit_user(void);
void sys_trace_idle_user(void);
void sys_trace_k_sem_give_enter_user(struct k_sem *sem);
void sys_trace_k_mutex_lock_enter_user(struct k_mutex *mutex, k_timeout_t timeout);
```

Enable this format with the `CONFIG_TRACING_USER` option. tests/subsys/tracing/tracing_user_callbacks shows an application overriding object callbacks.

Out-of-tree Tracing Formats A downstream module can supply its own tracing format without editing the Zephyr tree. Set `CONFIG_TRACING_FORMAT_HEADER` (a promptless Kconfig string, assigned from the module's `Kconfig` with a `default`) to the name of a header on the include path;

include/zephyr/tracing/tracing.h pulls it in with a computed `#include` when no in-tree format is selected.

The format header defines the `sys_port_trace_*` hooks it records and ends with:

```
/* default no-ops for the scheduler/idle/init/named function layer */
#include <zephyr/tracing/tracing_default_hooks.h>
/* no-op for every sys_port_trace_* hook this header does not define */
#include <zephyr/tracing/tracing_hooks.h>
```

so a format only has to implement the hooks it cares about. A tracing transport is added independently with `TRACING_BACKEND_DEFINE()` (see subsys/tracing/include/tracing_backend.h), so neither a custom format nor a custom backend requires changes to the core tree. tests/subsys/tracing/tracing_format_module is a complete worked example.

Transport Backends

The following backends are currently supported:

- UART
- USB
- File (Using the native port with POSIX architecture based targets)
- RTT (With SystemView)
- RAM (buffer to be retrieved by a debugger)

Using Tracing

The sample samples/subsys/tracing demonstrates tracing with different formats and backends.

To get started, the simplest way is to use the CTF format with the native_sim port, build the sample as follows:

Using west:

```
# From the root of the zephyr repository
west build -b native_sim samples/subsys/tracing -- -DCONF_FILE=prj_native_ctf.conf
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=native_sim -DCONF_FILE=prj_native_ctf.conf␣
↪samples/subsys/tracing

# Now run the build tool on the generated build system:
ninja -Cbuild
```

You can then run the resulting binary with the option `-trace-file` to generate the tracing data:

```
mkdir data
cp $ZEPHYR_BASE/subsys/tracing/ctf/tsdl/metadata data/
./build/zephyr/zephyr.exe -trace-file=data/channel0_0
```

The resulting CTF output can be visualized using babeltrace or TraceCompass by pointing the tool to the `data` directory with the metadata and trace files.

Using RAM backend For devices that do not have available I/O for tracing such as USB or UART but have enough RAM to collect trace data, the ram backend can be enabled with configuration `CONFIG_TRACING_BACKEND_RAM`. Adjust `CONFIG_RAM_TRACING_BUFFER_SIZE` to be able to record enough traces for your needs. Then thanks to a runtime debugger such as gdb this buffer can be fetched from the target to an host computer:

```
(gdb) dump binary memory data/channel0_0 <ram_tracing_start> <ram_tracing_end>
```

The resulting channel0_0 file have to be placed in a directory with the `metadata` file like the other backend.

Future LTTng Inspiration

Currently, the top-layer provided here is quite simple and bare-bones, and needlessly copied from Zephyr's Segger SystemView debug module.

For an OS like Zephyr, it would make sense to draw inspiration from Linux's LTTng and change the top-layer to serialize to the same format. Doing this would enable direct reuse of TraceCompass' canned analyses for Linux. Alternatively, LTTng-analyses in TraceCompass could be customized to Zephyr. It is ongoing work to enable TraceCompass visibility of Zephyr in a target-agnostic and open source way.

I/O Taxonomy

- Atomic Push/Produce/Write/Enqueue:
  - **synchronous:**
    means data-transmission has completed with the return of the call.
  - **asynchronous:**
    means data-transmission is pending or ongoing with the return of the call. Usually, interrupts/callbacks/signals or polling is used to determine completion.
  - **buffered:**
    means data-transmissions are copied and grouped together to form a larger ones. Usually for amortizing overhead (burst dequeue) or jitter-mitigation (steady dequeue).

  Examples:

  - **sync unbuffered**
    E.g. PIO via GPIOs having steady stream, no extra FIFO memory needed. Low jitter but may be less efficient (can't amortize the overhead of writing).
  - **sync buffered**
    E.g. `fwrite()` or enqueuing into FIFO. Blockingly burst the FIFO when its buffer-waterlevel exceeds threshold. Jitter due to bursts may lead to missed deadlines.
  - **async unbuffered**
    E.g. DMA, or zero-copying in shared memory. Be careful of data hazards, race conditions, etc!
  - **async buffered**
    E.g. enqueuing into FIFO.
- Atomic Pull/Consume/Read/Dequeue:
  - **synchronous:**
    means data-reception has completed with the return of the call.
  - **asynchronous:**
    means data-reception is pending or ongoing with the return of the call. Usually, interrupts/callbacks/signals or polling is used to determine completion.

- **buffered:**
  means data is copied-in in larger chunks than request-size. Usually for amortizing wait-time.

Examples:

- **sync unbuffered**
  E.g. Blocking read-call, `fread()` or SPI-read, zero-copying in shared memory.
- **sync buffered**
  E.g. Blocking read-call with caching applied. Makes sense if read pattern exhibits spatial locality.
- **async unbuffered**
  E.g. zero-copying in shared memory. Be careful of data hazards, race conditions, etc!
- **async buffered**
  E.g. `aio_read()` or DMA.

Unfortunately, I/O may not be atomic and may, therefore, require locking. Locking may not be needed if multiple independent channels are available.

- **The system has non-atomic write and one shared channel**
  E.g. UART. Locking required.

  `lock(); emit(a); emit(b); emit(c); release();`
- **The system has non-atomic write but many channels**
  E.g. Multi-UART. Lock-free if the bottom-layer maps each Zephyr thread+ISR to its own channel, thus alleviating races as each thread is sequentially consistent with itself.

  `emit(a,thread_id); emit(b,thread_id); emit(c,thread_id);`
- **The system has atomic write but one shared channel**
  E.g. `native_sim` or board with DMA. May or may not need locking.

  `emit(a ## b ## c); /* Concat to buffer */`

  `lock(); emit(a); emit(b); emit(c); release(); /* No extra mem */`
- **The system has atomic write and many channels**
  E.g. native_sim or board with multi-channel DMA. Lock-free.

  `emit(a ## b ## c, thread_id);`

Object tracking

The kernel can also keep track of the kernel objects in the system. Enabling `CONFIG_TRACING_OBJECT_TRACKING` enables the *object core framework*, which enumerates the objects of each kernel object type. For instance, to visit every mutex, one can write:

```
static int visit_mutex(struct k_obj_core *obj_core, void *data)
{
    struct k_mutex *mutex = CONTAINER_OF(obj_core, struct k_mutex, obj_core);

    /* Do something */

    return 0;
}

k_obj_type_walk_locked(k_obj_type_find(K_OBJ_TYPE_MUTEX_ID), visit_mutex, NULL);
```

Each object type can be excluded through its `CONFIG_OBJ_CORE_*` option. The object_cores sample can be built with object tracking enabled.

API

Common
Tracing hooks

Threads
Thread

Work Queues
Work item

Poll
Polling

Semaphore
Semaphore

Mutex
Mutex

Condition Variables
Conditional variable

Queues
Queue

FIFO
FIFO

LIFO
LIFO

Stacks
Stack

Message Queues
Message queue

Mailbox
Mailbox

Pipes
Pipe

Heaps
Heap

Memory Slabs
Memory slab

Timers
Timer

Syscalls
Syscall Tracing

Network tracing
Network

Network socket tracing
Network socket

4.4.3 Debugging

Thread analyzer

The thread analyzer module enables all the Zephyr options required to track the thread information, e.g. thread stack size usage and other runtime thread runtime statistics.

The analysis is performed on demand when the application calls `thread_analyzer_run()` or `thread_analyzer_print()`.

For example, to build the synchronization sample with Thread Analyser enabled, do the following:

```
# From the root of the zephyr repository
west build -b qemu_x86 samples/synchronization/ -- -
↪DCONFIG_QEMU_ICOUNT=n -DCONFIG_THREAD_ANALYZER=y \
-DCONFIG_THREAD_ANALYZER_USE_PRINTK=y -DCONFIG_THREAD_ANALYZER_AUTO=y \
-DCONFIG_THREAD_ANALYZER_AUTO_INTERVAL=5
```

When you run the generated application in Qemu, you will get the additional information from Thread Analyzer:

```
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 740 usage 284 / 1024 (27 %); CPU: 0 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 98 %
 idle                : STACK: unused 204 usage 116 / 320 (36 %); CPU: 0 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 7 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 9 %
 idle                : STACK: unused 204 usage 116 / 320 (36 %); CPU: 82 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
```

(continues on next page)

(continued from previous page)

```
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
Thread analyze:
 thread_b            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 7 %
 thread_analyzer     : STACK: unused 8 usage 504 / 512 (98 %); CPU: 0 %
 thread_a            : STACK: unused 648 usage 376 / 1024 (36 %); CPU: 8 %
 idle                : STACK: unused 204 usage 116 / 320 (36 %); CPU: 83 %
thread_b: Hello World from cpu 0 on qemu_x86!
thread_a: Hello World from cpu 0 on qemu_x86!
thread_b: Hello World from cpu 0 on qemu_x86!
```

Configuration Configure this module using the following options.

`CONFIG_THREAD_ANALYZER`
Enable the module.

`CONFIG_THREAD_ANALYZER_USE_PRINTK`
Use printk for thread statistics.

`CONFIG_THREAD_ANALYZER_USE_LOG`
Use the logger for thread statistics.

`CONFIG_THREAD_ANALYZER_AUTO`
Run the thread analyzer automatically. You do not need to add any code to the application when using this option.

`CONFIG_THREAD_ANALYZER_AUTO_INTERVAL`
The time for which the module sleeps between consecutive printing of thread analysis in automatic mode.

`CONFIG_THREAD_ANALYZER_AUTO_STACK_SIZE`
The stack for thread analyzer automatic thread.

`CONFIG_THREAD_NAME`
Print the name of the thread instead of its ID.

`CONFIG_THREAD_RUNTIME_STATS`
Print thread runtime data such as utilization. This options is automatically selected by `CONFIG_THREAD_ANALYZER`.

`CONFIG_THREAD_ANALYZER_LONG_FRAME_PER_INTERVAL`
Reset Longest Frame value statistics after printing. When using `SCHED_THREAD_USAGE_ANALYSIS` to get average and longest frame thread statistics, reset the Longest Frame value to zero after each time printing the thread statistics. This enables observation of the longest frame during the most recent interval rather than longest frame since startup.

`CONFIG_THREAD_ANALYZER_PRINT_THREAD_PRIORITY`
Print each thread's priority.

API documentation

Thread analyzer

Core Dump

The core dump module enables dumping the CPU registers and memory content for offline debugging. This module is called when a fatal error is encountered and prints or stores data according to which backends are enabled.

Configuration Configure this module using the following options.

- `DEBUG_COREDUMP`: enable the module.

Here are the options to enable output backends for core dump:

- `DEBUG_COREDUMP_BACKEND_LOGGING`: use log module for core dump output.
- `DEBUG_COREDUMP_BACKEND_LOGGING_UDP`: same as logging backend with optional raw UDP transfer; the peer is `DEBUG_COREDUMP_LOGGING_UDP_HOST`, a string parsed by `net_ipaddr_parse()` (IPv4 or IPv6, optional `:port` with default UDP port 17777 when omitted). Use scripts/coredump/coredump_udp_receiver.py to build the binary for scripts/coredump/coredump_gdbserver.py.
- `DEBUG_COREDUMP_BACKEND_FLASH_PARTITION`: use flash partition for core dump output.
- `DEBUG_COREDUMP_BACKEND_NULL`: fallback core dump backend if other backends cannot be enabled. All output is sent to null.

Here are the choices regarding memory dump:

- `DEBUG_COREDUMP_MEMORY_DUMP_MIN`: only dumps the stack of the exception thread, its thread struct, and some other bare minimal data to support walking the stack in the debugger. Use this only if absolute minimum of data dump is desired.
- `DEBUG_COREDUMP_MEMORY_DUMP_THREADS`: Dumps the thread struct and stack of all threads and all data required to debug threads.
- `DEBUG_COREDUMP_MEMORY_DUMP_LINKER_RAM`: Dumps the memory region between _image_ram_start[] and _image_ram_end[]. This includes at least data, noinit, and BSS sections. This is the default.

Additional memory can be included in a dump (even with the "DEBUG_COREDUMP_MEMORY_DUMP_MIN" config selected) through one or more *coredump devices*

Usage When the core dump module is enabled, during a fatal error, CPU registers and memory content are printed or stored according to which backends are enabled. This core dump data can be fed into a custom-made GDB server as a remote target for GDB (and other GDB compatible debuggers). CPU registers, memory content and stack can be examined in the debugger.

This usually involves the following steps:

1. Get the core dump log from the device depending on enabled backends. For example, if the log module backend is used, get the log output from the log module backend.
2. Convert the core dump log into a binary format that can be parsed by the GDB server. For example, scripts/coredump/coredump_serial_log_parser.py can be used to convert the serial console log into a binary file. If the UDP coredump backend is enabled (`DEBUG_COREDUMP_BACKEND_LOGGING_UDP`), run scripts/coredump/coredump_udp_receiver.py on the collector host to reassemble UDP datagrams into the **same** raw binary format.
3. Start the custom GDB server using the script scripts/coredump/coredump_gdbserver.py with the core dump binary log file, and the Zephyr ELF file as parameters. The GDB server can also be started from within GDB, see below.
4. Start the debugger corresponding to the target architecture.

> **Note**
>
> Developers for Intel ADSP CAVS 15-25 platforms using `ZEPHYR_TOOLCHAIN_VARIANT=zephyr` should use the debugger in the `xtensa-intel_apl_adsp` toolchain of the SDK.

5. When `DEBUG_COREDUMP_BACKEND_FLASH_PARTITION` is enabled the core dump data is stored in the flash partition. The flash partition must be defined in the device tree:

```
&flash0 {
   partitions {
      coredump_partition: partition@255000 {
         label = "coredump-partition";
         reg = <0x255000 DT_SIZE_K(4)>;
      };
   };
};
```

Example This example uses the log module backend tied to serial console. This was done on qemu_x86 where a null pointer was dereferenced.

This is the core dump log from the serial console, and is stored in `coredump.log`:

```
Booting from ROM..*** Booting Zephyr OS build zephyr-v2.3.0-1840-g7bba91944a63  ***
Hello World! qemu_x86
E: Page fault at address 0x0 (error code 0x2)
E: Linear address not present in page tables
E:   PDE: 0x0000000000115827 Writable, User, Execute Enabled
E:   PTE: Non-present
E: EAX: 0x00000000, EBX: 0x00000000, ECX: 0x00119d74, EDX: 0x000003f8
E: ESI: 0x00000000, EDI: 0x00101aa7, EBP: 0x00119d10, ESP: 0x00119d00
E: EFLAGS: 0x00000206 CS: 0x0008 CR3: 0x00119000
E: call trace:
E: EIP: 0x00100459
E:      0x00100477 (0x0)
E:      0x00100492 (0x0)
E:      0x001004c8 (0x0)
E:      0x00105465 (0x105465)
E:      0x00101abe (0x0)
E: >>> ZEPHYR FATAL ERROR 0: CPU exception on CPU 0
E: Current thread: 0x00119080 (unknown)
E: #CD:BEGIN#
E: #CD:5a4501000100050000000000
E: #CD:4101003800
E: #CD:0e000000020000000000000749d1100f8030000000000000009d1100109d1100
E: #CD:00000000a71a10005904100006020000080000000000901100
E: #CD:4d010080901100e0901100
E: #CD:0100000000000000000000000180000000000000000000000000000000000000
E: #CD:000000000000000000000000000000e36410000000000000000004c9c1100
E: #CD:0000000000000000000000000b4991100004000000000000fc03000000000000
E: #CD:4d0100b4991100b49d1100
E: #CD:f8030000020000000200000002000000f8030000fd03000a02000000dc9e1100
E: #CD:149a1160fd03000002000000dc9e1100249a110087201000049f11000a000000
E: #CD:349a11000a4f1000049f11000a9e1100449a11000a8b1000020000000002000000
E: #CD:449a1100388b1000049f11000a000000549a1100ad201000049f11000a000000
E: #CD:749a11000a201000049f11000a000000649a11000a201000049f11000a000000
E: #CD:749a1100e8201000049f11000a000000949a1100890b10000a0000000a000000
E: #CD:a49a1100890b10000a0000000a000000f8030000189b1100020000000002000000
E: #CD:f49a1100289b11000a000000189b1100049b11009b0710000a000000289b1100
E: #CD:f49a110087201000049f110045000000f49a110050901100a00000020901100
E: #CD:f49a110060901100049f1100ffffffff0000000000000000049f1100ffffffff
E: #CD:0000000000000000630b1000189b1100349b1100af0b1000630b1000289b1100
E: #CD:55891000789b110000000000020901100549b1100480000004a891000609b1100
```

(continues on next page)

(continued from previous page)

```
E: #CD:649b1100d00b10004a891000709b11000000000609b11000a00000000000000
E: #CD:849b1100709b11004a89100000000000949b1100794a10000000000058901100
E: #CD:20901100c34a10000a00001734020000d00100000000000d49b110038000000
E: #CD:c49b110078481000b499110000040000000000000000000000c9c11000c9c1100
E: #CD:149c110000000000d49b110038000000f49b1100da481000b499110000040000
E: #CD:0e000000020000000000000744d0100b4991100b49d1100009d1100109d1100
E: #CD:149c110099471000b4991100000400000800000000901100ad861000409c1100
E: #CD:349c1100e947100080901100000000000349c1100b6471000808610004500000
E: #CD:849c11002d53100000000000d09c110080901100208610000f5ffffff8c9c1100
E: #CD:00000000000000000000000a71a1000a49c110002020000809011000000000
E: #CD:a49c110002020000080000000000000a49c110019371000000000000d09c1100
E: #CD:0c9d0000bc9c0000b49d1100b4991100c49c1100ae37100000000000d09c1100
E: #CD:080000000000000c88810000000000109d11005d031000d09c1100009d1100
E: #CD:109d110000000000000000a71a1000f8030000000000000749d110002000000
E: #CD:5904100008000000060200000e000000020200000202000000000000002c9d1100
E: #CD:770410000000000d00b1000c9881000549d110000000000489d110092041000
E: #CD:00000000689d1100549d110000000000000000000689d1100c804100000000000
E: #CD:c088100000000000007c9d110000000000749d11007c9d110065541000065541000
E: #CD:0000000000000000009c9d1100be1a100000000000000000000000000000038041000
E: #CD:0800000002020000000000000000000000f4531000000000000000000000000000000
E: #CD:END#
E: Halting system
```

1. Run the core dump serial log converter:

   ```
   ./scripts/coredump/coredump_serial_log_parser.py coredump.log coredump.bin
   ```

2. Start the custom GDB server:

   ```
   ./scripts/coredump/coredump_gdbserver.py build/zephyr/zephyr.elf coredump.bin
   ```

3. Start GDB:

   ```
   <path to SDK>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb␣
   ↪build/zephyr/zephyr.elf
   ```

4. Inside GDB, connect to the GDB server via port 1234:

   ```
   (gdb) target remote localhost:1234
   ```

5. Examine the CPU registers:

   ```
   (gdb) info registers
   ```

   Output from GDB:

   ```
   eax            0x0                 0
   ecx            0x119d74            1154420
   edx            0x3f8               1016
   ebx            0x0                 0
   esp            0x119d00            0x119d00 <z_main_stack+844>
   ebp            0x119d10            0x119d10 <z_main_stack+860>
   esi            0x0                 0
   edi            0x101aa7            1055399
   eip            0x100459            0x100459 <func_3+16>
   eflags         0x206               [ PF IF ]
   cs             0x8                 8
   ss             <unavailable>
   ```

(continues on next page)

(continued from previous page)

```
ds             <unavailable>
es             <unavailable>
fs             <unavailable>
gs             <unavailable>
```

6. Examine the backtrace:

```
(gdb) bt
```

Output from GDB:

```
#0  0x00100459 in func_3 (addr=0x0) at␣
↪zephyr/rtos/zephyr/samples/hello_world/src/main.c:14
#1  0x00100477 in func_2 (addr=0x0) at␣
↪zephyr/rtos/zephyr/samples/hello_world/src/main.c:21
#2  0x00100492 in func_1 (addr=0x0) at␣
↪zephyr/rtos/zephyr/samples/hello_world/src/main.c:28
#3  0x001004c8 in main () at␣
↪zephyr/rtos/zephyr/samples/hello_world/src/main.c:42
```

Starting the GDB server from within GDB You can use `target remote |` to start the custom GDB server from inside GDB, instead of in a separate shell.

1. Start GDB:

```
<path to SDK>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb␣
↪build/zephyr/zephyr.elf
```

2. Inside GDB, start the GDB server using the `--pipe` option:

```
(gdb) target remote | ./scripts/coredump/coredump_gdbserver.py --pipe␣
↪build/zephyr/zephyr.elf coredump.bin
```

File Format The core dump binary file consists of one file header, one architecture-specific block, zero or one threads metadata block(s), and multiple memory blocks. All numbers in the headers below are little endian.

File Header The file header consists of the following fields:

Table 48: Core dump binary file header

Field	Data Type	Description
ID	`char[2]`	`Z`, `E` as identifier of file.
Header version	`uint16_`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Target code	`uint16_`	Indicate which target (e.g. architecture or SoC) so the parser can instantiate the correct register block parser.
Pointer size	'uint8_t'	Size of `uintptr_t` in power of 2. (e.g. 5 for 32-bit, 6 for 64-bit). This is needed to accommodate 32-bit and 64-bit target in parsing the memory block addresses.
Flags	`uint8_t`	
Fatal error reason	`unsigned int`	Reason for the fatal error, as the same in `enum k_fatal_error_reason` defined in include/zephyr/fatal.h

Architecture-specific Block The architecture-specific block contains the byte stream of data specific to the target architecture (e.g. CPU registers)

Table 49: Architecture-specific Block

Field	Data Type	Description
ID	`char`	`A` to indicate this is a architecture-specific block.
Header version	`uint16_`	Identify the version of this block. To be interpreted by the target architecture specific block parser.
Number of bytes	`uint16_`	Number of bytes following the header which contains the byte stream for target data. The format of the byte stream is specific to the target and is only being parsed by the target parser.
Register byte stream	`uint8_t`	Contains target architecture specific data.

Threads Metadata Block The threads metadata block contains the byte stream of data necessary for debugging threads.

Table 50: Threads Metadata Block

Field	Data Type	Description
ID	`char`	`T` to indicate this is a threads metadata block.
Header version	`uint16_`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Number of bytes	`uint16_`	Number of bytes following the header which contains the byte stream for target data.
Byte stream	`uint8_t`	Contains data necessary for debugging threads.

Memory Block The memory block contains the start and end addresses and the data within the memory region.

Table 51: Memory Block

Field	Data Type	Description
ID	`char`	`M` to indicate this is a memory block.
Header version	`uint16_`	Identify the version of the header. This needs to be incremented whenever the header struct is modified. This allows parser to reject older header versions so it will not incorrectly parse the header.
Start address	`uintptr`	The start address of the memory region.
End address	`uintptr`	The end address of the memory region.
Memory byte stream	`uint8_t`	Contains the memory content between the start and end addresses.

Adding New Target The architecture-specific block is target specific and requires new dumping routine and parser for new targets. To add a new target, the following needs to be done:

1. Add a new target code to the `enum coredump_tgt_code` in include/zephyr/debug/coredump.h.
2. Implement `arch_coredump_tgt_code_get()` simply to return the newly introduced target code.
3. Implement `arch_coredump_info_dump()` to construct a target architecture block and call `coredump_buffer_output()` to output the block to core dump backend.
4. Add a parser to the core dump GDB stub scripts under `scripts/coredump/gdbstubs/`
   1. Extends the `gdbstubs.gdbstub.GdbStub` class.
   2. During `__init__`, store the GDB signal corresponding to the exception reason in `self.gdb_signal`.
   3. Parse the architecture-specific block from `self.logfile.get_arch_data()`. This needs to match the format as implemented in step 3 (inside `arch_coredump_info_dump()`).
   4. Implement the abstract method `handle_register_group_read_packet` where it returns the register group as GDB expected. Refer to GDB's code and documentation on what it is expecting for the new target.
   5. Optionally implement `handle_register_single_read_packet` for registers not covered in the `g` packet.
5. Extend `get_gdbstub()` in scripts/coredump/gdbstubs/__init__.py to return the newly implemented GDB stub.

UDP logging backend sample Sample README pages that exercise the UDP coredump path are linked into this chapter so Sphinx includes them in the documentation tree:

UDP coredump via shell

Overview Zephyr shell stays running on the serial console. Run **"coredump_crash"** to fault the board; with **"CONFIG_DEBUG_COREDUMP_BACKEND_LOGGING_UDP"** the coredump byte stream is mirrored as `#CD:` hex on the UART and sent as framed UDP to **"CONFIG_DEBUG_COREDUMP_LOGGING_UDP_HOST"** (syntax is whatever `net_ipaddr_parse()` accepts, e.g. `192.0.2.2`, `192.0.2.2:17777`, `[2001:db8::1]:17777`; port defaults to **17777** when omitted).

Board overlay **"boards/versalnet_apu.conf"** sets lab-style static IPv4; adjust for your bench. Networking and UDP coredump options live in **"prj.conf"** (defaults use `192.0.2.x` before the overlay applies).

See *Core Dump* and subsys/debug/coredump/coredump_backend_logging_udp.c.

Build

```
west build -b versalnet_apu samples/subsys/debug/coredump_udp_demos/demo_shell
```

API documentation

Coredump APIs

Architecture-specific core dump APIs

GDB stub

- *Overview*
- *Features*
- *Enabling GDB Stub*
  - *Using Serial Backend*
- *Debugging*
  - *Using Serial Backend*
- *Example*

Overview The gdbstub feature provides an implementation of the GDB Remote Serial Protocol (RSP) that allows you to remotely debug Zephyr using GDB.

The protocol supports different connection types: serial, UDP/IP and TCP/IP. Zephyr currently supports only serial device communication.

The GDB program acts as a client while the Zephyr gdbstub acts as a server. When this feature is enabled, Zephyr stops its execution after `gdb_init()` starts gdbstub service and waits for a GDB connection. Once a connection is established it is possible to synchronously interact with Zephyr. Note that currently it is not possible to asynchronously send commands to the target.

Features The following features are supported:

- Add and remove breakpoints
- Continue and step the target
- Print backtrace
- Read or write general registers
- Read or write the memory

Enabling GDB Stub GDB stub can be enabled with the `CONFIG_GDBSTUB` option.

Using Serial Backend The serial backend for GDB stub can be enabled with the `CONFIG_GDBSTUB_SERIAL_BACKEND` option.

Since serial backend utilizes UART devices to send and receive GDB commands,

- If there are spare UART devices on the board, set `zephyr,gdbstub-uart` property of the chosen node to the spare UART device so that `printk()` and log messages are not being printed to the same UART device used for GDB.

- For boards with only one UART device, `printk()` and logging must be disabled if they are also using the same UART device for output. GDB related messages may interleave with log messages which may have unintended consequences. Usually this can be done by disabling `CONFIG_PRINTK` and `CONFIG_LOG`.

Debugging

Using Serial Backend

1. Build with GDB stub and serial backend enabled.
2. Flash built image onto board and reset the board.
   - Execution should now be paused at `gdb_init()`.
3. Execute GDB on development machine and connect to the GDB stub.

   ```
   target remote <serial device>
   ```

   For example,

   ```
   target remote /dev/ttyUSB1
   ```

4. GDB commands can be used to start debugging.

Example There is a test application tests/subsys/debug/gdbstub with one of its test cases `debug.gdbstub.breakpoints` demonstrating how the Zephyr GDB stub can be used. The test also has a case to connect to the QEMU's GDB stub implementation (at a custom port `tcp:1235`) as a reference to validate the test script itself.

Run the test with the following command from your *ZEPHYR_BASE* directory:

```
west twister -p qemu_x86 -T tests/subsys/debug/gdbstub
```

The test should run successfully, and now let's do something similar step-by-step to demonstrate how the Zephyr GDB stub works from the GDB user's perspective.

In the snippets below use and expect your appropriate directories instead of `<SDK install directory>`, `<build_directory>`, `<ZEPHYR_BASE>`.

1. Open two terminal windows.
2. On the first terminal, build and run the test application:

   ```
   # From the root of the zephyr repository
   west build -b qemu_x86 tests/subsys/debug/gdbstub -- '-
   ↪DCONFIG_QEMU_EXTRA_FLAGS="-serial tcp:localhost:5678,server"'
   west build -t run
   ```

   Note how we set `CONFIG_QEMU_EXTRA_FLAGS` to direct QEMU serial console port to the `localhost` TCP port `5678` to wait for a connection from the GDB `remote` command we are going to do on the next steps.

3. On the second terminal, start GDB:

   ```
   <SDK install directory>/x86_64-zephyr-elf/bin/x86_64-zephyr-elf-gdb
   ```

   1. Tell GDB where to look for the built ELF file:

      ```
      (gdb) symbol-file <build directory>/zephyr/zephyr.elf
      ```

      Response from GDB:

```
Reading symbols from <build directory>/zephyr/zephyr.elf...
```

2. Tell GDB to connect to the Zephyr gdbstub serial backend which is exposed earlier as a server through the TCP port `-serial` redirection at QEMU.

```
(gdb) target remote localhost:5678
```

Response from GDB:

```
Remote debugging using localhost:5678
arch_gdb_init () at <ZEPHYR_BASE>/arch/x86/core/ia32/gdbstub.c:252
252     }
```

GDB also shows where the code execution is stopped. In this case, it is at arch/x86/core/ia32/gdbstub.c, line 252.

3. Use command `bt` or `backtrace` to show the backtrace of stack frames.

```
(gdb) bt
#0  arch_gdb_init () at <ZEPHYR_BASE>/arch/x86/core/ia32/gdbstub.c:252
#1  0x00104140 in gdb_init () at
↪<ZEPHYR_BASE>/zephyr/subsys/debug/gdbstub.c:852
#2  0x00109c13 in z_sys_init_run_level (level=INIT_LEVEL_PRE_KERNEL_2) at
↪<ZEPHYR_BASE>/kernel/init.c:360
#3  0x00109e73 in z_cstart () at <ZEPHYR_BASE>/kernel/init.c:630
#4  0x00104422 in z_prep_c (arg=0x1245bc <x86_cpu_boot_arg>) at
↪<ZEPHYR_BASE>/arch/x86/core/prep_c.c:80
#5  0x001000c9 in __csSet () at
↪<ZEPHYR_BASE>/arch/x86/core/ia32/crt0.S:290
#6  0x001245bc in uart_dev ()
#7  0x00134988 in z_interrupt_stacks ()
#8  0x00000000 in ?? ()
```

4. Use command `list` to show the source code and surroundings where code execution is stopped.

```
(gdb) list
247             __asm__ volatile ("int3");
248
249     #ifdef CONFIG_GDBSTUB_TRACE
250             printk("gdbstub:%s GDB is connected\n", __func__);
251     #endif
252     }
253
254     /* Hook current IDT. */
255     _EXCEPTION_CONNECT_NOCODE(z_gdb_debug_isr, IV_DEBUG, 3);
256     _EXCEPTION_CONNECT_NOCODE(z_gdb_break_isr, IV_BREAKPOINT, 3);
```

5. Use command `s` or `step` to step through program until it reaches a different source line. Now that it finished executing `arch_gdb_init()` and is continuing in `gdb_init()`.

```
(gdb) s
gdb_init () at <ZEPHYR_BASE>/subsys/debug/gdbstub.c:857
857     return 0;
```

```
(gdb) list
852             arch_gdb_init();
853
```

(continues on next page)

(continued from previous page)

```
854     #ifdef CONFIG_GDBSTUB_TRACE
855             printk("gdbstub:%s exit\n", __func__);
856     #endif
857             return 0;
858     }
859
860     #ifdef CONFIG_XTENSA
861     /*
```

6. Use command `br` or `break` to setup a breakpoint. For this example set up a breakpoint at `main()`, and let code execution continue without any intervention using command `c` (or `continue`).

```
(gdb) break main
Breakpoint 1 at 0x10064d: file
↪<ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.c, line 27.
```

```
(gdb) continue
Continuing.
```

Once code execution reaches `main()`, execution will be stopped and GDB prompt returns.

```
Breakpoint 1, main () at
↪<ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.c:27
27              printk("%s():enter\n", __func__);
```

Now GDB is waiting at the beginning of `main()`:

```
(gdb) list
22
23      int main(void)
24      {
25              int ret;
26
27              printk("%s():enter\n", __func__);
28              ret = test();
29              printk("ret=%d\n", ret);
30              return 0;
31      }
```

7. To examine the value of `ret`, the command `p` or `print` can be used.

```
(gdb) p ret
$1 = 1273788
```

Since `ret` has not been initialized, it contains some random value.

8. If step (`s` or `step`) is used here, it will continue execution skipping the interior of `test()`. To examine code execution inside `test()`, a breakpoint can be set for `test()`, or simply using `si` (or `stepi`) to execute one machine instruction, where it has the side effect of going into the function. The GDB command `finish` can be used to continue execution without intervention until the function returns.

```
(gdb) finish
Run till exit from #0  test () at
↪<ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.c:17
0x00100667 in main () at
```

(continues on next page)

(continued from previous page)

```
↪<ZEPHYR_BASE>/tests/subsys/debug/gdbstub/src/main.c:28
28              ret = test();
Value returned is $2 = 30
```

9. Examine `ret` again which should have the return value from `test()`. Sometimes, the assignment is not done until another `step` is issued, as in this case. This is due to the assignment code is done after returning from function. The assignment code is generated by the toolchain as machine instructions which are not visible when viewing the corresponding C source file.

```
(gdb) p ret
$3 = 1273788
(gdb) step
29              printk("ret=%d\n", ret);
(gdb) p ret
$4 = 30
```

10. If `continue` is issued here, code execution will continue indefinitely as there are no breakpoints to further stop execution. Breaking execution in GDB via `Ctrl-C` does not currently work as the Zephyr gdbstub does not support this functionality yet. Switch to the first console with QEMU running the Zephyr image and stop it manually with `Ctrl+a x`. When the same test is executed by Twister, it automatically takes care of stopping the QEMU instance.

Cortex-M Debug Monitor

Monitor mode debugging is a Cortex-M feature, that provides a non-halting approach to debugging. With this it's possible to continue the execution of high-priority interrupts, even when waiting on a breakpoint. This strategy makes it possible to debug time-sensitive software, that would otherwise crash when the core halts (e.g. applications that need to keep communication links alive).

Zephyr provides support for enabling and configuring the Debug Monitor exception. It also contains a ready implementation of the interrupt, which can be used with SEGGER J-Link debuggers.

Configuration Configure this module using the following options.

- `CONFIG_CORTEX_M_DEBUG_MONITOR_HOOK`: enable the module. This option, by itself, requires an implementation of debug monitor interrupt that will be executed every time the program enters a breakpoint.

With a SEGGER debug probe, it's possible to use a ready, SEGGER-provided implementation of the interrupt.

- `CONFIG_SEGGER_DEBUGMON`: enables SEGGER debug monitor interrupt. Can be used with SEGGER JLinkGDBServer and a SEGGER debug probe.

Usage When monitor mode debugging is enabled, entering a breakpoint will not halt the processor, but rather generate an interrupt with ISR implemented under `z_arm_debug_monitor` symbol. `CONFIG_CORTEX_M_DEBUG_MONITOR_HOOK` config configures this interrupt to be the lowest available priority, which will allow other interrupts to execute while processor spins on a breakpoint.

Using SEGGER-provided ISR The ready implementation provided with `CONFIG_SEGGER_DEBUGMON` provides functionality required to debug in the monitor mode using regular GDB commands. Steps to configure SEGGER debug monitor:

1. Build a sample with `CONFIG_CORTEX_M_DEBUG_MONITOR_HOOK` and `CONFIG_SEGGER_DEBUGMON` configs enabled.
2. Attach JLink GDB server to the target. Example linux command: `JLinkGDBServerCLExe -device <device> -if swd`.
3. Connect to the server with your GDB installation. Example linux command: `arm-none-eabi-gdb --ex="file build/zephyr.elf" --ex="target remote localhost:2331"`.
4. Enable monitor mode debugging in GDB using command: `monitor exec SetMonModeDebug=1`.

After these steps use regular gdb commands to debug your program.

Using other custom ISR In order to provide a custom debug monitor interrupt, override `z_arm_debug_monitor` symbol. Additionally, manual configuration of some registers is required (see debugmon sample).

MIPI STP Decoder

The MIPI System Trace Protocol (MIPI STP) was developed as a generic base protocol that can be shared by multiple application-specific trace protocols. It serves as a wrapper protocol that merges disparate streams that typically contain different trace protocols from different trace sources. Stream consists of opcode (shortest is 4 bit long) followed by optional data and optional timestamp. There are opcodes for data (8, 16, 32, 64 bit data marked/not marked, with or without timestamp), stream recognition (master and channel), synchronization (ASYNC opcode) and others.

One example where protocol is used is ARM Coresight STM (System Trace Macrocell) where data written to Stimulus Port registers maps directly to STP stream.

This module can be used to perform on-chip decoding of the data stream. STP v2 is used.

Usage Decoder is initialized with a callback. A callback is called on each decoded opcode. Decoder has internal state since there are dependency between opcodes (e.g. timestamp can be relative). Decoder can be in synchronization or not. Initial state is configurable. If decoder is not synchronized to the stream then it decodes each nibble in search for ASYNC opcode. Loss of synchronization can be indicated to the decoder by calling `mipi_stp_decoder_sync_loss()`. `mipi_stp_decoder_decode()` is used to decode the data.

Limitations There are following limitations:

- Decoder supports only little endian architectures.
- When decoding nibbles, it is more efficient when core supports unaligned memory access. Implementation supports optimized version with unaligned memory access and generic one. Optimized version is used for ARM Cortex-M (expect for M0).
- Limited set of the most common opcodes is implemented.

API documentation
STP Decoder API

Symbol Table (Symtab)

The Symtab module, when enabled, will generate full symbol table during the Zephyr linking stage that keep tracks of the information about the functions' name and address, for advanced application with a lot of functions, this is expected to consume a sizable amount of ROM.

Currently, this is being used to look up the function names during a stack trace in supported architectures.

Usage Application can gain access to the symbol table data structure by including the `symtab.h` header file and call `symtab_get()`. For now, we only provide `symtab_find_symbol_name()` function to look-up the symbol name and offset of an address. More advanced functionalities and be achieved by directly accessing the members of the data structure.

Configuration Configure this module using the following options.

- `CONFIG_SYMTAB`: enable the generation of the symbol table.

API documentation
Symbol Table API

ARM Coresight Trace Deformatter

Formatter is a method of wrapping multiple trace streams (specified by 7 bit ID) into a single output stream. Formatter is using 16 byte frames which wraps up to 15 bytes of data. It is used, for example, by ETR (Embedded Trace Router) which is a circular RAM buffer where data from various trace streams can be saved. Typically tracing data is decoded offline by the host but deformatter can be used on-chip to decode the data during application runtime.

Usage Deformatter is initialized with a user callback. Data is decoded using `cs_trace_defmt_process()` in 16 bytes chunks. Callback is called whenever stream changes or end of chunk is reached. Callback contains stream ID and the data.

API documentation
Coresight Trace Deformatter

Zyclictest

The zyclictest module enables estimation of worst case latencies of realtime threads. It can measure the time from the hardware interrupt to the service routine and further to the Zephyr thread.

This module and its name is inspired from the cyclictest program on Linux. That's why it's a Zephyr cyclictest.

The idea is to use the timer interrupt as interrupt source as we know exactly the point of time when it occurred. In the interrupt service routine we take the time and calculate the difference to the programmed time of the timer interrupt.

Further a thread is also synchronizing itself on the timer and also measures the difference to the programmed timer.

Both times are put into an array which serves as data source of the histogram printed out at the end of the measurement.

This measurement is done in a cyclic manner with a certain interval time.

If someone wants to know the worst case latency of a task with priority <p> then the application needs to be running. While it is running we can start zyclictest with a priority at least one digit less than the application thread probed. For example if our application with the thread of interest has the priority -10 than we need to start zyclictest with a priority of -11 or less.

Another important argument is the interval time. It's not meant to have an interval less than the measured worst case latency. Therefore it's recommended to set up an interval with at least double the time of the expected or measured worst case latency.

If there are overflows indicated at the end of the measurement than this means that there is no deterministic worst case latency inside the range of the histogram.

For a meaningful output it's recommended to set up `CONFIG_SYS_CLOCK_TICKS_PER_SEC` to at least 1000000 as this means a minimum resolution of 1 microsecond. A tickless kernel is also needed (`CONFIG_TICKLESS_KERNEL`:).

Mode of operation Zyclictest can be started free running without a fixed number of cycles. This is the default. When stopped the result so far is printed out.

In the loop mode (option start with -l <loops>) zyclictest is running for a predefined number of cycles. When the number of cycles is reached there is a message on the shell indicating the end of the test. But the test can be canceled prematurely with zyclictest stop -c.

Configuration Configure this module using the following options.

- `CONFIG_ZYCLICTEST_SHELL`: enable the shell command.

Usage

zyclictest start [options]

-i <interval>	Interval in microseconds
-l <loops>	Use loop mode with predefined number of cycles
-p <prio>	Setup priority of Thread

zyclictest stop [options]

-c	Cancel loop mode prematurely
-q	Quiet mode, print summary, but no histogram data

Example In this example we want to know the worst case latency of a thread which is woken up by an interrupt and running as cooperative task with priority -10. We expect a latency of less then 200 us. We are using the free running mode.

1. Start the zyclictest thread with an interval of 400 us (double of expected worst case latency) and a priority of -11 (one digit higher than the application thread):

```
zyclictest start -i 400 -p -11
```

2. Do whatever should be tested....
3. Stop the measurement:

```
zyclictest stop
```

Output from zyclictest:

```
Count: 547329
                    IRQ  Thread
Max-Latency:         21      27
Errors:               0       0
Overflow:             0       0
Histogram:
[...]
 23                   0  547306
 24                   0       2
 25                   0       5
 26                   0       5
```

(continues on next page)

(continued from previous page)

```
 27                    0        2
 28                    0        0
[...]
```

4. Interpret the result:

   Where are no errors and no overflows which means the measurement can be used. There were 547329 cycles in the test. Worst case interrupt latency is 21 us and for threads it's 27 us.

4.4.4 Profiling

Required Kconfig: `CONFIG_PROFILING`

Perf

Perf is a profiler tool based on stack tracing. It can be used for lightweight profiling with minimal code overhead.

Work Principle The `perf record` shell command starts a timer with the perf tracer function. Timers are driven by interrupts, so the perf tracer function is called during an interruption. The Zephyr core saves the return address and frame pointer in the interrupt stack, `callee_saved` structure, or architecture-specific exception frame before calling the interrupt handler. Thus, the perf trace function makes stack traces by using the return address and frame pointer.

On Cortex-M, perf wraps the SysTick handler so it can sample the interrupted Thread-mode Process Stack Pointer (PSP) frame before the normal timer ISR uses the handler stack. The backend validates that frame before passing it to the Arm stack walker.

The Cortex-M backend is unavailable for Non-secure Trusted Execution images because Secure exception frames are inaccessible to Non-secure firmware.

The scripts/profiling/stackcollapse.py script can be used to convert return addresses in the stack trace to function names using symbols from the ELF file, and to prints them in the format expected by FlameGraph.

Configuration You can configure this module using the following options:

- `CONFIG_PROFILING_PERF`: Enables the module. This option adds the `perf` command to the shell.
- `CONFIG_PROFILING_PERF_BUFFER_SIZE`: Sets the size of the perf buffer where samples are saved before printing.

Architecture backends may require additional stack-unwind support. The Cortex-M backend requires SysTick, thread stack information, extra exception information, Arm stack walking support, and a uniprocessor configuration.

Usage Refer to the profiling-perf sample for an example of how to use the perf tool.

4.4.5 Instrumentation

Overview

The instrumentation subsystem provides compiler-managed runtime system instrumentation capabilities for Zephyr applications. It enables developers to trace function calls, observe context switches, and profile application performance with minimal manual instrumentation effort.

Unlike the *tracing* subsystem, which provides RTOS-aware tracing with structured event APIs, the instrumentation subsystem works at a lower level by leveraging compiler instrumentation hooks. This approach makes it possible to capture virtually any function entry and exit events without requiring manual tracing calls in the code.

Tracing vs. Instrumentation

When to use Tracing: Choose the tracing subsystem when you need RTOS-aware event tracing (e.g. thread switches, semaphore operations, etc.) and want to minimize overhead.

When to use Instrumentation: Choose instrumentation when you need a detailed view of function-level execution to better understand code flow, or to identify performance bottlenecks without adding manual trace points.

The instrumentation subsystem relies on compiler support for automatic function instrumentation. When enabled, the compiler automatically inserts calls to special instrumentation handler functions at the entry and exit of every function in your application (excluding those explicitly marked with `__no_instrumentation__`). Currently, only GCC is supported with the `-finstrument-functions` compiler flag.

The subsystem initializes automatically after RAM initialization and uses trigger/stopper functions to control when recording is active. The default trigger and stopper functions are both set to `main()` (configurable via Kconfig), meaning instrumentation captures the entire execution from when `main()` starts until it returns.

The recorded data is stored in RAM and can be accessed from a host computer thanks to a UART backend that exposes a set of simple commands. scripts/instrumentation/zaru.py script allows to execute these commands through a high-level command-line interface and makes it easy to obtain data in a format suitable for further analysis (e.g. using Perfetto).

Operational Modes

The instrumentation subsystem supports two modes that can be enabled independently or together:

Callgraph Mode (Tracing) In callgraph mode (enabled with `CONFIG_INSTRUMENTATION_MODE_CALLGRAPH`), the subsystem records function entry and exit events along with timestamps and context information in a memory buffer. This enables:

- Reconstruction of the complete function call graph
- Observation of thread context switches
- Analysis of execution flow and timing relationships

The trace buffer can operate in ring buffer mode (default, overwrites old entries) or fixed buffer mode (stops when full). Buffer size is configurable via `CONFIG_INSTRUMENTATION_MODE_CALLGRAPH_TRACE_BUFFER_SIZE`.

Listing 11: Example of callgraph mode output. See *zaru.py Usage* for more details.

```
$ ./scripts/instrumentation/zaru.py trace

   Thread Name      Thread ID  CPU  Mode     Timestamp          Function(s)
-------------------------------------------------------------------------------
↪-------------
            ... (truncated) ...

            main    0x20001a38   0)    0 |    187837720 ns |
```

(continues on next page)

(continued from previous page)

```
↪sys_dlist_append();
            main    0x20001a38   0)    0 |    188802680 ns |             };   /*␣
↪z_priq_simple_add */
            main    0x20001a38   0)    0 |    189282840 ns |           };   /*␣
↪add_to_waitq_locked */
            main    0x20001a38   0)    0 |    189770000 ns |          ␣
↪add_thread_timeout();
            main    0x20001a38   0)    0 |    190732920 ns |         };   /*␣
↪pend_locked */
            main    0x20001a38   0)    0 |    191198480 ns |        ␣
↪k_spin_release();
            main    0x20001a38   0)    0 |    192125560 ns |         z_swap() {
            main    0x20001a38   0)    0 |    192590080 ns |          ␣
↪k_spin_release();
            main    0x20001a38   0)    0 |    193520000 ns |          ␣
↪z_swap_irqlock() {
            main    0x20001a38   0)    0 |    193987840 ns |             __
↪set_BASEPRI() {
            main    0x20001a38   0)    0 |    194474640 ns | /* --> Scheduler␣
↪switched OUT from thread 'main' */
    thread-none   none-thread   0)    0 |    195178000 ns | /* <-- Scheduler␣
↪switched IN thread 'thread-none' */
    thread-none   none-thread   0)    0 |    195851520 ns | z_thread_entry() {
    thread-none   none-thread   0)    0 |    196312600 ns |  ␣
↪k_sched_current_thread_query() {
    thread-none   none-thread   0)    0 |    196774680 ns |    ␣
↪z_impl_k_sched_current_thread_query();
    thread-none   none-thread   0)    0 |    197694480 ns |   };   /*␣
↪k_sched_current_thread_query */
        thread_A    0x200000d8   0)    7 |    198160000 ns | thread_A() {
        thread_A    0x200000d8   0)    7 |    198443400 ns |  ␣
↪get_sem_and_exec_function() {
        thread_A    0x200000d8   0)    7 |    198727440 ns |     k_sem_take() {
        thread_A    0x200000d8   0)    7 |    199011840 ns |      ␣
↪z_impl_k_sem_take() {
        thread_A    0x200000d8   0)    7 |    199397520 ns |         k_spin_lock()
↪{
        thread_A    0x200000d8   0)    7 |    199784200 ns |           __
↪get_BASEPRI();
        thread_A    0x200000d8   0)    7 |    200557840 ns |           __
↪set_BASEPRI_MAX();
        thread_A    0x200000d8   0)    7 |    201333640 ns |           __ISB();
        thread_A    0x200000d8   0)    7 |    202111360 ns |          ␣
↪z_spinlock_validate_pre();
        thread_A    0x200000d8   0)    7 |    202891000 ns |          ␣
↪z_spinlock_validate_post();
        thread_A    0x200000d8   0)    7 |    203664760 ns |         };   /*␣
↪k_spin_lock */
        thread_A    0x200000d8   0)    7 |    204058000 ns |        ␣
↪k_spin_unlock() {
        thread_A    0x200000d8   0)    7 |    204450840 ns |           __
↪set_BASEPRI();
        thread_A    0x200000d8   0)    7 |    205231640 ns |           __ISB();
        thread_A    0x200000d8   0)    7 |    206009600 ns |         };   /*␣
↪k_spin_unlock */
        thread_A    0x200000d8   0)    7 |    206291600 ns |       };   /*␣
```

(continues on next page)

(continued from previous page)

```
↪z_impl_k_sem_take */
        thread_A    0x200000d8   0)    7 |    206572920 ns |     };   /*␣
↪k_sem_take */

        ... (truncated) ...
```

Statistical Mode (Profiling) In statistical mode (enabled with `CONFIG_INSTRUMENTATION_MODE_STATISTICAL`), the subsystem accumulates timing statistics for each unique function executed between the trigger and stopper points. This provides total execution time per function and helps identify performance bottlenecks. The subsystem tracks up to `CONFIG_INSTRUMENTATION_MODE_STATISTICAL_MAX_NUM_FUNC` unique functions.

Listing 12: Example of statistical mode output (top 10 most expensive functions). See *zaru.py Usage* for more details.

```
$ ./scripts/instrumentation/zaru.py profile -n 10

9.45% 0000061d main
6.00% 0000049d k_msleep
5.98% 00000469 k_sleep
5.95% 0000aea1 k_sleep_ticks
5.93% 0000ad6d z_impl_k_sleep_ticks
5.66% 00000431 k_sem_take
5.65% 00007e65 z_impl_k_sem_take
5.51% 0000ac29 z_pend_curr
2.83% 000063ed sys_clock_isr
2.67% 0000d361 sys_clock_announce
```

Configuration

Enable instrumentation with:

```
CONFIG_INSTRUMENTATION=y
CONFIG_INSTRUMENTATION_MODE_CALLGRAPH=y    # For tracing
CONFIG_INSTRUMENTATION_MODE_STATISTICAL=y  # For profiling
```

The instrumentation subsystem communicates with the target device via a UART console. Ensure that the `zephyr_console` chosen node points to the desired UART controller.

Retained memory allows trigger/stopper function addresses to persist across reboots. This feature is optional and enabled with the `CONFIG_INSTRUMENTATION_DYNAMIC_TRIGGER` Kconfig option. Once enabled, devicetree must specify a retained memory region:

```
/ {
    sram@2003FC00 {
        compatible = "zephyr,memory-region", "mmio-sram";
        reg = <0x2003FC00 DT_SIZE_K(1)>;
        zephyr,memory-region = "RetainedMem";

        retainedmem {
            compatible = "zephyr,retained-ram";
            status = "okay";

            instrumentation_triggers: retention@0 {
                compatible = "zephyr,retention";
                status = "okay";
```

(continues on next page)

(continued from previous page)

```
                reg = <0x0 0x10>;
            };
        };
    };
};

/* Adjust main SRAM to exclude retained region */
&sram0 {
    reg = <0x20000000 DT_SIZE_K(255)>;
};
```

See the instrumentation sample for complete configuration examples. Additional options include buffer sizes, trigger functions, and function/file exclusion lists (see Kconfig options starting with `CONFIG_INSTRUMENTATION_*`).

`zaru.py` Usage

The `zaru.py` command-line tool (located in scripts/instrumentation/zaru.py) provides an interface for controlling instrumentation and extracting data from the target over UART.

The tool offers several commands:

- `status`: Check if the target device supports
  - callgraph (tracing) mode
  - statistical (profiling) mode
  - dynamic trigger/stopper functions configuration
- `trace`: Capture and display function call traces.
- `profile`: Capture and display function profiling data.
- `reboot`: Reboot the target device.

You can get help for each command by running `zaru.py <command> --help`.

By default, `zaru.py` attempts to connect to the target device using `/dev/ttyACM0`. You can specify a different serial port using the `--serial` option:

```
$ ./scripts/instrumentation/zaru.py --serial /dev/ttyACM1 status
```

The `--build-dir` option can be used to specify the Zephyr build directory, which is needed to locate the ELF file for symbol resolution. If not provided, `zaru.py` will attempt to find it automatically.

See the instrumentation sample documentation for detailed usage instructions.

Limitations and Considerations

Compiler support
The instrumentation subsystem requires GCC with `-finstrument-functions` support. Other compilers are not supported.

Stack size requirements
Instrumentation adds overhead to every function call, which increases stack usage. You will likely need to increase thread stack sizes to accommodate the additional space required by instrumentation handlers and nested function calls.

Execution overhead
All function calls incur instrumentation overhead. Code size will increase due to added instrumentation calls, and performance will be impacted.

Initialization constraints
Code that runs before RAM initialization (e.g., early boot functions) is not captured as it runs before the instrumentation subsystem is initialized.

To reduce overhead, use trigger/stopper functions to instrument only code regions of interest, and exclude performance-critical functions via `CONFIG_INSTRUMENTATION_EXCLUDE_FUNCTION_LIST` and `CONFIG_INSTRUMENTATION_EXCLUDE_FILE_LIST`.

API Reference

Instrumentation

Related code samples

- InstrumentationDemonstrate the instrumentation subsystem tracing and profiling features.

4.4.6 CPU load

The CPU load module tracks the fraction of time the CPU spends outside of the idle state. Two measurement backends are available, selected with the `CPU_LOAD_BACKEND` Kconfig choice:

Scheduler runtime statistics
`CONFIG_CPU_LOAD_BACKEND_RUNTIME_STATS` derives the load from the scheduler per-CPU runtime statistics. It is portable across architectures and supports multiple CPUs.

Architecture idle hooks
`CONFIG_CPU_LOAD_BACKEND_IDLE_HOOK` measures the load using the architecture idle hooks, which are called before and after the CPU goes to idle. It has lower overhead than the runtime-statistics backend and can use a *Counter* device for higher precision (the counter path is single-CPU only). It is available on any architecture that emits the idle hooks (`CONFIG_ARCH_HAS_CPU_IDLE_HOOKS`). Compared to *Thread analyzer* it is more accurate since it takes into account time spent in the interrupt context as well. This backend does not depend on the tracing subsystem.

The load is retrieved with `cpu_load_get()` for the current CPU or `cpu_load_get_cpu()` for a specific CPU. Both return the load in per mille (0...1000) and can reset the measurement window. Use `CPU_LOAD_PERMILLE_TO_PERCENT` to convert the value to whole percent.

The load can also be reported periodically using a logging message. The period is configured using `CONFIG_CPU_LOAD_LOG_PERIODICALLY`.

For an example refer to the cpu_freq_on_demand sample.

Using a counter device

The idle-hook backend uses `k_cycle_get_32()` by default. When higher precision is needed a *Counter* device can be used by enabling `CONFIG_CPU_LOAD_USE_COUNTER` and setting the chosen node in devicetree.

```
chosen {
  zephyr,cpu-load-counter = &counter_device;
};
```

4.4.7 Shell

- *Overview*
- *Backends*
  - *Telnet*
  - *USB CDC ACM*
  - *Bluetooth LE (NUS)*
  - *Segger RTT*
- *Commands*
  - *Commonly-used command groups*
  - *Creating commands*
  - *Dictionary commands*
  - *Commands execution*
  - *Built-in commands*
- *Alias support*
- *Tab Feature*
- *History Feature*
- *Wildcards Feature*
- *Meta Keys Feature*
- *sys_getopt Feature*
- *Obscured Input Feature*
- *Reading User Input*
- *Shell Logger Backend Feature*
- *RTT Backend Channel Selection*
- *Remote Shell*
  - *Configuration*
  - *Limitations*
- *Usage*
- *API Reference*

Overview

This module allows you to create and handle a shell with a user-defined command set. You can use it in examples where more than simple button or LED user interaction is required. This module is a Unix-like shell with these features:

- Support for multiple instances.
- Advanced cooperation with the *Logging*.
- Support for static and dynamic commands.
- Support for dictionary commands.
- Smart command completion with the `Tab` key.
- Built-in commands: `aliases`, `clear`, `shell`, `colors`, `echo`, `history` and `resize`.

- Viewing recently executed commands using keys: ↑ ↓ or meta keys.
- Text edition using keys: ←, →, `Backspace`, `Delete`, `End`, `Home`, `Insert`.
- Support for ANSI escape codes: `VT100` and `ESC[n~` for cursor control and color printing.
- Support for editing multiline commands.
- Built-in handler to display help for the commands.
- Support for wildcards: `*` and ?.
- Support for meta keys.
- Support for getopt and getopt_long.
- Kconfig configuration to optimize memory usage.

> **Note**
>
> Some of these features have a significant impact on RAM and flash usage, but many can be disabled when not needed. To default to options which favor reduced RAM and flash requirements instead of features, you should enable `CONFIG_SHELL_MINIMAL` and selectively enable just the features you want.

Backends

The module can be connected to any transport for command input and output. At this point, the following transport layers are implemented:

- MQTT
- Segger RTT
- SMP
- Telnet
- UART
- USB
- Bluetooth LE (NUS)
- RPMSG
- DUMMY - not a physical transport layer.

Telnet Enabling `CONFIG_SHELL_BACKEND_TELNET` will allow users to use telnet as a shell backend. Connecting to it can be done using PuTTY or any `telnet` client. For example:

```
telnet <ip address> <port>
```

By default the telnet client won't handle telnet commands and configuration. Although command support can be enabled with `CONFIG_SHELL_TELNET_SUPPORT_COMMAND`. This will give the telnet client access to a very limited set of supported commands but still can be turned on if needed. One of the command options it supports is the `ECHO` option. This will allow the client to be in character mode (character at a time), similar to a UART backend in that regard. This will make the client send a character as soon as it is typed having the effect of increasing the network traffic considerably. For that cost, it will enable the line editing, *tab completion*, and *history* features of the shell.

USB CDC ACM To configure Shell USB CDC ACM backend, simply add the snippet `cdc-acm-console` to your build:

```
west build -S cdc-acm-console [...]
```

Details on the configuration settings are captured in the following files:

- snippets/cdc-acm-console/cdc-acm-console.conf.
- snippets/cdc-acm-console/cdc-acm-console.overlay.

Bluetooth LE (NUS) To configure Bluetooth LE (NUS) backend, simply add the snippet `nus-console` to your build:

```
west build -S nus-console [...]
```

Details on the configuration settings are captured in the following files:

- snippets/nus-console/nus-console.conf.
- snippets/nus-console/nus-console.overlay.

Segger RTT To configure Segger RTT backend, add the following configurations to your build:

- `CONFIG_USE_SEGGER_RTT`
- `CONFIG_SHELL_BACKEND_RTT`
- `CONFIG_SHELL_BACKEND_SERIAL`

Details on additional configuration settings are captured in: samples/subsys/shell/shell_module/prj_minimal_rtt.conf.

Using west Attach to and configure RTT with:

```
$ west rtt
```

Note

If your default runner does not have support for RTT, check your board's documentation page for any other runners that support RTT. You may then use the `--runner` option to specify a different runner.

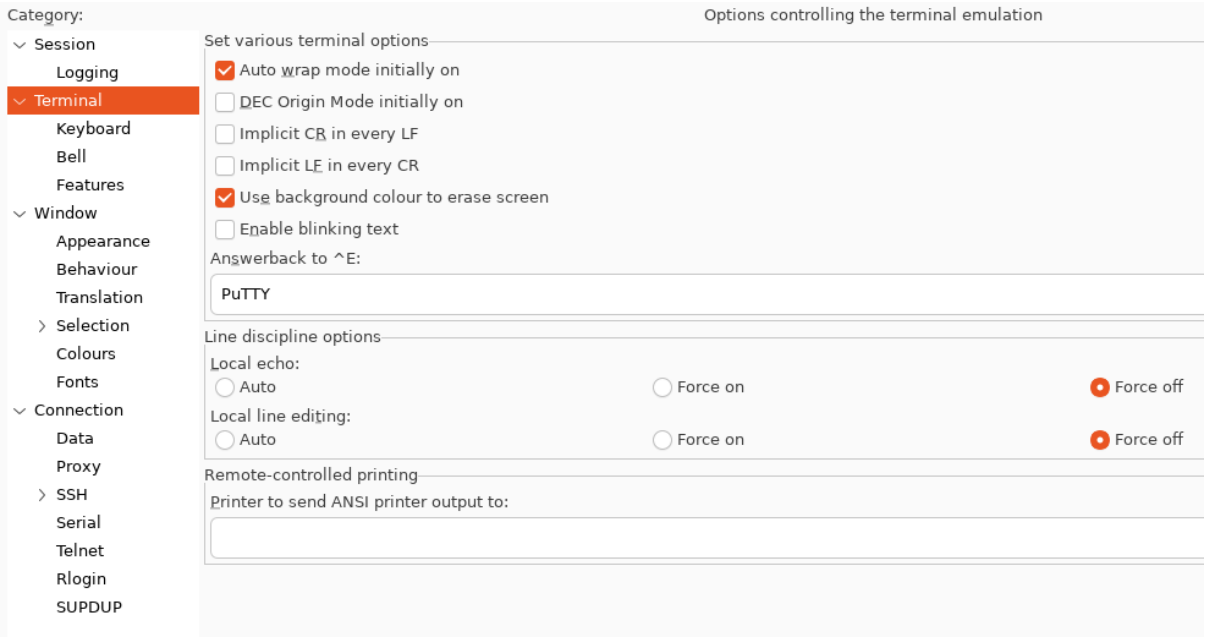
```
$ west rtt --runner <runner>
```

Using PuTTY Use following procedure:

- Open debug session and continue running the application.

  ```
  west attach
  ```

- Open `PuTTY`. Use telnet port 19021 and specific Terminal configuration. Set `Local echo` to `Force off` and `Local line editing` to `Force off` (see image below).



- Now you should have a network connection to RTT that will let you enter input to the shell.

Connecting to Segger RTT via TCP (on macOS, for example) On macOS JLinkRTTClient won't let you enter input. Instead, please use following procedure:

- Open up a first Terminal window and enter:

```
JLinkRTTLogger -Device NRF52840_XXAA -RTTChannel 1 -if SWD -Speed 4000 ~/
↪rtt.log
```

  (change device if required)

- Open up a second Terminal window and enter:

```
nc localhost 19021
```

- Now you should have a network connection to RTT that will let you enter input to the shell. However, contrary to *PuTTY* some features like `Tab` completion do not work.

Commands

Shell commands are organized in a tree structure and grouped into the following types:

- Root command (level 0): Gathered and alphabetically sorted in a dedicated memory section.
- Static subcommand (level > 0): Number and syntax must be known during compile time. Created in the software module.
- Dynamic subcommand (level > 0): Number and syntax does not need to be known during compile time. Created in the software module.

Commonly-used command groups The following list is a set of useful command groups and how to enable them:

GPIO

- `CONFIG_GPIO`
- `CONFIG_GPIO_SHELL`

I2C

- CONFIG_I2C
- CONFIG_I2C_SHELL

Sensor

- CONFIG_SENSOR
- CONFIG_SENSOR_SHELL

Flash

- CONFIG_FLASH
- CONFIG_FLASH_SHELL

File-System

- CONFIG_FILE_SYSTEM
- CONFIG_FILE_SYSTEM_SHELL

ADC

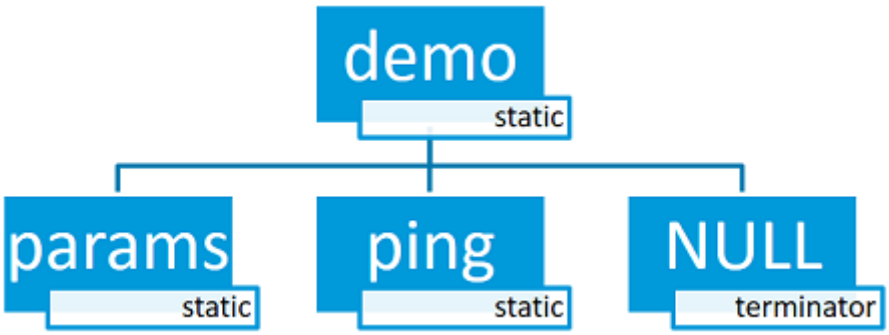
- CONFIG_ADC
- CONFIG_ADC_SHELL
- CONFIG_ADC_EMUL_SHELL (requires CONFIG_ADC_EMUL)

Creating commands Use the following macros for adding shell commands:

- SHELL_CMD_REGISTER - Create root command. All root commands must have different name.
- SHELL_COND_CMD_REGISTER - Conditionally (if compile time flag is set) create root command. All root commands must have different name.
- SHELL_CMD_ARG_REGISTER - Create root command with arguments. All root commands must have different name.
- SHELL_COND_CMD_ARG_REGISTER - Conditionally (if compile time flag is set) create root command with arguments. All root commands must have different name.
- SHELL_CMD - Initialize a command.
- SHELL_COND_CMD - Initialize a command if compile time flag is set.
- SHELL_EXPR_CMD - Initialize a command if compile time expression is non-zero.
- SHELL_CMD_ARG - Initialize a command with arguments.
- SHELL_COND_CMD_ARG - Initialize a command with arguments if compile time flag is set.
- SHELL_EXPR_CMD_ARG - Initialize a command with arguments if compile time expression is non-zero.
- SHELL_STATIC_SUBCMD_SET_CREATE - Create a static subcommands array.
- SHELL_SUBCMD_DICT_SET_CREATE - Create a dictionary subcommands array.
- SHELL_DYNAMIC_CMD_CREATE - Create a dynamic subcommands array.

Commands can be created in any file in the system that includes include/zephyr/shell/shell.h. All created commands are available for all shell instances.

Static commands Example code demonstrating how to create a root command with static subcommands.




```
/* Creating subcommands (level 1 command) array for command "demo". */
SHELL_STATIC_SUBCMD_SET_CREATE(sub_demo,
        SHELL_CMD(params, NULL, "Print params command.",
                                                cmd_demo_params),
        SHELL_CMD(ping,   NULL, "Ping command.", cmd_demo_ping),
        SHELL_SUBCMD_SET_END
);
/* Creating root (level 0) command "demo" */
SHELL_CMD_REGISTER(demo, &sub_demo, "Demo commands", NULL);
```

Example implementation can be found under following location: samples/subsys/shell/shell_module/src/main.c.

Dictionary commands This is a special kind of static commands. Dictionary commands can be used every time you want to use a pair: (string <-> corresponding data) in a command handler. The string is usually a verbal description of a given data. The idea is to use the string as a command syntax that can be prompted by the shell and corresponding data can be used to process the command.

Let's use an example. Suppose you created a command to set an ADC gain. It is a perfect place where a dictionary can be used. The dictionary would be a set of pairs: (string: gain_value, int: value) where int value could be used with the ADC driver API.

Abstract code for this task would look like this:

```
static int gain_cmd_handler(const struct shell *sh,
                            size_t argc, char **argv, void *data)
{
        int gain;

        /* data is a value corresponding to called command syntax */
        gain = (int)data;
        adc_set_gain(gain);

        shell_print(sh, "ADC gain set to: %s\n"
                        "Value send to ADC driver: %d",
                        argv[0],
                        gain);

        return 0;
}

SHELL_SUBCMD_DICT_SET_CREATE(sub_gain, gain_cmd_handler,
```

(continues on next page)

(continued from previous page)

```
        (gain_1, 1, "gain 1"), (gain_2, 2, "gain 2"),
        (gain_1_2, 3, "gain 1/2"), (gain_1_4, 4, "gain 1/4")
);
SHELL_CMD_REGISTER(gain, &sub_gain, "Set ADC gain", NULL);
```

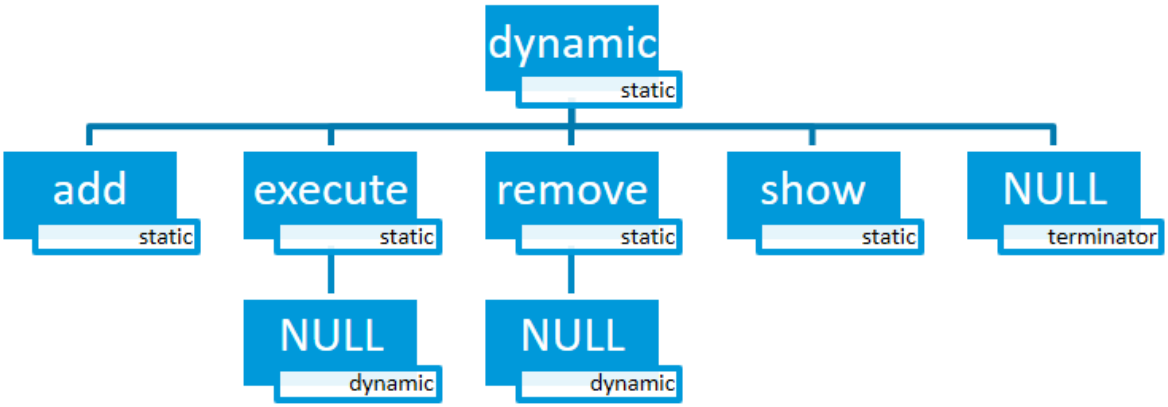
This is how it would look like in the shell:

```
uart:~$ gain ga
  gain_1    gain_2    gain_1_2  gain_1_4
uart:~$ gain gain_1
ADC gain set to: gain_1
Value send to ADC driver: 1
uart:~$ gain gain_2
ADC gain set to: gain_2
Value send to ADC driver: 2
uart:~$ gain gain_1_2
ADC gain set to: gain_1_2
Value send to ADC driver: 3
uart:~$ gain gain_1_4
ADC gain set to: gain_1_4
Value send to ADC driver: 4
uart:~$
```

Dynamic commands Example code demonstrating how to create a root command with static and dynamic subcommands. At the beginning dynamic command list is empty. New commands can be added by typing:

```
dynamic add <new_dynamic_command>
```

Newly added commands can be prompted or autocompleted with the `Tab` key.




```
/* Buffer for 10 dynamic commands */
static char dynamic_cmd_buffer[10][50];
```

(continues on next page)

(continued from previous page)

```
/* commands counter */
static uint8_t dynamic_cmd_cnt;

/* Function returning command dynamically created
 * in  dynamic_cmd_buffer.
 */
static void dynamic_cmd_get(size_t idx,
                            struct shell_static_entry *entry)
{
        if (idx < dynamic_cmd_cnt) {
                entry->syntax = dynamic_cmd_buffer[idx];
                entry->handler  = NULL;
                entry->subcmd = NULL;
                entry->help = "Show dynamic command name.";
        } else {
                /* if there are no more dynamic commands available
                 * syntax must be set to NULL.
                 */
                entry->syntax = NULL;
        }
}

SHELL_DYNAMIC_CMD_CREATE(m_sub_dynamic_set, dynamic_cmd_get);
SHELL_STATIC_SUBCMD_SET_CREATE(m_sub_dynamic,
        SHELL_CMD(add, NULL,"Add new command to dynamic_cmd_buffer and"
                  " sort them alphabetically.",
                  cmd_dynamic_add),
        SHELL_CMD(execute, &m_sub_dynamic_set,
                  "Execute a command.", cmd_dynamic_execute),
        SHELL_CMD(remove, &m_sub_dynamic_set,
                  "Remove a command from dynamic_cmd_buffer.",
                  cmd_dynamic_remove),
        SHELL_CMD(show, NULL,
                  "Show all commands in dynamic_cmd_buffer.",
                  cmd_dynamic_show),
        SHELL_SUBCMD_SET_END
);
SHELL_CMD_REGISTER(dynamic, &m_sub_dynamic,
           "Demonstrate dynamic command usage.", cmd_dynamic);
```

Example implementation can be found under following location: samples/subsys/shell/shell_module/src/dynamic_cmd.c.

Commands execution Each command or subcommand may have a handler. The shell executes the handler that is found deepest in the command tree and further subcommands (without a handler) are passed as arguments. Characters within parentheses are treated as one argument. If shell won't find a handler it will display an error message.

Commands can be also executed from a user application using any active backend and a function `shell_execute_cmd()`, as shown in this example:

```
int main(void)
{
        /* Below code will execute "clear" command on a DUMMY backend */
        shell_execute_cmd(NULL, "clear");
```

(continues on next page)

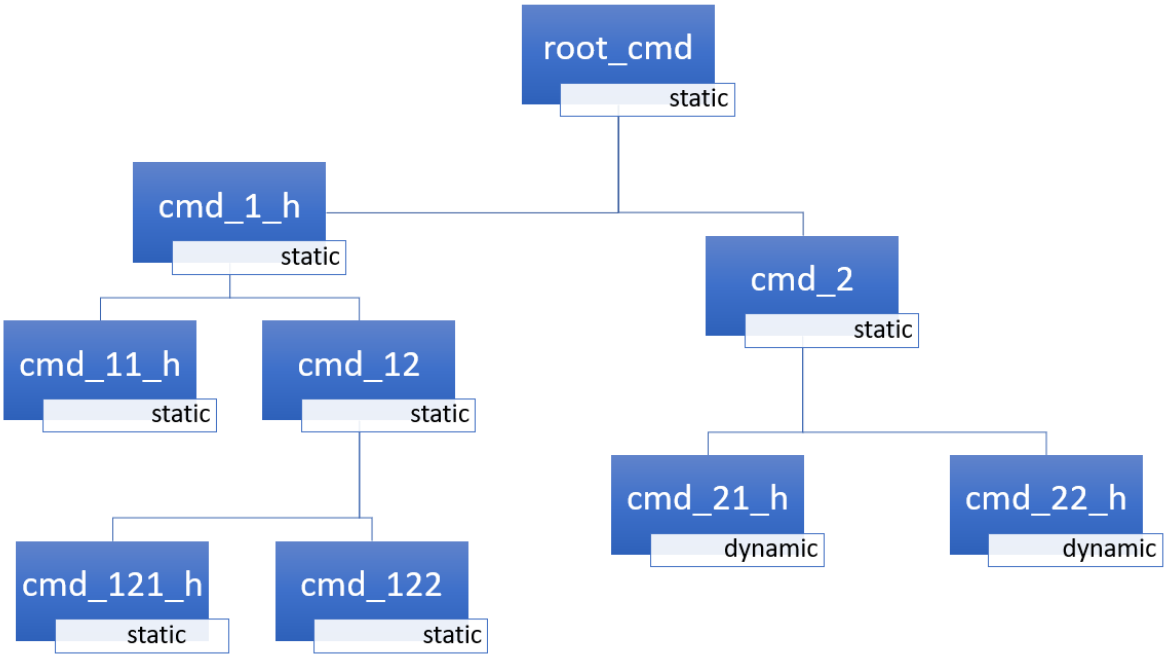
(continued from previous page)

```
        /* Below code will execute "shell colors off" command on
         * an UART backend
         */
        shell_execute_cmd(shell_backend_uart_get_ptr(),
                          "shell colors off");
}
```

Enable the DUMMY backend by setting the Kconfig `CONFIG_SHELL_BACKEND_DUMMY` option.

Commands execution example Let's assume a command structure as in the following figure, where:

- `root_cmd` - root command without a handler
- `cmd_xxx_h` - command has a handler
- `cmd_xxx` - command does not have a handler



Example 1 Sequence: `root_cmd cmd_1_h cmd_12_h cmd_121_h parameter` will execute command `cmd_121_h` and `parameter` will be passed as an argument.

Example 2 Sequence: `root_cmd cmd_2 cmd_22_h parameter1 parameter2` will execute command `cmd_22_h` and `parameter1 parameter2` will be passed as an arguments.

Example 3 Sequence: `root_cmd cmd_1_h parameter1 cmd_121_h parameter2` will execute command `cmd_1_h` and `parameter1`, `cmd_121_h` and `parameter2` will be passed as an arguments.

Example 4 Sequence: `root_cmd parameter cmd_121_h parameter2` will not execute any command.

Command handler Simple command handler implementation:

```
static int cmd_handler(const struct shell *sh, size_t argc,
                       char **argv)
{
        ARG_UNUSED(argc);
        ARG_UNUSED(argv);

        shell_fprintf(sh, SHELL_INFO, "Print info message\n");

        shell_print(sh, "Print simple text.");

        shell_warn(sh, "Print warning text.");

        shell_error(sh, "Print error text.");

        return 0;
}
```

Function `shell_fprintf()` or the shell print macros: `shell_print`, `shell_info`, `shell_warn` and `shell_error` can be used from the command handler or from threads, but not from an interrupt context. Instead, interrupt handlers should use *Logging* for printing.

Command help Every user-defined command or subcommand can have its own help description. The help for commands and subcommands can be created with respective macros: `SHELL_CMD_REGISTER`, `SHELL_CMD_ARG_REGISTER`, `SHELL_CMD`, and `SHELL_CMD_ARG`.

Shell prints this help message when you call a command or subcommand with `-h` or `--help` parameter.

Parent commands In the subcommand handler, you can access both the parameters passed to commands or the parent commands, depending on how you index `argv`.

- When indexing `argv` with positive numbers, you can access the parameters.
- When indexing `argv` with negative numbers, you can access the parent commands.
- The subcommand to which the handler belongs has the `argv` index of 0.

```
static int cmd_handler(const struct shell *sh, size_t argc,
                       char **argv)
{
        ARG_UNUSED(argc);

        /* If it is a subcommand handler parent command syntax
         * can be found using argv[-1].
         */
        shell_print(sh, "This command has a parent command: %s",
                    argv[-1]);

        /* Print this command syntax */
        shell_print(sh, "This command syntax is: %s", argv[0]);

        /* Print first argument */
        shell_print(sh, "%s", argv[1]);

        return 0;
}
```

Built-in commands These commands are activated by CONFIG_SHELL_CMDS set to y.

- **aliases** - Shows currently configured aliases. This command needs extra activation: CONFIG_SHELL_CMDS_ALIASES set to y.
- **clear** - Clears the screen.
- **history** - Shows the recently entered commands.
- **resize** - Must be executed when terminal width is different than 80 characters or after each change of terminal width. It ensures proper multiline text display and ←, →, End, Home keys handling. Currently this command works only with UART flow control switched on. It can be also called with a subcommand:
  - **default** - Shell will send terminal width = 80 to the terminal and assume successful delivery.

  These command needs extra activation: CONFIG_SHELL_CMDS_RESIZE set to y.
- **select** - It can be used to set new root command. Exit to main command tree is with alt+r. This command needs extra activation: CONFIG_SHELL_CMDS_SELECT set to y.
- **shell** - Root command with useful shell-related subcommands like:
  - **echo** - Toggles shell echo.
  - **colors** - Toggles colored syntax. This might be helpful in case of Bluetooth shell to limit the amount of transferred bytes.
  - **stats** - Shows shell statistics.

Alias support

The shell can expand aliases before executing a command when CONFIG_SHELL_ALIASES is enabled. Zephyr always provides the built-in alias **?**, which expands to **help**.

To load additional aliases from a host file, set CONFIG_SHELL_ALIASES_FILE and generate the corresponding include file from the application CMakeLists.txt file. When CONFIG_SHELL_ALIASES_FILE is non-empty, CONFIG_SHELL_ALIASES_HAS_FILE is enabled automatically.

```
if(CONFIG_SHELL_ALIASES_HAS_FILE)
  set(gen_dir ${ZEPHYR_BINARY_DIR}/include/generated/)
  set(aliases_src ${CONFIG_SHELL_ALIASES_FILE})

  generate_shell_aliases_inc_file_for_target(
    app
    ${aliases_src}
    ${gen_dir}/generated-shell-aliases.inc
  )
endif()
```

The aliases file uses one alias=command entry per line. Lines beginning with # are comments, and command strings containing spaces must be quoted. The sample at samples/net/sockets/echo_service/shell_aliases.txt uses the following format:

```
# Example shell aliases file
stacks="kernel thread stacks"
scan="wifi scan"
connect="wifi connect"
```

Use the **aliases** command to inspect the aliases available in the running shell.

Tab Feature

The Tab button can be used to suggest commands or subcommands. This feature is enabled by `CONFIG_SHELL_TAB` set to `y`. It can also be used for partial or complete auto-completion of commands. This feature is activated by `CONFIG_SHELL_TAB_AUTOCOMPLETION` set to `y`. When user starts writing a command and presses the `Tab` button then the shell will do one of 3 possible things:

- Autocomplete the command.
- Prompts available commands and if possible partly completes the command.
- Will not do anything if there are no available or matching commands.

```
uart:~$ log
  backend        disable        enable         go             halt
  list_backends  status
uart:~$ log enable
  dbg   err   inf   none  wrn
uart:~$ log enable err
  app                app_test         log              os
  shell.shell_uart   shell_uart
uart:~$ log enable err shell
  shell.shell_uart   shell_uart
uart:~$ log enable err shell
```

History Feature

This feature enables commands history in the shell. It is activated by: `CONFIG_SHELL_HISTORY` set to `y`. History can be accessed using keys: ↑↓ or `Ctrl+n` and `Ctrl+p` if meta keys are active. Number of commands that can be stored depends on size of `CONFIG_SHELL_HISTORY_BUFFER` parameter.

Wildcards Feature

The shell module can handle wildcards. Wildcards are interpreted correctly when expanded command and its subcommands do not have a handler. For example, if you want to set logging level to `err` for the `app` and `app_test` modules you can execute the following command:

```
log enable err a*
```

```
uart:~$ log status
module_name                                | current | built-in
-----------------------------------------------------------
app                                        | inf     | inf
app_test                                   | inf     | inf
shell.uart_shell                           | inf     | inf
uart:~$ log enable err a*
uart:~$ log status
module_name                                | current | built-in
-----------------------------------------------------------
app                                        | err     | inf
app_test                                   | err     | inf
shell.uart_shell                           | inf     | inf
uart:~$
```

This feature is activated by `CONFIG_SHELL_WILDCARD` set to `y`.

> **Note**
>
> When the wildcard feature is enabled, the characters `*` and `?` cannot be used as normal characters in commands or arguments.

Meta Keys Feature

The shell module supports the following meta keys:

Table 52: Implemented meta keys

Meta keys	Action
`Ctrl+a`	Moves the cursor to the beginning of the line.
`Ctrl+b`	Moves the cursor backward one character.
`Ctrl+c`	Preserves the last command on the screen and starts a new command in a new line.
`Ctrl+d`	Deletes the character under the cursor.
`Ctrl+e`	Moves the cursor to the end of the line.
`Ctrl+f`	Moves the cursor forward one character.
`Ctrl+k`	Deletes from the cursor to the end of the line.
`Ctrl+l`	Clears the screen and leaves the currently typed command at the top of the screen.
`Ctrl+n`	Moves in history to next entry.
`Ctrl+p`	Moves in history to previous entry.
`Ctrl+t`	Toggles logs output on the shell when `CONFIG_SHELL_LOG_BACKEND` is set.
`Ctrl+u`	Clears the currently typed command.
`Ctrl+w`	Removes the word or part of the word to the left of the cursor. Words separated by period instead of space are treated as one word.
`Alt+b`	Moves the cursor backward one word.
`Alt+f`	Moves the cursor forward one word.

This feature is activated by `CONFIG_SHELL_METAKEYS` set to `y`.

sys_getopt Feature

Some shell users apart from subcommands might need to use options as well. the arguments string, looking for supported options. Typically, this task is accomplished by the `getopt` family functions.

For this purpose shell supports a variant of the getopt and getopt_long libraries from the FreeBSD project called sys_getopt and sys_getopt_long. This feature is activated by: `CONFIG_GETOPT` set to `y` and `CONFIG_GETOPT_LONG` set to `y`.

This feature can be used in thread safe as well as non thread safe manner. The former is full compatible with regular getopt usage while the latter a bit differs.

An example non-thread safe usage:

```
char *cvalue = NULL;
while ((char c = sys_getopt(argc, argv, "abhc:")) != -1) {
      switch (c) {
      case 'c':
              cvalue = sys_getopt_optarg;
              break;
      default:
```

(continues on next page)

(continued from previous page)

```
                break;
        }
}
```

An example thread safe usage:

```
char *cvalue = NULL;
struct sys_getopt_state *state;
while ((char c = sys_getopt(argc, argv, "abhc:")) != -1) {
        state = sys_getopt_state_get();
        switch (c) {
        case 'c':
                cvalue = state->optarg;
                break;
        default:
                break;
        }
}
```

Thread safe sys_getopt functionality is activated by `CONFIG_SHELL_GETOPT` set to `y`.

Obscured Input Feature

With the obscured input feature, the shell can be used for implementing a login prompt or other user interaction whereby the characters the user types should not be revealed on screen, such as when entering a password.

Once the obscured input has been accepted, it is normally desired to return the shell to normal operation. Such runtime control is possible with the `shell_obscure_set` function.

An example of login and logout commands using this feature is located in samples/subsys/shell/shell_module/src/main.c and the config file samples/subsys/shell/shell_module/prj_login.conf.

This feature is activated upon startup by `CONFIG_SHELL_START_OBSCURED` set to `y`. With this set either way, the option can still be controlled later at runtime. `CONFIG_SHELL_CMDS_SELECT` is useful to prevent entry of any other command besides a login command, by means of the `shell_set_root_cmd` function. Likewise, `CONFIG_SHELL_PROMPT_UART` allows you to set the prompt upon startup, but it can be changed later with the `shell_prompt_change` function.

Reading User Input

The shell provides a `shell_readline()` function that allows command handlers to interactively read a line of input from the user. This is useful for implementing commands that need to prompt the user for additional information, such as confirmation dialogs, multi-step wizards, or interactive data entry.

The function reads from the shell transport until a newline character is received, storing the data in a provided buffer. The newline character itself is not included in the buffer. The buffer is automatically null-terminated on success.

Use `shell_readline_prompt_set()` before calling `shell_readline()` to display a prompt string (e.g. `"Proceed? [y/N]: "`). The prompt is printed automatically at the start of readline and restored after any log messages that interrupt the input line. It is cleared when `shell_readline()` returns.

Note

The `shell_readline()` function should be called from the shell thread in a shell command handler and blocks the thread until a result is returned.

Example usage:

```
static int cmd_read_secret(const struct shell *sh, size_t argc, char **argv)
{
        uint8_t input_buf[256];
        int ret;

        shell_obscure_set(sh, true);
        shell_readline_prompt_set(sh, "Enter your secret: ");
        ret = shell_readline(sh, input_buf, sizeof(input_buf), K_SECONDS(10));
        shell_obscure_set(sh, false);

        if (ret < 0) {
                if (ret == -ETIMEDOUT) {
                        shell_error(sh, "Timeout waiting for input");
                } else if (ret == -ECANCELED) {
                        shell_error(sh, "Input canceled");
                } else if (ret == -ENOBUFS) {
                        shell_error(sh, "Input too long");
                }
                return ret;
        }

        shell_print(sh, "Secret input received (%d bytes)", ret);
        return 0;
}
```

Shell Logger Backend Feature

Shell instance can act as the *Logging* backend. Shell ensures that log messages are correctly multiplexed with shell output. Log messages from logger thread are enqueued and processed in the shell thread. Logger thread will block for configurable amount of time if queue is full, blocking logger thread context for that time. Oldest log message is removed from the queue after timeout and new message is enqueued. Use the `shell stats show` command to retrieve number of log messages dropped by the shell instance. Log queue size and timeout are `SHELL_DEFINE` arguments.

This feature is activated by: `CONFIG_SHELL_LOG_BACKEND` set to `y`.

Warning

Enqueuing timeout must be set carefully when multiple backends are used in the system. The shell instance could have a slow transport or could block, for example, by a UART with hardware flow control. If timeout is set too high, the logger thread could be blocked and impact other logger backends.

Warning

As the shell is a complex logger backend, it can not output logs if the application crashes before the shell thread is running. In this situation, you can enable one of the simple logging backends instead, such as UART (`CONFIG_LOG_BACKEND_UART`) or RTT (`CONFIG_LOG_BACKEND_RTT`), which are available earlier during system initialization.

Note

The shell log backend is enabled by `shell_start()`. With `CONFIG_SHELL_AUTOSTART` set to `y` this happens in the shell thread, when it is scheduled for the first time. The shell thread runs at a low priority by default, so this moment depends on when higher priority threads, including the main thread executing system initialization, block or finish. Log messages and `printk()` output routed through the logger (`CONFIG_LOG_PRINTK`) issued before that point are handled by the logging core, not by the shell:

- In deferred mode (`CONFIG_LOG_MODE_DEFERRED`) they are buffered and printed once the shell starts, as long as they fit in `CONFIG_LOG_BUFFER_SIZE`.
- In immediate mode (`CONFIG_LOG_MODE_IMMEDIATE`) they are lost, because there is no active backend to output them.

If the application needs early output on the shell, set `CONFIG_SHELL_AUTOSTART` to `n` and call `shell_start()` explicitly, for example from a `SYS_INIT` hook with a priority higher than the shell backend initialization priority (e.g. `CONFIG_SHELL_BACKEND_SERIAL_INIT_PRIORITY`). Alternatively, do not route `printk()` through the logger.

RTT Backend Channel Selection

Instead of using the shell as a logger backend, RTT shell backend and RTT log backend can also be used simultaneously, but over different channels. By separating them, the log can be captured or monitored without shell output or the shell may be scripted without log interference. Enabling both the Shell RTT backend and the Log RTT backend does not work by default, because both default to channel `0`. There are two options:

1. The Shell buffer can use an alternate channel, for example using `CONFIG_SHELL_BACKEND_RTT_BUFFER` set to `1`. This allows monitoring the log using JLinkRTTViewer while a script interfaces over channel 1.

2. The Log buffer can use an alternate channel, for example using `CONFIG_LOG_BACKEND_RTT_BUFFER` set to `1`. This allows interactive use of the shell through JLinkRTTViewer, while the log is written to file.

See *shell backends* for details on how to enable RTT as a Shell backend.

Remote Shell

The shell module supports remote shell. Remote shell allows to execute commands on the remote core using the main core shell interface. User interface features like `Tab` autocompletion or history navigation are supported on the remote shell client. There can be multiple remote clients connected to the main core shell. Each remote client has its own tree of commands. Remote client commands are executed in the context of the remote client core. Remote shell is using *IPC service* to communicate with the remote core.

If only a single remote client is enabled then `remote_shell` command is registered as a root command for the remote core tree of commands. If multiple remote clients are enabled then `remote_shell <core_name>` is used as a root for the given remote client.

Remote client implementation supports case where threads are disabled (`CONFIG_MULTITHREADING` set to `n`).

This approach has two main benefits:

1. It allows to use the same shell interface for both the main and the remote cores.
2. Full shell implementation is present only on the host core, significantly reducing the memory footprint of the remote cores. Implementation uses less than 3 kB per remote client. Half of the memory is used for string formatting and can be shared with other modules that use string formatting (for example, logging).

See shell-module for details on how to enable remote shell.

Configuration Following Kconfig options are available for remote shell host configuration:

`CONFIG_SHELL_REMOTE`: Enables remote shell on the host core.

`CONFIG_SHELL_REMOTE_MULTI_CLI`: Enables multiple remote clients.

`CONFIG_SHELL_REMOTE_TMP_BUF_SIZE`: Size of the temporary buffer for remote shell command data. Need to be large enough to hold the command and its arguments.

Following Kconfig options are available for remote shell client configuration:

`CONFIG_SHELL_REMOTE_CLI`: Enables remote shell client on the remote core.

`CONFIG_SHELL_REMOTE_CLI_BUF_SIZE`: Size of the internal buffer used for remote messages. Need to be large enough to hold the command syntax and help text. Help text is truncated if the buffer is too small. It is also used to hold data for shell print messages. If it is too small then the print message is replaced with the error message.

`CONFIG_SHELL_REMOTE_CLI_KWORK`: Use kwork context for command execution.

`CONFIG_SHELL_REMOTE_CLI_STACK_SIZE`: Thread stack size for remote shell client.

If a single client is used then IPC service device is automatically chosen based on the devicetree configuration. User can specify the IPC device tree node using chosen node. If not specified then if there is only a single IPC device available that one is used.

IPC service device tree node configuration example:

```
chosen {
        zephyr,ipc_shell = &ipc0;
};
```

Limitations Some shell features are not supported on the remote shell client:

- Bypass mode is not supported on the remote shell client.
- Select mode is not supported on the remote shell client.

IPC communication assumes same endianness and alignment of the data on both sides. It has been tested on the following core pairs:

- Cortex-M33 (host) and Cortex-M33 (client)
- Cortex-M33 (host) and RISC-V 32 (client)

Usage

The following code shows a simple use case of this library:

```
int main(void)
{

}

static int cmd_demo_ping(const struct shell *sh, size_t argc,
                         char **argv)
{
        ARG_UNUSED(argc);
        ARG_UNUSED(argv);

        shell_print(sh, "pong");
        return 0;
```

(continues on next page)

(continued from previous page)

```
}

static int cmd_demo_params(const struct shell *sh, size_t argc,
                           char **argv)
{
        int cnt;

        shell_print(sh, "argc = %d", argc);
        for (cnt = 0; cnt < argc; cnt++) {
                shell_print(sh, "  argv[%d] = %s", cnt, argv[cnt]);
        }
        return 0;
}

/* Creating subcommands (level 1 command) array for command "demo". */
SHELL_STATIC_SUBCMD_SET_CREATE(sub_demo,
        SHELL_CMD(params, NULL, "Print params command.",
                                                cmd_demo_params),
        SHELL_CMD(ping,   NULL, "Ping command.", cmd_demo_ping),
        SHELL_SUBCMD_SET_END
);
/* Creating root (level 0) command "demo" without a handler */
SHELL_CMD_REGISTER(demo, &sub_demo, "Demo commands", NULL);

/* Creating root (level 0) command "version" */
SHELL_CMD_REGISTER(version, NULL, "Show kernel version", cmd_version);
```

Users may use the Tab key to complete a command/subcommand or to see the available subcommands for the currently entered command level. For example, when the cursor is positioned at the beginning of the command line and the Tab key is pressed, the user will see all root (level 0) commands:

```
clear  demo  shell  history  log  resize  version
```

> **Note**
>
> To view the subcommands that are available for a specific command, you must first type a space after this command and then hit Tab.

These commands are registered by various modules, for example:

- **clear**, **shell**, **history**, and **resize** are built-in commands which have been registered by subsys/shell/shell.c
- **demo** and **version** have been registered in example code above by main.c
- **log** has been registered by subsys/logging/log_cmds.c

Then, if a user types a **demo** command and presses the Tab key, the shell will only print the subcommands registered for this command:

```
params  ping
```

API Reference

Shell

Related code samples

- Custom Shell moduleRegister shell commands using the Shell API
- devmem load shellLoad data to device memory using devmem shell cmd.
- Telnet consoleAccess Zephyr shell over telnet.

4.4.8 Task Watchdog

Overview

Many microcontrollers feature a hardware watchdog timer peripheral. Its purpose is to trigger an action (usually a system reset) in case of severe software malfunctions. Once initialized, the watchdog timer has to be restarted ("fed") in regular intervals to prevent it from timing out. If the software got stuck and does not manage to feed the watchdog anymore, the corrective action is triggered to bring the system back to normal operation.

In real-time operating systems with multiple tasks running in parallel, a single watchdog instance may not be sufficient anymore, as it can be used for only one task. This software watchdog based on kernel timers provides a method to supervise multiple threads or tasks (called watchdog channels).

An existing hardware watchdog can be used as an optional fallback if the task watchdog itself or the scheduler has a malfunction.

The task watchdog uses a kernel timer as its backend. If configured properly, the timer ISR is never actually called during normal operation, as the timer is continuously updated in the feed calls.

It's currently not possible to have multiple instances of task watchdogs. Instead, the task watchdog API can be accessed globally to add or delete new channels without passing around a context or device pointer in the firmware.

The maximum number of channels is predefined via Kconfig and should be adjusted to match exactly the number of channels required by the application.

Configuration Options

Related configuration options can be found under subsys/task_wdt/Kconfig.

- `CONFIG_TASK_WDT`
- `CONFIG_TASK_WDT_CHANNELS`
- `CONFIG_TASK_WDT_HW_FALLBACK`
- `CONFIG_TASK_WDT_MIN_TIMEOUT`
- `CONFIG_TASK_WDT_HW_FALLBACK_DELAY`

API Reference

Task Watchdog APIs

Related code samples

- Task watchdogMonitor a thread using a task watchdog.

Storage & Configuration

4.5 Storage & Configuration

4.5.1 Disk Access

Overview

The disk access API provides access to storage devices.

Initializing Disks

Since many disk devices (such as SD cards) are hotpluggable, the disk access API provides IOCTLs to initialize and de-initialize the disk. They are as follows:

- `DISK_IOCTL_CTRL_INIT`: Initialize the disk. Must be called before additional I/O operations can be run on the disk device. Equivalent to calling the legacy function `disk_access_init()`.
- `DISK_IOCTL_CTRL_DEINIT`: De-initialize the disk. Once this IOCTL is issued, the `DISK_IOCTL_CTRL_INIT` must be issued before the disk can be used for addition I/O operations.

Init/deinit IOCTL calls are balanced, so a disk will not de-initialize until an equal number of deinit IOCTLs have been issued as init IOCTLs.

It is also possible to force a disk de-initialization by passing a pointer to a boolean set to `true` as a parameter to the `DISK_IOCTL_CTRL_DEINIT` IOCTL. This is an unsafe operation which each disk driver may handle differently, but it will always return a value indicating success.

Note that de-initializing a disk is a low level operation- typically the de-initialization and initialization calls should be left to the filesystem implementation, and the user application should not need to manually de-initialize the disk and can instead call `fs_unmount()`

SD Card support

Zephyr has support for some SD card controllers and support for interfacing SD cards via SPI. These drivers use disk driver interface and a file system can access the SD cards via disk access API. Both standard and high-capacity SD cards are supported.

> **Note**
>
> FAT filesystems are not power safe so the filesystem may become corrupted if power is lost or if the card is removed without unmounting the filesystem

SD Memory Card subsystem Zephyr supports SD memory cards via the disk driver API, or via the SDMMC subsystem. This subsystem can be used transparently via the disk driver API, but also supports direct block level access to cards. The SDMMC subsystem interacts with the *sd host controller api* to communicate with attached SD cards.

SD Card support via SPI Example devicetree fragment below shows how to add SD card node to `spi1` interface. Example uses pin `PA27` for chip select, and runs the SPI bus at 24 MHz once the SD card has been initialized:

```
&spi1 {
        status = "okay";
        cs-gpios = <&porta 27 GPIO_ACTIVE_LOW>;

        sdhc0: sdhc@0 {
                compatible = "zephyr,sdhc-spi-slot";
```

(continues on next page)

(continued from previous page)

```
                reg = <0>;
                status = "okay";
                mmc {
                    compatible = "zephyr,sdmmc-disk";
                    disk-name = "SD";
                    status = "okay";
                };
                spi-max-frequency = <24000000>;
        };
};
```

The SD card will be automatically detected and initialized by the filesystem driver when the board boots.

To read and write files and directories, see the *File Systems* in include/zephyr/fs/fs.h such as `fs_open()`, `fs_read()`, and `fs_write()`.

eMMC Device Support

Zephyr also has support for eMMC devices using the Disk Access API. MMC in zephyr is implemented using the SD subsystem because the MMC bus shares a lot of similarity with the SD bus. MMC controllers also use the SDHC device driver API.

Emulated block device on flash partition support

Zephyr flashdisk driver makes it possible to use flash memory partition as a block device. The flashdisk instances are defined in devicetree:

```
/ {
    msc_disk0 {
        compatible = "zephyr,flash-disk";
        partition = <&storage_partition>;
        disk-name = "NAND";
        cache-size = <4096>;
    };
};
```

The cache size specified in `zephyr,flash-disk` node should be equal to backing partition minimum erasable block size.

NVMe disk support

NVMe disks are also supported

NVMe

NVMe is a standardized logical device interface on PCIe bus exposing storage devices.

NVMe controllers and disks are supported. Disks can be accessed via the *Disk Access API* they expose and thus be used through the *File System API*.

Driver design

The driver is sliced up in 3 main parts:

- NVMe controller: drivers/disk/nvme/nvme_controller.c
- NVMe commands: drivers/disk/nvme/nvme_cmd.c
- NVMe namespace: drivers/disk/nvme/nvme_namespace.c

Where the NVMe controller is the root of the device driver. This is the one that will get device driver instances. Note that this is only what DTS describes: the NVMe controller, and none of its namespaces (disks). The NVMe command is the generic logic used to communicate with the controller and the namespaces it exposes. Finally the NVMe namespace is the dedicated part to

deal with an actual namespace which, in turn, enables applications accessing each ones through the Disk Access API drivers/disk/nvme/nvme_disk.c.

If a controller exposes more than 1 namespace (disk), it will be possible to raise the amount of built-in namespace support by tweaking the configuration option CONFIG_NVME_MAX_NAMESPACES (see below).

Each exposed disk, via it's related disk_info structure, will be distinguished by its name which is inherited from it's related namespace. As such, the disk name follows NVMe naming which is nvme<k>n<n> where k is the controller number and n the namespame number. Most of the time, if only one NVMe disk is plugged into the system, one will see 'nvme0n0' as an exposed disk.

NVMe configuration

DTS Any board exposing an NVMe disk should provide a DTS overlay to enable its use within Zephyr

```
#include <zephyr/dt-bindings/pcie/pcie.h>
/ {
    pcie0 {
        nvme0: nvme0 {
            compatible = "nvme-controller";
            vendor-id = <VENDOR_ID>;
            device-id = <DEVICE_ID>;
            status = "okay";
        };
    };
};
```

Where VENDOR_ID and DEVICE_ID are the ones from the exposed NVMe controller.

Options

- `CONFIG_NVME`

Note that NVME requires the target to support PCIe multi-vector MSI-X in order to function.

- `CONFIG_NVME_MAX_NAMESPACES`

Important note for users NVMe specifications mandate the data buffer to be placed in a dword (4 bytes) aligned address. While this is not a problem for advanced OS managing virtual memory and dynamic allocations below the user processes, this can become an issue in Zephyr as soon as buffer addresses map directly to physical memory.

At this stage then, it is up to the user to make sure the buffer address being provided to `disk_access_read()` and `disk_access_write()` are dword aligned.

VirtIO block device support

VirtIO block devices are also supported

VirtIO Block VirtIO is a standardized interface for exposing virtual devices to a guest, typically under a hypervisor such as QEMU. The `virtio,blk` driver presents a VirtIO block device as a Zephyr disk: it is reachable through the *Disk Access API* and, through it, the *File System API*.

The driver is transport-agnostic and runs unchanged over both VirtIO transports Zephyr supports, PCI and MMIO (see *VirtIO*). An instance is created for every enabled `virtio,blk` devicetree node.

Driver design The driver lives in drivers/disk/virtio_blk.c and is layered on top of the generic VirtIO API. A `virtio,blk` node is a child of a VirtIO transport device (`DT_INST_PARENT`), so the same code serves a PCI or an MMIO device without change.

Single in-flight request The `disk_access` read/write API is synchronous and blocking. The driver also serializes callers with a mutex, so each device keeps exactly one request in flight: it adds a single descriptor chain to the request virtqueue, notifies the device, and blocks on a semaphore that the completion callback signals from interrupt context.

Because there is no request pipelining, the whole virtqueue is spent on a single request's scatter-gather chain. The user-facing knob is the maximum number of data segments per request (`CONFIG_DISK_VIRTIO_BLK_MAX_SEGMENTS`); the driver sizes the virtqueue by rounding `MAX_SEGMENTS + 2` up to a power of two, leaving room for the request header and the status byte.

Block size The VirtIO block protocol always addresses the device in fixed 512-byte sectors and reports device capacity in those units. The logical block size exposed to `disk_access` may differ:

- When the device offers `VIRTIO_BLK_F_BLK_SIZE`, the driver negotiates it and exposes the device-advertised `blk_size`.
- Otherwise it falls back to `CONFIG_DISK_VIRTIO_BLK_SECTOR_SIZE`.

All addressing is converted to 512-byte VirtIO sectors internally. The block size must be a multiple of 512 and, under `CONFIG_MMU`, must not exceed the MMU page size; an unsupported value is rejected at initialization.

Zero-copy transfers Caller buffers are handed to the device directly, without a bounce buffer. Under `CONFIG_MMU` a buffer may lie outside the kernel's linear RAM mapping (for example the thread-stack work area `f_mkfs()` passes down) and need not be physically contiguous. The driver walks such a buffer page by page with `arch_page_phys_get()`, coalesces physically adjacent runs into scatter-gather segments, and, when a transfer is more fragmented than the segment budget allows, splits it across several device requests.

Feature negotiation Besides `VIRTIO_BLK_F_BLK_SIZE` the driver negotiates:

- `VIRTIO_BLK_F_RO` — a read-only device is reported as `DISK_STATUS_WR_PROTECT` and write requests are rejected with `-EROFS`.
- `VIRTIO_BLK_F_FLUSH` — `DISK_IOCTL_CTRL_SYNC` issues a flush to the device. When the device did not offer `FLUSH` there is no writeback cache to flush, so the ioctl is a successful no-op rather than an error.

A completed request whose status byte is not `OK` is logged and mapped to an errno: an unsupported request becomes `-ENOTSUP` and an I/O error `-EIO`.

VirtIO block configuration

DTS A `virtio,blk` node is placed under its VirtIO transport parent and requires a `disk-name` used by the `disk_access` APIs. Over the PCI transport:

```
#include <zephyr/dt-bindings/pcie/pcie.h>

/ {
    pcie0 {
        virtio-blk-pci {
            compatible = "virtio,pci";
            vendor-id = <0x1af4>;
            device-id = <0x1001>;
```

(continues on next page)

(continued from previous page)

```
            interrupts = <0xb 0x0 0x0>;
            interrupt-parent = <&intc>;
            status = "okay";

            virtio_blk: virtio-blk {
                compatible = "virtio,blk";
                disk-name = "VIRTIOBLK0";
                status = "okay";
            };
        };
    };
};
```

Over the MMIO transport the node is a child of an existing `virtio,mmio` bus:

```
&virtio_mmio4 {
    status = "okay";

    virtio_blk: virtio-blk {
        compatible = "virtio,blk";
        disk-name = "VIRTIOBLK0";
        status = "okay";
    };
};
```

Options

- `CONFIG_DISK_DRIVER_VIRTIO_BLK`
- `CONFIG_DISK_VIRTIO_BLK_MAX_SEGMENTS`
- `CONFIG_DISK_VIRTIO_BLK_SECTOR_SIZE`

QEMU options When `CONFIG_DISK_DRIVER_VIRTIO_BLK` is enabled on a QEMU board, the emulation layer creates a raw backing image and attaches it to the emulated virtio-blk device. PCI transports are attached from the common QEMU emulation code; the `qemu_cortex_a53` board attaches an MMIO `virtio-blk-device` to `virtio-mmio-bus.4`.

- `CONFIG_QEMU_VIRTIO_BLK_LOGICAL_BLOCK_SIZE`
- `CONFIG_QEMU_VIRTIO_BLK_DISK_SIZE`

Limitations

- Exactly one request is in flight at a time. This matches the synchronous `disk_access` contract, so it costs that consumer no throughput, but the driver provides no request pipelining.
- Like the rest of Zephyr's VirtIO subsystem, the driver assumes coherent DMA and performs no cache maintenance on descriptors, rings, or data buffers.
- Under an MMU the logical block size cannot exceed the MMU page size.

Disk Access API Configuration Options

Related configuration options:

- `CONFIG_DISK_ACCESS`

API Reference

Disk Access Interface

Related code samples

- File system manipulationUse file system API with various filesystems and storage devices.

Disk Driver Configuration Options

Related driver configuration options:

- `CONFIG_DISK_DRIVERS`

Disk Driver Interface

Disk Access

4.5.2 Flash Circular Buffer (FCB)

Flash circular buffer provides an abstraction through which you can treat flash like a FIFO. You append entries to the end, and read data from the beginning.

Description

Entries in the flash contain the length of the entry, the data within the entry, and checksum over the entry contents.

Storage of entries in flash is done in a FIFO fashion. When you request space for the next entry, space is located at the end of the used area. When you start reading, the first entry served is the oldest entry in flash.

Entries can be appended to the end of the area until storage space is exhausted. You have control over what happens next; either erase oldest block of data, thereby freeing up some space, or stop writing new data until existing data has been collected. FCB treats underlying storage as an array of flash sectors; when it erases old data, it does this a sector at a time.

Entries in the flash are checksummed. That is how FCB detects whether writing entry to flash completed ok. It will skip over entries which don't have a valid checksum.

Usage

To add an entry to circular buffer:

- Call `fcb_append()` to get the location where data can be written. If this fails due to lack of space, you can call `fcb_rotate()` to erase the oldest sector which will make the space. And then call `fcb_append()` again.
- Use `flash_area_write()` to write entry contents.
- Call `fcb_append_finish()` when done. This completes the writing of the entry by calculating the checksum.

To read contents of the circular buffer:

- Call `fcb_walk()` with a pointer to your callback function.
- Within callback function copy in data from the entry using `flash_area_read()`. You can tell when all data from within a sector has been read by monitoring the returned entry's area pointer. Then you can call `fcb_rotate()`, if you're done with that data.

Alternatively:

- Call `fcb_getnext()` with 0 in entry offset to get the pointer to the oldest entry.
- Use `flash_area_read()` to read entry contents.
- Call `fcb_getnext()` with pointer to current entry to get the next one. And so on.

API Reference

The FCB subsystem APIs are provided by `fcb.h`:

Data structures

Flash Circular Buffer Data Structures

API functions

fcb API

4.5.3 File Systems

Zephyr RTOS Virtual Filesystem Switch (VFS) allows applications to mount multiple file systems at different mount points (e.g., `/fatfs` and `/lfs`). The mount point data structure contains all the necessary information required to instantiate, mount, and operate on a file system. The File system Switch decouples the applications from directly accessing an individual file system's specific API or internal functions by introducing file system registration mechanisms.

In Zephyr, any file system implementation or library can be plugged into or pulled out through a file system registration API. Each file system implementation must have a globally unique integer identifier; use `FS_TYPE_EXTERNAL_BASE` to avoid clashes with in-tree identifiers.

```
int fs_register(int type, const struct fs_file_system_t *fs);

int fs_unregister(int type, const struct fs_file_system_t *fs);
```

Zephyr RTOS supports multiple instances of a file system by making use of the mount point as the disk volume name, which is used by the file system library while formatting or mounting a disk.

A file system is declared as:

```
static struct fs_mount_t mp = {
.type = FS_FATFS,
.mnt_point = FATFS_MNTP,
.fs_data = &fat_fs,
};
```

where

- `FS_FATFS` is the file system type like FATFS or LittleFS.
- `FATFS_MNTP` is the mount point where the file system will be mounted.
- `fat_fs` is the file system data which will be used by fs_mount() API.

Samples

Samples for the VFS are mainly supplied in `samples/subsys/fs`, although various examples of the VFS usage are provided as important functionalities in samples for different subsystems. Here is the list of samples worth looking at:

- fs is an example of FAT file system usage with SDHC media;

- shell-fs is an example of Shell fs subsystem, using internal flash partition formatted to LittleFS;
- usb-mass example of USB Mass Storage device that uses FAT FS driver with RAM or SPI connected FLASH, or LittleFS in flash, depending on the sample configuration.

API Reference

File System APIs

Related code samples

- EXT2 filesystem fstabDefine ext2 filesystems in the devicetree.
- Fatfs filesystem fstabDefine fatfs filesystems in the devicetree.
- File system manipulationUse file system API with various filesystems and storage devices.
- File system shellAccess a LittleFS file system partition in flash using the file system shell.
- Format filesystemFormat different storage devices for different file systems.
- LittleFS filesystemUse file system API over LittleFS.
- USB Mass StorageExpose board's RAM or FLASH as a USB disk using USB Mass Storage driver.
- virtiofs filesystemUse file system API over virtiofs.

4.5.4 Flash map

The `<zephyr/storage/flash_map.h>` API allows accessing information about device flash partitions via `flash_area` structures.

Each `flash_area` describes a flash partition. The API provides access to a "flash map", which contains predefined flash areas accessible via globally unique ID numbers. The map is created from "fixed-partitions" and "zephyr,mapped-partition" compatible entries in DTS file. Users may also create `flash_area` objects at runtime for application-specific purposes.

This documentation uses "flash area" when referencing single "fixed-partitions" or "zephyr,mapped-partition" entities.

The `flash_area` contains a pointer to a `device`, which can be used to access the flash device an area is placed on directly with the *flash API*. Each flash area is characterized by a device it is placed on, offset from the beginning of the device and size on the device. An additional identifier parameter is used by the `flash_area_open()` function to find flash area in flash map.

The flash_map.h API provides functions for operating on a `flash_area`. The main examples are `flash_area_read()` and `flash_area_write()`. These functions are basically wrappers around the flash API with additional offset and size checks, to limit flash operations to a predefined area.

Most `<zephyr/storage/flash_map.h>` API functions require a `flash_area` object pointer characterizing the flash area they will be working on. There are two possible methods to obtain such a pointer:

- obtain it using `flash_area_open()`;
- defining a `flash_area` type object, which requires providing a valid `device` object pointer with offset and size of the area within the flash device.

`flash_area_open()` uses numeric identifiers to search flash map for `flash_area` objects and returns, if found, a pointer to an object representing area with given ID. The ID number for a flash

area can be obtained from a "fixed-partitions" or a "zephyr,mapped-partition" DTS node label using `PARTITION_ID()`; these labels are obtained from the devicetree as described below.

Relationship with Devicetree

The flash_map.h API uses data generated from the *Devicetree API*, in particular its *Fixed and mapped flash partitions*. Zephyr additionally has some partitioning conventions used for *Device Firmware Upgrade* via the MCUboot bootloader, as well as defining partitions usable by *file systems* or other non-volatile *storage*.

Here is an example devicetree fragment which uses fixed flash partitions for both MCUboot and a storage partition. Some details were left out for clarity.

```
/ {
        soc {
                flashctrl: flash-controller@deadbeef {
                        flash0: flash@0 {
                                compatible = "soc-nv-flash";
                                reg = <0x0 0x100000>;

                                partitions {
                                        compatible = "fixed-partitions";
                                        #address-cells = <0x1>;
                                        #size-cells = <0x1>;

                                        boot_partition: partition@0 {
                                                reg = <0x0 0x10000>;
                                                read-only;
                                        };

                                        storage_partition: partition@1e000 {
                                                reg = <0x1e000 0x2000>;
                                        };

                                        slot0_partition: partition@20000 {
                                                reg = <0x20000 0x60000>;
                                        };

                                        slot1_partition: partition@80000 {
                                                reg = <0x80000 0x60000>;
                                        };

                                        scratch_partition: partition@e0000 {
                                                reg = <0xe0000 0x20000>;
                                        };
                                };
                        };
                };
        };
};
```

Partition offset shall be expressed in relation to the flash memory beginning address, to which the partition belongs to.

The `boot_partition`, `slot0_partition`, `slot1_partition`, and `scratch_partition` node labels are defined for MCUboot, though not all MCUboot configurations require all of them to be defined. See the MCUboot documentation for more details.

The `storage_partition` node is defined for use by a file system or other nonvolatile storage API.

Numeric flash area ID is obtained by passing DTS node label to `PARTITION_ID()`; for example to obtain ID number for `slot0_partition`, user would invoke `PARTITION_ID(slot0_partition)`.

All `PARTITION_*` macros take DTS node labels as partition identifiers.

Users do not have to obtain a `flash_area` object pointer using `flash_map_open()` to get information on flash area size, offset or device, if such area is defined in DTS file. Knowing the DTS node label of an area, users may use `PARTITION_OFFSET()`, `PARTITION_SIZE()` or `PARTITION_DEVICE()` respectively to obtain such information directly from DTS node definition. For example to obtain offset of `storage_partition` it is enough to invoke `PARTITION_OFFSET(storage_partition)`.

Below example shows how to obtain a `flash_area` object pointer using `flash_area_open()` and DTS node label:

```
const struct flash_area *my_area;
int err = flash_area_open(PARTITION_ID(slot0_partition), &my_area);

if (err != 0) {
     handle_the_error(err);
} else {
     flash_area_read(my_area, ...);
}
```

API Reference

flash area Interface

Related code samples

- LittleFS filesystemUse file system API over LittleFS.
- nRF SoC Internal StorageUse the flash API to interact with the SoC flash.

4.5.5 Non-Volatile Memory (NVMEM)

The NVMEM subsystem provides a generic interface for accessing non-volatile memory devices. It abstracts the underlying hardware and provides a unified API for reading and writing data.

Key Concepts

NVMEM Provider An NVMEM provider is a driver that exposes NVMEM cells. For example, an EEPROM driver can be an NVMEM provider. The NVMEM provider is responsible for reading and writing data to the underlying hardware.

NVMEM Cell An NVMEM cell is a region of non-volatile memory. It is defined in the devicetree and has properties like offset, size, and read-only status.

NVMEM Consumer An NVMEM consumer is a driver or application that uses NVMEM cells to store or retrieve data.

Configuration

- `CONFIG_NVMEM`: Enables the NVMEM subsystem.
- `CONFIG_NVMEM_BBRAM`: Enables NVMEM support for Battery Backed RAM.
- `CONFIG_NVMEM_EEPROM`: Enables NVMEM support for EEPROM devices.
- `CONFIG_NVMEM_FLASH.*`: Configure NVMEM support for flash devices.

- `CONFIG_NVMEM_OTP.*`: Configure NVMEM support for OTP devices.

Devicetree Bindings

The NVMEM subsystem relies on devicetree bindings to define NVMEM cells. The following is an example of how to define an NVMEM provider and cells in the devicetree:

```
&eeprom0 {
        nvmem-layout {
                compatible = "fixed-layout";
                #address-cells = <1>;
                #size-cells = <1>;

                mac_address: mac_address@0 {
                        reg = <0x0 6>;
                        #nvmem-cell-cells = <0>;
                        read-only;
                };

                calibration_data: calibration_data@6 {
                        reg = <0x6 100>;
                        #nvmem-cell-cells = <0>;
                };
        };
};
```

The reg property is an array containing:

- The offset in the memory in which we are creating the cell,
- The size of the cell, in bytes.

`#nvmem-cell-cells` describes the number of property items in the phandle, see *Specifier cell names (*-cells)*, typically set to zero.

A consumer can then reference the NVMEM cells like this:

```
my_consumer: my-consumer {
        compatible = "my,consumer";
        nvmem-cells = <&mac_address>, <&calibration_data>;
        nvmem-cell-names = "mac-address", "calibration-data";
};
```

Usage Example

The following is an example of how to use the NVMEM API to read data from an NVMEM cell:

```
#include <zephyr/nvmem.h>

static const struct nvmem_cell mac_address =
        NVMEM_CELL_GET_BY_NAME(DT_NODELABEL(my_consumer), mac_address);

int main(void)
{
        uint8_t mac[6];
        int ret;

        if (!nvmem_cell_is_ready(&mac_address)) {
                printk("NVMEM cell is not ready\n");
```

(continues on next page)

(continued from previous page)

```
                return -ENODEV;
        }

        ret = nvmem_cell_read(&mac_address, mac, 0, sizeof(mac));
        if (ret < 0) {
                printk("Failed to read MAC address: %d\n", ret);
                return ret;
        }

        /* ... */
}
```

API Reference

NVMEM

4.5.6 Non-Volatile Storage (NVS)

Elements, represented as id-data pairs, are stored in flash using a FIFO-managed circular buffer. The flash area is divided into sectors. Elements are appended to a sector until storage space in the sector is exhausted. Then a new sector in the flash area is prepared for use (erased). Before erasing the sector it is checked that identifier - data pairs exist in the sectors in use, if not the id-data pair is copied.

The id is a 16-bit unsigned number. NVS ensures that for each used id there is at least one id-data pair stored in flash at all time.

NVS allows storage of binary blobs, strings, integers, longs, and any combination of these.

Each element is stored in flash as metadata (8 byte) and data. The metadata is written in a table starting from the end of a nvs sector, the data is written one after the other from the start of the sector. The metadata consists of: id, data offset in sector, data length, part (unused), and a CRC. This CRC is only calculated over the metadata and only ensures that a write has been completed. The actual data of the element can be protected by a different (and optional) CRC-32. Use the `CONFIG_NVS_DATA_CRC` configuration item to enable the data part CRC.

> **Note**
>
> The data CRC is checked only when the whole data of the element is read. The data CRC is not checked for a partial read, as it is stored at the end of the element data area.

> **Note**
>
> Enabling the data CRC feature on a previously existing NVS content without data CRC will make all existing data invalid.

A write of data to nvs always starts with writing the data, followed by a write of the metadata. Data that is written in flash without metadata is ignored during initialization.

During initialization NVS will verify the data stored in flash, if it encounters an error it will ignore any data with missing/incorrect metadata.

NVS checks the id-data pair before writing data to flash. If the id-data pair is unchanged no write to flash is performed.

To protect the flash area against frequent erases it is important that there is sufficient free space. NVS has a protection mechanism to avoid getting in a endless loop of flash page erases when there is limited free space. When such a loop is detected NVS returns that there is no more space available.

For NVS the file system is declared as:

```
static struct nvs_fs fs = {
.flash_device = NVS_FLASH_DEVICE,
.sector_size = NVS_SECTOR_SIZE,
.sector_count = NVS_SECTOR_COUNT,
.offset = NVS_STORAGE_OFFSET,
};
```

where

- `NVS_FLASH_DEVICE` is a reference to the flash device that will be used. The device needs to be operational.
- `NVS_SECTOR_SIZE` is the sector size, it has to be a multiple of the flash erase page size and a power of 2.
- `NVS_SECTOR_COUNT` is the number of sectors, it is at least 2, one sector is always kept empty to allow copying of existing data.
- `NVS_STORAGE_OFFSET` is the offset of the storage area in flash.

Flash wear

When writing data to flash a study of the flash wear is important. Flash has a limited life which is determined by the number of times flash can be erased. Flash is erased one page at a time and the pagesize is determined by the hardware. As an example a nRF51822 device has a pagesize of 1024 bytes and each page can be erased about 20,000 times.

Calculating expected device lifetime Suppose we use a 4 bytes state variable that is changed every minute and needs to be restored after reboot. NVS has been defined with a sector_size equal to the pagesize (1024 bytes) and 2 sectors have been defined.

Each write of the state variable requires 12 bytes of flash storage: 8 bytes for the metadata and 4 bytes for the data. When storing the data the first sector will be full after 1024/12 = 85.33 minutes. After another 85.33 minutes, the second sector is full. When this happens, because we're using only two sectors, the first sector will be used for storage and will be erased after 171 minutes of system time. With the expected device life of 20,000 writes, with two sectors writing every 171 minutes, the device should last about 171 * 20,000 minutes, or about 6.5 years.

More generally then, with

- `NS` as the number of storage requests per minute,
- `DS` as the data size in bytes,
- `SECTOR_SIZE` in bytes, and
- `PAGE_ERASES` as the number of times the page can be erased,

the expected device life (in minutes) can be calculated as:

```
SECTOR_COUNT * SECTOR_SIZE * PAGE_ERASES / (NS * (DS+8)) minutes
```

From this formula it is also clear what to do in case the expected life is too short: increase `SECTOR_COUNT` or `SECTOR_SIZE`.

Flash write block size migration

It is possible that during a DFU process, the flash driver used by the NVS changes the supported minimal write block size. The NVS in-flash image will stay compatible unless the physical ATE size changes. Especially, migration between 1,2,4,8-bytes write block sizes is allowed.

Sample

A sample of how NVS can be used is supplied in `samples/subsys/kvss/nvs`.

Troubleshooting

MPU fault while using NVS, or `-ETIMEDOUT` error returned
NVS can use the internal flash of the SoC. While the MPU is enabled, the flash driver requires MPU RWX access to flash memory, configured using `CONFIG_MPU_ALLOW_FLASH_WRITE`. If this option is disabled, the NVS application will get an MPU fault if it references the internal SoC flash and it's the only thread running. In a multi-threaded application, another thread might intercept the fault and the NVS API will return an `-ETIMEDOUT` error.

API Reference

The NVS subsystem APIs are provided by `nvs.h`:

Non-volatile Storage Data Structures

Non-volatile Storage APIs

Related code samples

- Non-Volatile Storage (NVS)Store and retrieve data from flash using the NVS API.

4.5.7 Retention System

The retention system provides an API which allows applications to read and write data from and to memory areas or devices that retain the data while the device is powered. This allows for sharing information between different applications or within a single application without losing state information when a device reboots. The stored data should not persist in the event of a power failure (or during some low-power modes on some devices) nor should it be stored to a non-volatile storage like *Flash*, *EEPROM API*, or battery-backed RAM.

The retention system builds on top of the retained data driver, and adds additional software-level features to it for ensuring the validity of data. Optionally, a magic header can be used to check if the front of the retained data memory section contains this specific value, and an optional checksum (1, 2, or 4-bytes in size) of the stored data can be appended to the end of the data. Additionally, the retention system API allows partitioning of the retained data sections into multiple distinct areas. For example, a 64-byte retained data area could be split up into 4 bytes for a boot mode, 16 bytes for a timestamp, 44 bytes for a last log message. All of these sections can be accessed or updated independently. The prefix and checksum can be set per-instance using devicetree.

Devicetree setup

To use the retention system, a retained data driver must be setup for the board you are using, there is a zephyr driver which can be used which will use some RAM as non-init for this purpose. The retention system is then initialised as a child node of this device 1 or more times - note that the memory region will need to be decremented to account for this reserved portion of RAM. See the following example (examples in this guide are based on the nrf52840dk board and memory layout):

```
/ {
        sram@2003FC00 {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x2003FC00 DT_SIZE_K(1)>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
                        #address-cells = <1>;
                        #size-cells = <1>;

                        /* This creates a 256-byte partition */
                        retention0: retention@0 {
                                compatible = "zephyr,retention";
                                status = "okay";

                                /* The total size of this area is 256
                                 * bytes which includes the prefix and
                                 * checksum, this means that the usable
                                 * data storage area is 256 - 3 = 253
                                 * bytes
                                 */
                                reg = <0x0 0x100>;

                                /* This is the prefix which must appear
                                 * at the front of the data
                                 */
                                prefix = [08 04];

                                /* This uses a 1-byte checksum */
                                checksum = <1>;
                        };

                        /* This creates a 768-byte partition */
                        retention1: retention@100 {
                                compatible = "zephyr,retention";
                                status = "okay";

                                /* Start position must be after the end
                                 * of the previous partition. The total
                                 * size of this area is 768 bytes which
                                 * includes the prefix and checksum,
                                 * this means that the usable data
                                 * storage area is 768 - 6 = 762 bytes
                                 */
                                reg = <0x100 0x300>;

                                /* This is the prefix which must appear
                                 * at the front of the data
                                 */
                                prefix = [00 11 55 88 fa bc];

                                /* If omitted, there will be no
                                 * checksum
                                 */
```

(continues on next page)

(continued from previous page)

```
                        };
                };
        };
};

/* Reduce SRAM0 usage by 1KB to account for non-init area */
&sram0 {
        reg = <0x20000000 DT_SIZE_K(255)>;
};
```

The retention areas can then be accessed using the data retention API (once enabled with `CONFIG_RETENTION`, which requires that `CONFIG_RETAINED_MEM` be enabled) by getting the device by using:

```
#include <zephyr/device.h>
#include <zephyr/retention/retention.h>

const struct device *retention0 = DEVICE_DT_GET(DT_NODELABEL(retention0));
const struct device *retention1 = DEVICE_DT_GET(DT_NODELABEL(retention1));
```

When the write function is called, the magic header and checksum (if enabled) will be set on the area, and it will be marked as valid from that point onwards.

Mutex protection

Mutex protection of retention areas is enabled by default when applications are compiled with multithreading support. This means that different threads can safely call the retention functions without clashing with other concurrent thread function usage, but means that retention functions cannot be used from ISRs. It is possible to disable mutex protection globally on all retention areas by enabling `CONFIG_RETENTION_MUTEX_FORCE_DISABLE` - users are then responsible for ensuring that the function calls do not conflict with each other. Note that to use this, retention driver mutex support must also be disabled by enabling `CONFIG_RETAINED_MEM_MUTEX_FORCE_DISABLE`.

Boot mode

An addition to the retention subsystem is a boot mode interface, this can be used to dynamically change the state of an application or run a different application with a minimal set of functions when a device is rebooted (an example is to have a buttonless way of entering mcuboot's serial recovery feature from the main application).

To use the boot mode feature, a data retention entry must exist in the device tree, which is dedicated for use as the boot mode selection (the user area data size only needs to be a single byte), and this area be assigned to the chosen node of `zephyr,boot-mode`. See the following example:

```
/ {
        sram@2003FFFF {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x2003FFFF 0x1>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
                        #address-cells = <1>;
                        #size-cells = <1>;
```

(continues on next page)

(continued from previous page)

```
                        retention0: retention@0 {
                                compatible = "zephyr,retention";
                                status = "okay";
                                reg = <0x0 0x1>;
                        };
                };
        };

        chosen {
                zephyr,boot-mode = &retention0;
        };
};

/* Reduce SRAM0 usage by 1 byte to account for non-init area */
&sram0 {
        reg = <0x20000000 0x3FFFF>;
};
```

The boot mode interface can be enabled with `CONFIG_RETENTION_BOOT_MODE` and then accessed by using the boot mode functions. If using mcuboot with serial recovery, it can be built with `CONFIG_MCUBOOT_SERIAL` and `CONFIG_BOOT_SERIAL_BOOT_MODE` enabled which will allow rebooting directly into the serial recovery mode by using:

```
#include <zephyr/retention/bootmode.h>
#include <zephyr/sys/reboot.h>

bootmode_set(BOOT_MODE_TYPE_BOOTLOADER);
sys_reboot(0);
```

Retention system modules

Modules can expand the functionality of the retention system by using it as a transport (e.g. between a bootloader and application).

Bootloader Information The bootloader information (abbreviated to blinfo) subsystem is an extension of the *Retention System* which allows for reading shared data from a bootloader and allowing applications to query it. It has an optional feature of organising the information retrieved from the bootloader and storing it in the *Settings* with the `blinfo/` prefix.

Devicetree setup To use the bootloader information subsystem, a retention area needs to be created which has a retained data section as its parent, generally non-init RAM is used for this purpose. See the following example (examples in this guide are based on the nrf52840dk board and memory layout):

```
/ {
        sram@2003FC00 {
                compatible = "zephyr,memory-region", "mmio-sram";
                reg = <0x2003FC00 DT_SIZE_K(1)>;
                zephyr,memory-region = "RetainedMem";
                status = "okay";

                retainedmem {
                        compatible = "zephyr,retained-ram";
                        status = "okay";
```

(continues on next page)

(continued from previous page)

```
                        #address-cells = <1>;
                        #size-cells = <1>;

                        boot_info0: boot_info@0 {
                                compatible = "zephyr,retention";
                                status = "okay";
                                reg = <0x0 0x100>;
                        };
                };
        };

        chosen {
                zephyr,bootloader-info = &boot_info0;
        };
};


/* Reduce SRAM0 usage by 1KB to account for non-init area */
&sram0 {
        reg = <0x20000000 DT_SIZE_K(255)>;
};
```

Note that this configuration needs to be applied on both the bootloader (MCUboot) and application to be usable. It can be combined with other retention system APIs such as the *Boot mode*

MCUboot setup Once the above devicetree configuration is applied, MCUboot needs to be configured to store the shared data in this area, the following Kconfigs need to be set for this:

- `CONFIG_RETAINED_MEM` - Enables retained memory driver
- `CONFIG_RETENTION` - Enables retention system
- `CONFIG_BOOT_SHARE_DATA` - Enables shared data
- `CONFIG_BOOT_SHARE_DATA_BOOTINFO` - Enables boot information shared data type
- `CONFIG_BOOT_SHARE_BACKEND_RETENTION` - Stores shared data using retention/blinfo subsystem

Application setup The application must enable the following base Kconfig options for the bootloader information subsystem to function:

- `CONFIG_RETAINED_MEM`
- `CONFIG_RETENTION`
- `CONFIG_RETENTION_BOOTLOADER_INFO`
- `CONFIG_RETENTION_BOOTLOADER_INFO_TYPE_MCUBOOT`

The following include is needed to use the bootloader information subsystem:

```
#include <zephyr/retention/blinfo.h>
```

By default, only the lookup function is provided: `blinfo_lookup()`, the application can call this to query the information from the bootloader. This function is enabled by default with `CONFIG_RETENTION_BOOTLOADER_INFO_OUTPUT_FUNCTION`, however, applications can optionally choose to use the settings storage feature instead. In this mode, the bootloader information can be queries by using settings keys, the following Kconfig options need to be enabled for this mode:

- `CONFIG_SETTINGS`

- CONFIG_SETTINGS_RUNTIME
- CONFIG_RETENTION_BOOTLOADER_INFO_OUTPUT_SETTINGS

This allows the information to be queried via the settings_runtime_get() function with the following keys:

- blinfo/mode The mode that MCUboot is configured for (enum mcuboot_mode value)
- blinfo/signature_type The signature type MCUboot is configured for (enum mcuboot_signature_type value)
- blinfo/recovery The recovery type enabled in MCUboot (enum mcuboot_recovery_mode value)
- blinfo/running_slot The running slot, useful for direct-XIP mode to know which slot to use for an update
- blinfo/bootloader_version Version of the bootloader (struct image_version object)
- blinfo/max_application_size Maximum size of an application (in bytes) that can be loaded

In addition to the previous include, the following includes are required for this mode:

```
#include <bootutil/boot_status.h>
#include <bootutil/image.h>
#include <zephyr/mcuboot_version.h>
#include <zephyr/settings/settings.h>
```

API Reference

Bootloader information API
Bootloader info interface

API Reference

Retention system API
Retention API

Boot mode interface
Boot mode interface

4.5.8 Secure Storage

The secure storage subsystem provides an implementation of the functions defined in the Platform Security Architecture (PSA) Secure Storage API.
It can be enabled on *board targets* that don't already have an implementation of the API.

Overview

The secure storage subsystem makes the PSA Secure Storage API available on all board targets with non-volatile memory support. As such, it provides an implementation of the API on those that don't already have one, ensuring functional support for the API. Board targets with *Trusted Firmware-M (TF-M)* enabled (ending in /ns), for instance, cannot enable the subsystem because TF-M already provides an implementation of the API.

In addition to providing functional support for the API, depending on device-specific security features and the configuration, the subsystem may secure the data stored via the PSA Secure Storage API at rest.
Keep in mind, however, that it's preferable to use a secure processing environment like TF-M when possible because it's able to provide more security due to isolation guarantees.

Limitations

The secure storage subsystem's implementation of the PSA Secure Storage API:

- does not aim at full compliance with the specification.

  Its foremost goal is functional support for the API on all board targets.
  See below for important ways the implementation deviates from the specification.

- does not guarantee that the data it stores will be secure at rest in all cases.

  This depends on device-specific security features and the configuration.

- does not yet provide an implementation of the Protected Storage (PS) API as of this writing.

  Instead, the PS API directly calls into the Internal Trusted Storage (ITS) API (unless a *custom implementation* of the PS API is provided).

Below are some ways the implementation purposefully deviates from the specification and an explanation why. This is not an exhaustive list.

- The UID type is only 30 bits by default. (Against 2.5 UIDs.)

  This is an optimization done to make it more convenient to directly use the UIDs as storage entry IDs (e.g., with *ZMS* when `CONFIG_SECURE_STORAGE_ITS_STORE_IMPLEMENTATION_ZMS` is enabled).
  Zephyr defines numerical ranges to be used by different users of the API which guarantees that there are no collisions and that they all fit within 30 bits. See the header files in include/zephyr/psa for more information.

- The data stored in the ITS is by default encrypted and authenticated (Against `1.` in 3.2. Internal Trusted Storage requirements.)

  The specification considers the storage underlying the ITS to be `implicitly confidential and protected from replay` (2.4. The Internal Trusted Storage API) because `most embedded microprocessors (MCU) have on-chip flash storage that can be made inaccessible except to software running on the MCU` (2.2. Technical Background).
  This is not the case on all MCUs. Thus, additional protection is provided to the stored data.

  However, this does not guarantee that the data stored will be secure at rest in all cases, because this depends on device-specific security features and the configuration. It requires a random entropy source and especially a secure encryption key provider (`CONFIG_SECURE_STORAGE_ITS_TRANSFORM_AEAD_KEY_PROVIDER`).

  In addition, the data stored in the ITS is not protected against replay attacks, because this requires storage that is protected by hardware.

- The data stored via the PSA Secure Storage API is not protected from direct read/write by software or debugging. (Against `2.` and `10.` in 3.2. Internal Trusted Storage requirements.)

  It is only secured at rest. Protecting it at runtime as well requires specific hardware mechanisms to support this.

- The `PSA_STORAGE_FLAG_WRITE_ONCE` flag only protects an entry against modification through the API, not against modification of the storage medium itself.

  Upholding the flag requires knowing that an entry was created in the first place, which is state that has to survive the storage medium being rewritten. Like replay protection, that requires storage that is protected by hardware.
  An attacker who can write to the storage medium can therefore have a write-once entry overwritten or removed: either by tampering with it, after which the subsystem treats it as corrupted and allows it to be replaced, or simply by erasing it, after which it appears never to have existed. Neither can be prevented by encrypting and authenticating the entry.

- The `psa_its_get*()` functions can return `PSA_ERROR_INVALID_SIGNATURE` and `PSA_ERROR_DATA_CORRUPT`.

  The specification doesn't define these for the ITS API because it assumes that the storage underlying it is protected by hardware, and thus that data read back from it is always intact. As it's not the case here, these error codes are passed on to let callers tell an entry that has been tampered with apart from an internal failure.

Configuration

To configure the implementation of the PSA Secure Storage API provided by Zephyr, have a look at the available Kconfig options. They are defined in the various Kconfig files found under subsys/secure_storage/.

Customization

Custom implementations can also replace those of Zephyr at different levels if the functionality provided by the existing implementations isn't enough.

Whole API If you already have an implementation of the whole ITS or PS API and want to make use of it, you can do so by enabling the following Kconfig option and implementing the relevant functions:

- `CONFIG_SECURE_STORAGE_ITS_IMPLEMENTATION_CUSTOM`, for the ITS API.
- `CONFIG_SECURE_STORAGE_PS_IMPLEMENTATION_CUSTOM`, for the PS API.

ITS API Zephyr's implementation of the ITS API (`CONFIG_SECURE_STORAGE_ITS_IMPLEMENTATION_ZEPHYR`) makes use of the ITS transform and store modules, which can be configured and customized separately. Have a look at the ITS transform and store Kconfig options to see the different configuration possibilities.

It's especially recommended to use or implement a secure encryption key provider.

Samples

- persistent_key
- psa_its

PSA Secure Storage API reference

PSA Secure Storage API

Related code samples

- PSA Internal Trusted Storage APIUse the PSA ITS API.

4.5.9 Settings

The settings subsystem gives modules a way to store persistent per-device configuration and runtime state. A variety of storage implementations are provided behind a common API using FCB, NVS, ZMS or a file system. These different implementations give the application developer flexibility to select an appropriate storage medium, and even change it later as needs change. This subsystem is used by various Zephyr components and can be used simultaneously by user applications.

Settings items are stored as key-value pair strings. By convention, the keys can be organized by the package and subtree defining the key, for example the key `id/serial` would define the `serial` configuration element for the package `id`.

Convenience routines are provided for converting a key value to and from a string type.

For an example of the settings subsystem refer to settings sample.

Note

As of Zephyr release 4.1 the recommended backends for non-filesystem storage are *NVS* and *ZMS*.

Handlers

Settings handlers for subtree implement a set of handler functions. These are registered using a call to `settings_register()` for dynamic handlers or defined using a call to `SETTINGS_STATIC_HANDLER_DEFINE()` for static handlers.

h_get
: This gets called when asking for a settings element value by its name using `settings_runtime_get()` from the runtime backend.

h_set
: This gets called when the value is loaded from persistent storage with `settings_load()`, or when using `settings_runtime_set()` from the runtime backend.

h_commit
: This gets called after the settings have been loaded in full. Sometimes you don't want an individual setting value to take effect right away, for example if there are multiple settings which are interdependent.

h_export
: This gets called to write all current settings. This happens when `settings_save()` tries to save the settings or transfer to any user-implemented back-end.

Settings handlers also have a commit priority `cprio` that allows to prioritize the `h_commit` calls. This can be advantageous when e.g. a subsystem initializes a service that other `h_commit` calls depend on.

Settings handlers `h_commit` routines are by default initialized with `cprio = 0`, initializing a settings handler with a different priority is done using a call to `settings_register_with_cprio()` for dynamic handlers or using a call to `SETTINGS_STATIC_HANDLER_DEFINE_WITH_CPRIO()` for

static handlers. The specified `cprio` value is an integer where lower values mean higher priority.

Backends

Backends are meant to load and save data to/from setting handlers, and implement a set of handler functions. These are registered using a call to `settings_src_register()` for backends that can load data, and/or `settings_dst_register()` for backends that can save data. The current implementation allows for multiple source backends but only a single destination backend.

csi_load
This gets called when loading values from persistent storage using `settings_load()`.

csi_load_one
This gets called when loading only one item from persistent storage using `settings_load_one()`.

csi_get_val_len
This gets called when getting a value's length from persistent storage using `settings_get_val_len()`.

csi_save
This gets called when saving a single setting to persistent storage using `settings_save_one()`.

csi_save_start
This gets called when starting a save of all current settings using `settings_save()` or `settings_save_subtree()`.

csi_save_end
This gets called after having saved of all current settings using `settings_save()` or `settings_save_subtree()`.

Zephyr Storage Backends

Zephyr offers the following storage backends:

- Flash Circular Buffer (`CONFIG_SETTINGS_FCB`).
- A file in the filesystem (`CONFIG_SETTINGS_FILE`).
- Non-Volatile Storage (`CONFIG_SETTINGS_NVS`).
- Zephyr Memory Storage (`CONFIG_SETTINGS_ZMS`).

You can declare multiple sources for settings; settings from all of these are restored when `settings_load()` is called.

There can be only one target for writing settings; this is where data is stored when you call `settings_save()`, or `settings_save_one()`.

FCB read target is registered using `settings_fcb_src()`, and write target using `settings_fcb_dst()`. As a side-effect, `settings_fcb_src()` initializes the FCB area, so it must be called before calling `settings_fcb_dst()`. File read target is registered using `settings_file_src()`, and write target by using `settings_file_dst()`.

Non-volatile storage read target is registered using `settings_nvs_src()`, and write target by using `settings_nvs_dst()`.

Zephyr Memory Storage (ZMS) read target is registered using `settings_zms_src()`, and write target is registered using `settings_zms_dst()`.

ZMS backend has the particularity of using hash functions to hash the settings key before storing it to the persistent storage. This implementation implies that some collisions between key's hashes could occur if a big number of different keys are stored. This number depends on the selected hash function.

ZMS backend can handle 2^n maximum collisions where n is defined by (CONFIG_SETTINGS_ZMS_MAX_COLLISIONS_BITS).

Storage Location

The FCB, non-volatile storage (NVS) and ZMS backends look for a fixed partition with label "storage" by default. A different partition can be selected by setting the zephyr,settings-partition property of the chosen node in the devicetree.

The file path used by the file backend to store settings is selected via the option CONFIG_SETTINGS_FILE_PATH.

Loading data from persistent storage

A call to settings_load() uses an h_set implementation to load settings data from storage to volatile memory. After all data is loaded, the h_commit handler is issued, signalling the application that the settings were successfully retrieved.

Alternatively, a call to settings_load_one() will load only one Settings entry and store it in the provided buffer.

Optionally, to get only the value's length associated with the Settings entry, a call to settings_get_val_len() can be performed. This is used for example by applications that allocates dynamically the data buffer and needs to get the data size before reading it by settings_load_one().

Technically FCB and file backends may store some history of the entities. This means that the newest data entity is stored after any older existing data entities. Starting with Zephyr 2.1, the back-end must filter out all old entities and call the callback with only the newest entity.

Storing data to persistent storage

A call to settings_save_one() uses a backend implementation to store settings data to the storage medium. A call to settings_save() uses an h_export implementation to store different data in one operation using settings_save_one(). A key needs to be covered by a h_export only if it is supposed to be stored by settings_save() call.

For both FCB and file back-end only storage requests with data which changes most actual key's value are stored, therefore there is no need to check whether a value changed by the application. Such a storage mechanism implies that storage can contain multiple value assignments for a key , while only the last is the current value for the key.

Garbage collection When storage becomes full (FCB) or consumes too much space (file), the backend removes non-recent key-value pairs records and unnecessary key-delete records.

Secure domain settings

Currently settings doesn't provide scheme of being secure, and non-secure configuration storage simultaneously for the same instance. It is recommended that secure domain uses its own settings instance and it might provide data for non-secure domain using dedicated interface if needed (case dependent).

Example: Device Configuration

This is a simple example, where the settings handler only implements h_set and h_export. h_set is called when the value is restored from storage (or when set initially), and h_export is used to write the value to storage thanks to storage_func(). The user can also implement some other export functionality, for example, writing to the shell console).

```
#define DEFAULT_FOO_VAL_VALUE 1

static int8 foo_val = DEFAULT_FOO_VAL_VALUE;

static int foo_settings_set(const char *name, size_t len,
                            settings_read_cb read_cb, void *cb_arg)
{
    const char *next;
    int rc;

    if (settings_name_steq(name, "bar", &next) && !next) {
        if (len != sizeof(foo_val)) {
            return -EINVAL;
        }

        rc = read_cb(cb_arg, &foo_val, sizeof(foo_val));
        if (rc >= 0) {
            /* key-value pair was properly read.
             * rc contains value length.
             */
            return 0;
        }
        /* read-out error */
        return rc;
    }

    return -ENOENT;
}

static int foo_settings_export(int (*storage_func)(const char *name,
                                                   const void *value,
                                                   size_t val_len))
{
    return storage_func("foo/bar", &foo_val, sizeof(foo_val));
}

struct settings_handler my_conf = {
    .name = "foo",
    .h_set = foo_settings_set,
    .h_export = foo_settings_export
};
```

Example: Persist Runtime State

This is a simple example showing how to persist runtime state. In this example, only `h_set` is defined, which is used when restoring value from persistent storage.

In this example, the `main` function increments `foo_val`, and then persists the latest number. When the system restarts, the application calls `settings_load()` while initializing, and `foo_val` will continue counting up from where it was before restart.

```
#include <zephyr/kernel.h>
#include <zephyr/sys/reboot.h>
#include <zephyr/settings/settings.h>
#include <zephyr/sys/printk.h>
#include <inttypes.h>
```

(continues on next page)

(continued from previous page)

```
#define DEFAULT_FOO_VAL_VALUE 0

static uint8_t foo_val = DEFAULT_FOO_VAL_VALUE;

static int foo_settings_set(const char *name, size_t len,
                            settings_read_cb read_cb, void *cb_arg)
{
    const char *next;
    int rc;

    if (settings_name_steq(name, "bar", &next) && !next) {
        if (len != sizeof(foo_val)) {
            return -EINVAL;
        }

        rc = read_cb(cb_arg, &foo_val, sizeof(foo_val));
        if (rc >= 0) {
            return 0;
        }

        return rc;
    }


    return -ENOENT;
}

struct settings_handler my_conf = {
    .name = "foo",
    .h_set = foo_settings_set
};

int main(void)
{
    settings_subsys_init();
    settings_register(&my_conf);
    settings_load();

    foo_val++;
    settings_save_one("foo/bar", &foo_val, sizeof(foo_val));

    printk("foo: %d\n", foo_val);

    k_msleep(1000);
    sys_reboot(SYS_REBOOT_COLD);
}
```

Example: Custom Backend Implementation

This is a simple example showing how to register a simple custom backend handler (`CONFIG_SETTINGS_CUSTOM`).

```
static int settings_custom_load(struct settings_store *cs,
                                const struct settings_load_arg *arg)
{
    //...
```

(continues on next page)

(continued from previous page)

```
}

static int settings_custom_save(struct settings_store *cs, const char *name,
                                const char *value, size_t val_len)
{
    //...
}

/* custom backend interface */
static struct settings_store_itf settings_custom_itf = {
    .csi_load = settings_custom_load,
    .csi_save = settings_custom_save,
};

/* custom backend node */
static struct settings_store settings_custom_store = {
    .cs_itf = &settings_custom_itf
};

int settings_backend_init(void)
{
    /* register custom backend */
    settings_dst_register(&settings_custom_store);
    settings_src_register(&settings_custom_store);
    return 0;
}
```

API Reference

The Settings subsystem APIs are provided by include/zephyr/settings/settings.h.

API for general settings usage

Settings

Related code samples

- Settings APILoad and save configuration values using the settings API.

API for key-name processing

Settings name processing

Related code samples

- Settings APILoad and save configuration values using the settings API.

API for runtime settings manipulation

Settings subsystem runtime

Related code samples

- Settings APILoad and save configuration values using the settings API.

API of backend interface

Settings backend interface

4.5.10 Stream Flash

The Stream Flash module takes contiguous fragments of a stream of data (e.g. from radio packets), aggregates them into a user-provided buffer, then when the buffer fills (or stream ends) writes it to a raw flash partition. It supports providing the read-back buffer to the client to use in validating the persisted stream content.

One typical use of a stream write operation is when receiving a new firmware image to be used in a DFU operation.

There are several reasons why one might want to use buffered writes instead of writing the data directly as it is made available. Some devices have hardware limitations which does not allow flash writes to be performed in parallel with other operations, such as radio RX and TX. Also, fewer write operations result in faster response times seen from the application.

Persistent stream write progress

Some stream write operations, such as DFU operations, may run for a long time. When performing such long running operations it can be useful to be able to save the stream write progress to persistent storage so that the operation can resume at the same point after an unexpected interruption.

The Stream Flash module offers an API for loading, saving and clearing stream write progress to persistent storage using the *Settings* module. The API can be enabled using `CONFIG_STREAM_FLASH_PROGRESS`.

API Reference

Stream to flash interface

4.5.11 Zephyr Memory Storage (ZMS)

Zephyr Memory Storage is a new key-value storage system that is designed to work with all types of non-volatile storage technologies. It supports classical on-chip NOR flash as well as new technologies like RRAM and MRAM that do not require a separate erase operation at all, that is, data on these types of devices can be overwritten directly at any time.

General behavior

ZMS divides the memory space into sectors (minimum 2), and each sector is filled with key-value pairs until it is full.

The key-value pair is divided into two parts:

- The key part is written in an ATE (Allocation Table Entry) called “ID-ATE” which is stored starting from the bottom of the sector.
- The value part is defined as “data” and is stored raw starting from the top of the sector.

Additionally, for each sector we store at the last positions header ATEs which are ATEs that are needed for the sector to describe its status (closed, open) and the current version of ZMS.

When the current sector is full we verify first that the following sector is empty, we garbage collect the sector N+2 (where N is the current sector number) by moving the valid ATEs to the N+1 empty sector, we erase the garbage-collected sector and then we close the current sector by writing a garbage_collect_done ATE and the close ATE (one of the header entries). Afterwards we move forward to the next sector and start writing entries again.

This behavior is repeated until it reaches the end of the partition. Then it starts again from the first sector after garbage collecting it and erasing its content.

Composition of a sector A sector is organized in this form (example with 3 sectors):

Sector 0 (closed)	Sector 1 (open)	Sector 2 (empty)
Data_a0	Data_b0	Data_c0
Data_a1	Data_b1	Data_c1
Data_a2	Data_b2	Data_c2
GC_done	.	.
.	.	.
.	.	.
.	ID ATE_b2	ID ATE_c2
ID ATE_a2	ID ATE_b1	ID ATE_c1
ID ATE_a1	ID ATE_b0	ID ATE_c0
ID ATE_a0	GC_done ATE	GC_done ATE
Close ATE (cyc=1)	Close ATE (cyc=1)	Close ATE (cyc=1)
Empty ATE (cyc=1)	Empty ATE (cyc=2)	Empty ATE (cyc=2)

Definition of each element in the sector `Empty ATE` is written when erasing a sector (last position of the sector).

`Close ATE` is written when closing a sector (second to last position of the sector).

`GC_done ATE` is written to indicate that the next sector has already been garbage-collected. This ATE could be at any position of the sector.

`ID ATE` contains a key of type `zms_id_t` and describes where the data is stored, its size and its CRC32.

`Data` is the actual value associated to the ID-ATE.

How does ZMS work?

Mounting the storage system Mounting the storage system starts by getting the flash parameters, checking that the file system properties are correct (sector_size, sector_count ...) then calling the zms_init function to make the storage ready.

By default, `zms_mount()` returns an error if the partition cannot be mounted. For recovery-oriented use cases, `zms_mount_force()` can be used to automatically wipe and reinitialize the partition when the first mount attempt fails.

Mount flags ZMS mount behavior can be controlled with optional flags in `zms_fs.mount_flags`.

- Default behavior (no optional flags set, `mount_flags = 0`): if the partition is erased and no valid ZMS header is found, `zms_mount()` formats the partition by creating the initial ZMS header.
- `ZMS_MOUNT_FLAG_NO_FORMAT`: if the partition is erased and no valid ZMS header is found, `zms_mount()` does not format the partition and returns `-ENOTSUP`.

To mount the filesystem the following elements in the `zms_fs` structure must be initialized:

```
struct zms_fs {
        /** File system offset in flash **/
        off_t offset;

        /** Storage system is split into sectors, each sector size must be␣
↪multiple of
```

(continues on next page)

(continued from previous page)

```
         * erase-blocks if the device has erase capabilities
         */
        uint32_t sector_size;
        /** Number of sectors in the file system */
        uint32_t sector_count;

        /** Optional mount behavior flags (enum zms_mount_flags) */
        uint32_t mount_flags;

        /** Flash device runtime structure */
        const struct device *flash_device;
};
```

Initialization As ZMS has a fast-forward write mechanism, it must find the last sector and the last pointer of the entry where it stopped the last time. It must look for a closed sector followed by an open one, then within the open sector, it finds (recovers) the last written ATE. After that, it checks that the sector after this one is empty, or it will erase it.

ZMS ID/data write To avoid rewriting the same data with the same ID again, ZMS must look in all the sectors if the same ID exists and then compares its data. If the data is identical, no write is performed. If it must perform a write, then an ATE and the data (if the operation is not a delete) are written in the sector. If the sector is full (cannot hold the current data + ATE), ZMS has to move to the next sector, garbage collect the sector after the newly opened one then erase it.

ZMS ID/data read (with history) By default ZMS looks for the last data with the same ID by browsing through all stored ATEs from the most recent ones to the oldest ones. If it finds a valid ATE with a matching ID it retrieves its data and returns the number of bytes that were read. If a history count is provided and different than 0, older data with same ID is retrieved.

ZMS entry enumeration (iterator API) Applications can enumerate stored entries without supplying a predefined list of IDs. This is useful when IDs are derived from payload data (for example packed addresses) and the application does not keep a separate index in flash.

This API family is generally used at boot, when the application rebuilds its in-memory state from the ID/length metadata persisted in ZMS.

The iterator API consists of:

- `zms_iter_init()` to initialize iterator state.
- `zms_iter_init_with_config()` to initialize iterator state with optional ID mask and inclusive ID range and predicate filters.
- `zms_iter_next()` to retrieve one live entry (unique ID) at a time.
- `zms_iter_next_all()` to retrieve all matching entries (full history, including delete markers).

Choosing between `zms_iter_next()` and `zms_iter_next_all()`

- `zms_iter_next()` returns unique live `(id, len)` pairs:
  - Sector/header entries are skipped.
  - Delete markers (entries with `len == 0`) are skipped.
  - If the same ID appears multiple times in storage history, only the newest valid occurrence is returned.

  - To guarantee uniqueness, ZMS checks for newer entries of each candidate ID, which can require additional scanning.
- `zms_iter_next_all()` is faster and returns all matching `(id, len)` pairs as encountered:
  - Historical entries for the same ID are returned.
  - Delete markers (entries with `len == 0`) are returned.
  - No uniqueness filtering is performed.
  - The application is responsible for filtering/compacting the returned pairs according to its own policy.

  Use `zms_iter_init_with_config()` when you want to enumerate only IDs matching a bitmask, a specific inclusive ID range, a predicate callback, or any combination of those filters.

Iterator return values Calling either `zms_iter_next()` or `zms_iter_next_all()` returns:

- `1` when an entry is found; output arguments contain ID and data length.
- `0` when there are no more entries to enumerate.
- Negative errno on error (for example `-EINVAL`, `-EIO`, `-ENXIO`).

The iterator always reports metadata (ID and length). In addition, when the caller provides a non-NULL `data` buffer, the iterator also copies the entry's content directly into it, but only for entries small enough to be stored inside the ATE. For larger entries, or when no buffer is provided, the data must be read explicitly using `zms_read()` for the latest value, or `zms_read_hist()` for older revisions.

Concurrency and snapshot behavior The iterator captures a traversal boundary when `zms_iter_init()` is called. Entries written after initialization are not part of that traversal.

Do not call `zms_write()` (including delete operations) on the same `zms_fs` between two iterator-step calls (`zms_iter_next()` or `zms_iter_next_all()`). If a write triggers garbage collection while iterating, iterator behavior is undefined. This rule applies to the thread doing the iteration regardless of locking: taking the lock described below only keeps *other* threads from writing during the traversal.

Locking out concurrent writers `zms_fs` exposes its internal write lock as a plain `struct k_mutex zms_lock` field. `zms_write()` and `zms_delete()` take this mutex internally for the duration of the write. An application with multiple threads sharing the same `zms_fs` can hold this same mutex across an entire iteration to block writes from *other* threads for as long as the traversal boundary must stay valid:

```
k_mutex_lock(&fs.zms_lock, K_FOREVER);

rc = zms_iter_init(&fs, &iter);
/* ... */

while ((rc = zms_iter_next(&fs, &iter, &id, &len, NULL, 0)) == 1) {
  /* ... */
}

k_mutex_unlock(&fs.zms_lock);
```

Usage example

```
struct zms_iter iter;
bool predicate_id_not_2(zms_id_t id)
```

(continues on next page)

(continued from previous page)

```
{
  return id != (zms_id_t)2;
}

struct zms_iter_config iter_config = {
  .mask_id = (zms_id_t)0x03,
  .min_id = (zms_id_t)0x01,
  .max_id = (zms_id_t)0x03,
  .use_mask = true,
  .use_range = true,
  .use_predicate = true,
  .predicate_func = predicate_id_not_2,
};
zms_id_t id;
size_t len;
uint8_t data[ZMS_DATA_IN_ATE_SIZE];
int rc;

/* Block writes from other threads for the duration of the traversal. */
k_mutex_lock(&fs.zms_lock, K_FOREVER);

rc = zms_iter_init_with_config(&fs, &iter, &iter_config);
if (rc) {
  /* handle error */
}

while ((rc = zms_iter_next(&fs, &iter, &id, &len, data, sizeof(data))) == 1) {
  /* id: entry ID matching configured mask/range/predicate, len: current stored␣
↪size */
  /* data: filled with the entry's content when it is small enough to be
   * stored directly inside the ATE; otherwise left untouched, in which case
   * zms_read() must be used to fetch it.
   */
  printk("id=0x%llx len=%zu\n", (unsigned long long)id, len);
}

k_mutex_unlock(&fs.zms_lock);

if (rc < 0) {
  /* handle iterator error */
}
```

The optional `data`/`data_len` arguments can be set to `NULL`/`0` when only the `(id, len)` pairs are needed. When a buffer is provided, only the data that ZMS stores directly inside the ATE is copied; larger entries must still be retrieved with `zms_read()`.

Zero-initialize `zms_iter_config` or leave `use_mask`, `use_range`, and `use_predicate` disabled to keep the default behavior: accept all IDs and the full `zms_id_t` range.

To determine how many revisions exist for an ID returned by the iterator, call `zms_read_hist()` with increasing `cnt` until `-ENOENT` is returned.

ZMS free space calculation ZMS can also return the free space remaining in the partition. However, this operation is very time-consuming as it needs to browse through all valid ATEs in all sectors of the partition and for each valid ATE try to find if an older one exists. It is not recommended for applications to use this function often, as it is time-consuming and could slow down the calling thread.

The cycle counter Each sector has a lead cycle counter which is a `uint8_t` that is used to validate all the other ATEs. The lead cycle counter is stored in the empty ATE. To become valid, an ATE must have the same cycle counter as the one stored in the empty ATE. Each time an ATE is moved from a sector to another it must get the cycle counter of the destination sector. To erase a sector, the cycle counter of the empty ATE is incremented and a single write of the empty ATE is done. All the ATEs in that sector become invalid.

Closing sectors To close a sector a close ATE is added at the end of the sector and it must have the same cycle counter as the empty ATE. When closing a sector, all the remaining space that has not been used is filled with garbage data to avoid having old ATEs with a valid cycle counter.

Triggering garbage collection Some applications need to make sure that storage writes have a maximum defined latency. When calling ZMS to make a write, the current sector could be almost full such that ZMS needs to trigger the GC to switch to the next sector. This operation is time-consuming and will cause some applications to not meet their real time constraints. ZMS adds an API for the application to get the current remaining free space in a sector. The application could then decide when to switch to the next sector if the current one is almost full. This will of course trigger the garbage collection operation on the next sector. This will guarantee the application that the next write won't trigger the garbage collection.

ATE (Allocation Table Entry) structure An entry uses 16 bytes to encode its information. The exact structure is determined by ATE format which can be selected for a given application.

ZMS defines multiple ATE formats tailored for different feature sets. At runtime, it recognizes the format using the metadata field in empty ATEs, which has the same byte position in all formats.

Table 53: Entry format for 32-bit IDs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
crc8	cycle_cnt	len		id				data (if len <= 8)							
								offset				data_crc			
												metadata			

This is the default format which is captured in the API documentation for `zms_ate`. The `data_crc` is optionally included to integrity-check data stored at the top of the sector.

Note

The CRC of the data is checked only when a full read of the data is made. The CRC of the data is not checked for a partial read, as it is computed for the whole element.

Warning

Enabling the CRC feature on previously existing ZMS content that did not have it enabled will make all existing data invalid.

Table 54: Entry format for 64-bit IDs

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
crc8	cycle_cnt	len		id								data (if len <= 4)			
												offset			
												metadata			

This format is selected when `CONFIG_ZMS_ID_64BIT` is enabled.

Warning

Selecting a different ATE format than the one used by previously existing ZMS content will make all existing data invalid.

Note

The ZMS backend for *Settings* does not support this format.

Available space for user data (key-value pairs)

ZMS always needs an empty sector to be able to perform the garbage collection (GC). So, if we suppose that 4 sectors exist in a partition, ZMS will only use 3 sectors to store key-value pairs and keep one sector empty to be able to perform GC. The empty sector will rotate between the 4 sectors in the partition.

Note

The maximum single data length that can be written at once in a sector is 64K (this could change in future versions of ZMS).

Small data values Values which are sufficiently small will be stored within the entry (ATE) itself, without writing data at the top of the sector. The amount of data that can fit inside the entry depends on its selected format. See the *ATE structure* section.

ZMS has an entry size of 16 bytes which means that the maximum available space in a partition to store data is computed in this scenario as:

$$\frac{(NUM_SECTORS - 1) \times (SECTOR_SIZE - (5 \times ATE_SIZE)) \times (DATA_SIZE)}{ATE_SIZE}$$

Where:

`NUM_SECTOR`: Total number of sectors

`SECTOR_SIZE`: Size of the sector

`ATE_SIZE`: 16 bytes

`(5 * ATE_SIZE)`: Reserved ATEs for header and delete items

`DATA_SIZE`: 8 bytes or 4 bytes depending on the ATE format

For example for 4 sectors of 1024 bytes, with the default ATE format, free space for 8-byte length data is $\frac{3 \times 944 \times 8}{16} = 1416$ bytes.

Large data values Values exceeding `DATA_SIZE` are stored outside of the ATE at the top of the sector. In this case, it is hard to estimate the free available space, as this depends on the size of the data. But we can take into account that for N bytes of data added at the top of the sector, an additional 16 bytes of ATE must be added at the bottom of the sector, which adds up to $N + 16$ bytes for the key-value pair.

Let's take an example:

For a partition that has 4 sectors of 1024 bytes and for data size of 64 bytes. Only 3 sectors are available for writes with a capacity of 944 bytes each, which makes it possible to store 11 key-value pairs in each sector ($\frac{944}{64+16}$). Total data that could be stored in this partition for this case is $11 \times 3 \times 64 = 2112$ bytes.

Available space calculator

Wear leveling

This storage system is optimized for devices that do not require an erase. Storage systems that rely on an erase value (NVS as an example) need to emulate the erase with write operations. This causes a significant decrease in the life expectancy of these devices as well as more delays for write operations and initialization of the device when it is empty. ZMS uses a cycle count mechanism that avoids emulating erase operations for these devices. It also guarantees that every memory location is written only once for each cycle of sector write.

As an example, to erase a 4096-byte sector on devices that do not require an erase operation using NVS, 256 flash writes must be performed (supposing that `write-block-size` = 16 bytes), while using ZMS, only 1 write of 16 bytes is needed. This operation is 256 times faster in this case.

The garbage collection operation also reduces the memory cell life expectancy as it performs write operations when moving blocks from one sector to another. To make the garbage collector not affect the life expectancy of the device it is recommended to dimension the partition appropriately. Its size should be the double of the maximum size of data (including headers) that could be written in the storage.

See *Available space for user data*.

Device lifetime calculation Storage devices, whether they are classical flash or new technologies like RRAM/MRAM, have a limited life expectancy which is determined by the number of times memory cells can be erased/written. Flash devices are erased one page at a time as part of their functional behavior (otherwise memory cells cannot be overwritten), and for storage devices that do not require an erase operation, memory cells can be overwritten directly.

A typical scenario is shown here to calculate the life expectancy of a device: Let's suppose that we store a 4-byte variable using the same ID but its content changes every minute. The partition has 4 sectors with 1024 bytes each. Each write of the variable requires 16 bytes of storage. As we have 944 bytes available for ATEs for each sector, and because ZMS is a fast-forward storage system, we are going to rewrite the first location of the first sector after $\frac{(944 \times 4)}{16} = 236$ minutes.

In addition to the normal writes, the garbage collector will move the data that is still valid from old sectors to new ones. As we are using the same ID and a big partition size, no data will be moved by the garbage collector in this case. For storage devices that can be written 20 000 times, the storage will last about 4 720 000 minutes (~9 years).

To make a more general formula we must first compute the effective used size in ZMS by our typical set of data. For ID/data pairs with *small data*, `effective_size` is `16` bytes, while for *large data*, `effective_size` is `16 + sizeof(data)` bytes. Let's suppose that `total_effective_size` is the total size of the data that is written in the storage and that the partition is sized appropriately (double of the effective size) to avoid having the garbage collector moving blocks all the time.

The expected lifetime of the device in minutes is computed as:

$$\frac{(SECTOR_EFFECTIVE_SIZE \times SECTOR_NUMBER \times MAX_NUM_WRITES)}{(TOTAL_EFFECTIVE_SIZE \times WR_MIN)}$$

Where:

`SECTOR_EFFECTIVE_SIZE`: The sector size - header size (80 bytes)

`SECTOR_NUMBER`: The number of sectors

`MAX_NUM_WRITES`: The life expectancy of the storage device in number of writes

`TOTAL_EFFECTIVE_SIZE`: Total effective size of the set of written data

`WR_MIN`: Number of writes of the set of data per minute

Device lifetime calculator

Features

ZMS has introduced many features compared to existing storage system like NVS and will evolve from its initial version to include more features that satisfies new technologies requirements such as low latency and bigger storage space.

Existing features

Version 1

- Supports storage devices that do not require an erase operation (only one write operation to invalidate a sector)
- Supports large partition and sector sizes (64-bit address space)
- Supports 32-bit IDs and 64-bit IDs
- Small data values are stored in the ATE itself
- Built-in data CRC32 (included in the ATE)
- Versioning of ZMS (to handle future evolutions)
- Supports large `write-block-size` (only for platforms that need it)
- Supports multiple ATE formats to satisfy the requirements of different applications
- Supports forced mount recovery via `zms_mount_force()` (automatic partition wipe and reinitialization on mount failure)
- Supports a lookup cache to speed up repeated ID lookups with optional per-instance cache configuration to customize the cache size on a per-partition basis
- Supports live-entry enumeration without predeclared ID lists via `zms_iter_init()` and `zms_iter_next()`

Future features

- Add the possibility to mount multiple filesystems with different ATE formats (currently, all filesystems in the same application must use the same format)
- Add the possibility to skip garbage collector for some application usage where ID/value pairs are written periodically and do not exceed half of the partition size (there is always an old entry with the same ID).
- Divide IDs into namespaces and allocate IDs on demand from application to handle collisions between IDs used by different subsystems or samples.
- Add the possibility to retrieve the wear out value of the device based on the cycle count value
- Add a recovery function that can recover a storage partition if something went wrong
- Add a library/application to allow migration from NVS entries to ZMS entries

ZMS and other storage systems in Zephyr This section describes ZMS in the wider context of storage systems in Zephyr (not full filesystems, but simpler, non-hierarchical ones). Today Zephyr includes at least two other systems that are somewhat comparable in scope and functionality: *NVS* and *FCB*. Which one to use in your application will depend on your needs and the hardware you are using, and this section provides information to help make a choice.

- If you are using devices that do not require an erase operation like RRAM or MRAM, *ZMS* is definitely the best fit for your storage subsystem as it is designed to avoid emulating erase operation using large block writes for these devices and replaces it with a single write call.
- For devices that have a large `write_block_size` and/or need a sector size that is different than the classical flash page size (equal to erase_block_size), *ZMS* is also the best fit as there is the possibility to customize these parameters and add the support of these devices in ZMS.
- For classical flash technology devices, *NVS* is recommended as it has low footprint (smaller ATEs and smaller header ATEs). Erasing flash in NVS is also very fast and do not require an additional write operation compared to ZMS. For these devices, NVS reads/writes will be faster as well than ZMS as it has smaller ATE size.
- If your application needs more than 64K IDs for storage, *ZMS* is recommended here because the ID field is up to 64-bit.
- If your application is working in a FIFO mode (First-in First-out) then *FCB* is the best storage solution for this use case.

More generally to make the right choice between NVS and ZMS, all the blockers should be first verified to make sure that the application could work with one subsystem or the other, then if both solutions could be implemented, the best choice should be based on the calculations of the life expectancy of the device described in this section: *Wear leveling*.

Recommendations to increase performance

Sector size and count

- The total size of the storage partition should be set appropriately to achieve the best performance with ZMS. All the information regarding the effectively available free space in ZMS can be found in the documentation. See *Available space for user data*. It's recommended to choose a storage partition size that is double the size of the key-value pairs that will be written in the storage.
- The sector size needs to be set such that a sector can fit the maximum data size that will be stored. Increasing the sector size will slow down the garbage collection operation and make it occur less frequently. Decreasing its size, on the opposite, will make the garbage collection operation faster but also occur more frequently.
- For some subsystems like *Settings*, all path-value pairs are split into two ZMS entries (ATEs). The headers needed by the two entries should be accounted for when computing the needed storage space.
- Using *small data values* can increase the performance, as this data is written within the entry. For example, for the *Settings* subsystem, choosing a path name that is less than or equal to 8 bytes can make reads and writes faster.

Cache size

- When using the ZMS API directly, the recommendation for the cache size is to make it at least equal to the number of different entries that will be written in the storage.
- Each additional cache entry will add 8 bytes to your RAM usage. Cache size should be carefully chosen.

- If you use ZMS through *Settings*, you have to take into account that each Settings entry is divided into two ZMS entries. The recommendation for the cache size is to make it at least twice the number of Settings entries.

Manual per-instance cache When `CONFIG_ZMS_LOOKUP_CACHE_MANUAL` is enabled, each `zms_fs` instance can provide its own lookup cache buffer instead of using the global `CONFIG_ZMS_LOOKUP_CACHE_SIZE` setting.

Call `zms_set_lookup_cache()` before `zms_mount()` to assign the cache buffer for that instance. If no buffer is assigned, the lookup cache remains disabled for that `zms_fs` instance.

```
static struct zms_fs zms_a;
static struct zms_fs zms_b;
static uint64_t zms_a_cache[512];
static uint64_t zms_b_cache[32];

zms_set_lookup_cache(&zms_a, zms_a_cache, ARRAY_SIZE(zms_a_cache));
zms_set_lookup_cache(&zms_b, zms_b_cache, ARRAY_SIZE(zms_b_cache));
```

ID size

- The 64-bit ID space is expected to be larger than necessary for most applications. Unless you have a particular need for this, it's recommended to stick with 32-bit IDs. This is expected to have a slight impact on code size and performance, even on 64-bit systems, because the byte position of IDs in storage is not aligned to an 8-byte boundary.

API Reference

The ZMS API is provided by `zms.h`:

ZMS data structures

ZMS API

Related code samples

- Zephyr Memory Storage (ZMS)Store and retrieve data from storage using the ZMS API.
- ZMS Cycle Count VerificationVerify that ZMS sector cycle counters are tracked correctly across garbage collection cycles, including past the uint8_t wraparound boundary.

Memory Management

4.6 Memory Management

This section covers services for managing memory and data buffers, including dynamic allocation, memory attributes and mappings, and network-style buffer pools.

4.6.1 Memory Attributes

It is possible in the devicetree to mark the memory regions with attributes by using the `zephyr,memory-attr` property. This property and the related memory region can then be retrieved at run-time by leveraging a provided helper library.

The set of general attributes that can be specified in the property are defined and explained in include/zephyr/dt-bindings/memory-attr/memory-attr.h.

For example, to mark a memory region in the devicetree as non-volatile, cacheable, out-of-order:

```
mem: memory@10000000 {
    compatible = "mmio-sram";
    reg = <0x10000000 0x1000>;
    zephyr,memory-attr = <(DT_MEM_NON_VOLATILE | DT_MEM_CACHEABLE | DT_MEM_000)>;
};
```

Note

The `zephyr,memory-attr` usage does not result in any memory region actually created. When it is needed to create an actual section out of the devicetree defined memory region, it is possible to use the compatible `zephyr,memory-region` that will result (only when supported by the architecture) in a new linker section and region.

The `zephyr,memory-attr` property can also be used to set architecture-specific and software-specific custom attributes that can be interpreted at run time. This is leveraged, among other things, to create MPU regions out of devicetree defined memory regions, for example:

```
mem: memory@10000000 {
    compatible = "mmio-sram";
    reg = <0x10000000 0x1000>;
    zephyr,memory-region = "NOCACHE_REGION";
    zephyr,memory-attr = <DT_MEM_ARM_MPU_RAM_NOCACHE>;
};
```

See include/zephyr/dt-bindings/memory-attr/memory-attr-arm.h and *Memory map and MPU considerations* in the *Arm Cortex-M Developer Guide* for more details about MPU usage. Also see *Caching Basics* for details on how Zephyr handles caching.

The conventional and recommended way to deal and manage with memory regions marked with attributes is by using the provided `mem-attr` helper library by enabling `CONFIG_MEM_ATTR`. When this option is enabled the list of memory regions and their attributes are compiled in a user-accessible array and a set of functions is made available that can be used to query, probe and act on regions and attributes (see next section for more details).

Note

The `zephyr,memory-attr` property is only a descriptive property of the capabilities of the associated memory region, but it does not result in any actual setting for the memory to be set. The user, code or subsystem willing to use this information to do some work (for example creating an MPU region out of the property) must use either the provided `mem-attr` library or the usual devicetree helpers to perform the required work / setting. Note, however, that for some architectures (such as ARM and ARM64) the MPU driver uses this information to properly initialize caching at boot. See `CONFIG_ARM_MPU`, `CONFIG_RISCV_PMP`, etc.

A test for the `mem-attr` library and its usage is provided in `tests/subsys/mem_mgmt/mem_attr/`.

Memory Attributes Heap Allocator

It is possible to leverage the memory attribute property `zephyr,memory-attr` to define and create a set of memory heaps from which the user can allocate memory from with certain attributes / capabilities.

When the `CONFIG_MEM_ATTR_HEAP` is set, every region marked with one of the memory attributes listed in include/zephyr/dt-bindings/memory-attr/memory-attr-sw.h is added to a pool of memory heaps used for dynamic allocation of memory buffers with certain attributes.

Here a non exhaustive list of possible attributes:

```
DT_MEM_SW_ALLOC_CACHE
DT_MEM_SW_ALLOC_NON_CACHE
DT_MEM_SW_ALLOC_DMA
```

For example we can define several memory regions with different attributes and use the appropriate attribute to indicate that it is possible to dynamically allocate memory from those regions:

```
mem_cacheable: memory@10000000 {
    compatible = "mmio-sram";
    reg = <0x10000000 0x1000>;
    zephyr,memory-attr = <(DT_MEM_CACHEABLE | DT_MEM_SW_ALLOC_CACHE)>;
};

mem_non_cacheable: memory@20000000 {
    compatible = "mmio-sram";
    reg = <0x20000000 0x1000>;
    zephyr,memory-attr = <(DT_MEM_NON_CACHEABLE | ATTR_SW_ALLOC_NON_CACHE)>;
};

mem_cacheable_big: memory@30000000 {
    compatible = "mmio-sram";
    reg = <0x30000000 0x10000>;
    zephyr,memory-attr = <(DT_MEM_CACHEABLE | DT_MEM_OOO | DT_MEM_SW_ALLOC_CACHE)>;
};

mem_cacheable_dma: memory@40000000 {
    compatible = "mmio-sram";
    reg = <0x40000000 0x10000>;
    zephyr,memory-attr = <(DT_MEM_CACHEABLE      | DT_MEM_DMA |
                           DT_MEM_SW_ALLOC_CACHE | DT_MEM_SW_ALLOC_DMA)>;
};
```

The user can then dynamically carve memory out of those regions using the provided functions, the library will take care of allocating memory from the correct heap depending on the provided attribute and size:

```
// Init the pool
mem_attr_heap_pool_init();

// Allocate 0x100 bytes of cacheable memory from `mem_cacheable`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_CACHE, 0x100);

// Allocate 0x200 bytes of non-cacheable memory aligned to 32 bytes
// from `mem_non_cacheable`
block = mem_attr_heap_aligned_alloc(ATTR_SW_ALLOC_NON_CACHE, 0x100, 32);

// Allocate 0x100 bytes of cacheable and dma-able memory from `mem_cacheable_dma`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_CACHE | DT_MEM_SW_ALLOC_DMA, 0x100);
```

When several regions are marked with the same attributes, the memory is allocated:

1. From the regions where the `zephyr,memory-attr` property has the requested property (or properties).
2. Among the regions as at point 1, from the smallest region if there is any unallocated space left for the requested size
3. If there is not enough space, from the next bigger region able to accommodate the requested size

The following example shows the point 3:

```
// This memory is allocated from `mem_non_cacheable`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_NON_CACHE, 0x100);

// This memory is allocated from `mem_cacheable_big`
block = mem_attr_heap_alloc(DT_MEM_SW_ALLOC_CACHE, 0x5000);
```

Note

The framework is assuming that the memory regions used to create the heaps are usable by the code and available at init time. The user must take of initializing and setting the memory area before calling `mem_attr_heap_pool_init()`.

That means that the region must be correctly configured in terms of MPU / MMU (if needed) and that an actual heap can be created out of it, for example by leveraging the `zephyr, memory-region` property to create a proper linker section to accommodate the heap.

API Reference

Memory Attribute Interface

Memory heaps based on memory attributes

4.6.2 Network Buffers

- *Overview*
- *Creating buffers*
- *Common Operations*
- *Reference Counting*
- *Ownership*
- *API Reference*

Overview

Network buffers are a core concept of how the networking stack (as well as the Bluetooth stack) pass data around. The API for them is defined in include/zephyr/net_buf.h:.

Creating buffers

Network buffers are created by first defining a pool of them:

```
NET_BUF_POOL_DEFINE(pool_name, buf_count, buf_size, user_data_size, NULL);
```

The pool is a static variable, so if it's needed to be exported to another module a separate pointer is needed.

Once the pool has been defined, buffers can be allocated from it with:

```
buf = net_buf_alloc(&pool_name, timeout);
```

There is no explicit initialization function for the pool or its buffers, rather this is done implicitly as `net_buf_alloc()` gets called.

If there is a need to reserve space in the buffer for protocol headers to be prepended later, it's possible to reserve this headroom with:

```
net_buf_reserve(buf, headroom);
```

In addition to actual protocol data and generic parsing context, network buffers may also contain protocol-specific context, known as user data. Both the maximum data and user data capacity of the buffers is compile-time defined when declaring the buffer pool.

The buffers have native support for being passed through k_fifo kernel objects. Use `k_fifo_put()` and `k_fifo_get()` to pass buffer from one thread to another.

Special functions exist for dealing with buffers in single linked lists, where the `net_buf_slist_put()` and `net_buf_slist_get()` functions must be used instead of `sys_slist_append()` and `sys_slist_get()`.

Common Operations

The network buffer API provides some useful helpers for encoding and decoding data in the buffers. To fully understand these helpers it's good to understand the basic names of operations used with them:

Add

Add data to the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Requires that there is enough tailroom in the buffer. Some examples of APIs for adding data:

```
void *net_buf_add(struct net_buf *buf, size_t len);
void *net_buf_add_mem(struct net_buf *buf, const void *mem, size_t len);
uint8_t *net_buf_add_u8(struct net_buf *buf, uint8_t value);
void net_buf_add_le16(struct net_buf *buf, uint16_t value);
void net_buf_add_le32(struct net_buf *buf, uint32_t value);
```

Remove

Remove data from the end of the buffer. Modifies the data length value while leaving the actual data pointer intact. Some examples of APIs for removing data:

```
void *net_buf_remove_mem(struct net_buf *buf, size_t len);
uint8_t net_buf_remove_u8(struct net_buf *buf);
uint16_t net_buf_remove_le16(struct net_buf *buf);
uint32_t net_buf_remove_le32(struct net_buf *buf);
```

Push

Prepend data to the beginning of the buffer. Modifies both the data length value as well as the data pointer. Requires that there is enough headroom in the buffer. Some examples of APIs for pushing data:

```
void *net_buf_push(struct net_buf *buf, size_t len);
void *net_buf_push_mem(struct net_buf *buf, const void *mem, size_t len);
void net_buf_push_u8(struct net_buf *buf, uint8_t value);
void net_buf_push_le16(struct net_buf *buf, uint16_t value);
```

Pull

Remove data from the beginning of the buffer. Modifies both the data length value as well as the data pointer. Some examples of APIs for pulling data:

```
void *net_buf_pull(struct net_buf *buf, size_t len);
void *net_buf_pull_mem(struct net_buf *buf, size_t len);
uint8_t net_buf_pull_u8(struct net_buf *buf);
uint16_t net_buf_pull_le16(struct net_buf *buf);
uint32_t net_buf_pull_le32(struct net_buf *buf);
```

The Add and Push operations are used when encoding data into the buffer, whereas the Remove and Pull operations are used when decoding data from a buffer.

Reference Counting

Each network buffer is reference counted. The buffer is initially acquired from a free buffers pool by calling `net_buf_alloc()`, resulting in a buffer with reference count 1. The reference count can be incremented with `net_buf_ref()` or decremented with `net_buf_unref()`. When the count drops to zero the buffer is automatically placed back to the free buffers pool.

Ownership

Each reference to a buffer has exactly one owner. The owner either releases its reference with `net_buf_unref()` or moves it to a new owner, such as a FIFO, a list or a function that takes ownership of the buffer. After the move the previous owner must no longer use the buffer, unless it holds another reference of its own.

A fragment chain is owned through its head. Each buffer in the chain owns a reference to the next fragment and releases it when the buffer itself is freed, so releasing the last reference to the head releases the chain up to the first fragment that is still referenced elsewhere. `net_buf_frag_insert()` takes ownership of the fragment it is given, and so does `net_buf_frag_add()` when the head is not `NULL`. With a `NULL` head, `net_buf_frag_add()` returns the fragment with a new reference and the caller keeps its own.

A function or callback that is given a buffer does one of the following with the reference, and its documentation should say which:

Borrows
: The function may use the buffer until it returns, but may neither keep nor move it. To keep a borrowed buffer, it acquires a reference of its own with `net_buf_ref()` first.

Takes ownership
: The reference moves to the function on every return, errors included.

Takes ownership on success
: The reference moves only if the function succeeds. On error the caller still owns the buffer.

Two helpers operate on the pointer that holds a reference, so that the previous owner is left with `NULL` instead of a pointer to a buffer it no longer owns. `net_buf_take()` moves the reference out of the pointer, and `net_buf_drop()` releases it. `net_buf_drop()` is meant for a pointer that outlives the release, such as a structure member, or one that may already be `NULL`. A local pointer that holds a reference as it goes out of scope only needs `net_buf_unref()`.

Putting a buffer in a FIFO or a list moves the reference to the queue. The receiving side may process and free the buffer even before `k_fifo_put()` returns, for instance when a higher-priority thread is waiting on the FIFO. Move the reference with `net_buf_take()` in the same expression:

```
k_fifo_put(&tx_queue, net_buf_take(&buf));
```

Do this even where `buf` goes out of scope right after the call: a plain pointer argument looks the same as one that is only borrowed, while `net_buf_take()` shows at the call site that the reference moves.

Passing `net_buf_take(&buf)` as an argument only works for calls that always take ownership, such as `k_fifo_put()`, `net_buf_slist_put()` and `net_buf_frag_insert()`. When a function that

takes ownership only on success is given a plain pointer, the caller's pointer is unchanged either way, and only the return value tells whether the buffer is still its own.

New functions that take ownership should therefore take a pointer to the caller's buffer pointer, and move the reference out with `net_buf_take()` when they take it. The caller's pointer is then `NULL` exactly when ownership has moved, and `net_buf_drop()` is correct after the call whatever the outcome:

```
int foo_send(struct foo *foo, struct net_buf **buf)
{
    if (!foo->ready) {
        return -EAGAIN;
    }

    k_fifo_put(&foo->tx_queue, net_buf_take(buf));

    return 0;
}

err = foo_send(foo, &buf);
if (err != 0) {
    LOG_WRN("Not sent (err %d)", err);
}

/* Releases the buffer only if foo_send() did not take it */
net_buf_drop(&buf);
```

Changing an existing function to this form breaks its callers, so it is best done when the function is reworked anyway.

API Reference

Network Buffer Library

Power Management

4.7 Power Management

This section covers services for controlling energy consumption and performance: system and device power management, CPU frequency scaling, controlled shutdown, and cooperative resource lifecycle management.

4.7.1 Power Management

Zephyr RTOS power management subsystem provides several means for a system integrator to implement power management support that can take full advantage of the power saving features of SOCs.

Overview

The interfaces and APIs provided by the power management subsystem are designed to be architecture and SOC independent. This enables power management implementations to be easily adapted to different SOCs and architectures.

The architecture and SOC independence is achieved by separating the core PM infrastructure from implementations of the SOC specific components. Thus a coherent abstraction is presented to the rest of the OS and the application layer.

The power management features are classified into the following categories.

- System Power Management
- Device Power Management

System Power Management

Introduction The kernel enters the idle state when it has nothing to schedule. Enabling `CONFIG_PM` allows the kernel to call upon the power management subsystem to put an idle system into one of the supported power states. The kernel requests an amount of time it would like to suspend, then the PM subsystem decides the appropriate power state to transition to based on the configured power management policy.

It is the application's responsibility to set up a wake-up event. A wake-up event will typically be an interrupt triggered by an SoC peripheral module. Examples include a SysTick, RTC, counter, or GPIO. Keep in mind that depending on the SoC and the power mode in question, not all peripherals may be active, and therefore some wake-up sources may not be usable in all power modes.

The following diagram describes system power management:

The idle thread locks interrupts before calling `pm_system_suspend()` and keeps ownership of the original architecture interrupt key. If the PM subsystem enters a low-power state, PM resume bookkeeping runs with interrupts still locked. By default, system-managed devices are resumed first, followed by `pm_state_exit_post_ops()`, PM exit notifications, and system clock idle-exit accounting. A device resume callback cannot rely on a timer-based `k_busy_wait()` if the low-power state stopped that timer. After that bookkeeping is complete and `pm_system_suspend()` returns, the idle thread restores the original interrupt key.

> **Note**
>
> The default-off experimental option `CONFIG_PM_EARLY_SYSTEM_TIMER_RESUME` changes the sequence to `pm_state_exit_post_ops()`, system clock idle-exit accounting, device resume, then PM exit notifications. This can make a timer-based `k_busy_wait()` usable in device resume callbacks, but requires platform-specific validation. SoC post-ops must not depend on resumed devices, and timer idle-exit must not depend on still-suspended devices. Timer idle-exit may also invoke `k_timer` expiry functions before devices are resumed.

Architectures and SoCs that do not select `CONFIG_PM_STATE_SET_IRQ_UNLOCKED` use the architecture hooks immediately around the low-power instruction so that it can still observe a wake event without dispatching the wake-source ISR first. In that mode, `pm_state_set()` and `pm_state_exit_post_ops()` must be used for SoC-specific hardware work only, not as final interrupt unmask points.

When this default contract is used, the interrupt ownership sequence is:

SoC implementations that use this default contract must not unmask interrupts from `pm_state_set()` or `pm_state_exit_post_ops()`. Legacy implementations that still call `irq_unlock(0)`, `arch_irq_unlock(0)`, `__enable_irq()`, or equivalent operations from those hooks must select `CONFIG_PM_STATE_SET_IRQ_UNLOCKED` until they are migrated.

> **Note**
>
> The ordering guarantee above applies only to interrupts that `arch_irq_lock()` can mask. Zero-latency interrupts are outside this ordering. See *Zero Latency Interrupts*.

Power States The power management subsystem defines a set of states described by the power consumption and context retention associated with each of them.

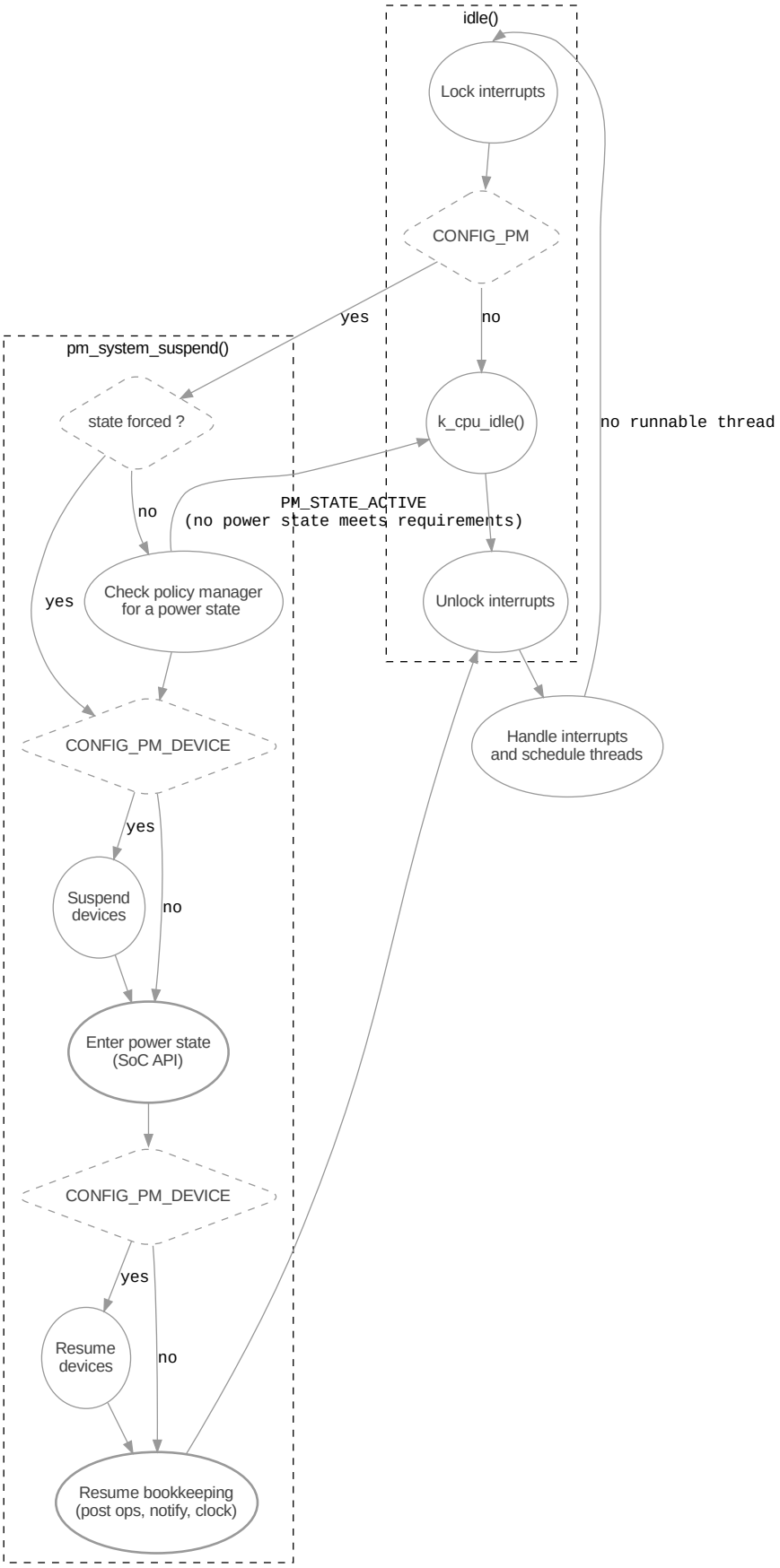


Fig. 38: System power management

Fig. 39: System PM interrupt restore ownership

The set of power states is defined by `pm_state`. In general, lower power states (higher index in the enum) will offer greater power savings and have higher wake latencies.

Power Management Policies The power management subsystem supports the following power management policies:

- Residency based
- Application defined

The policy manager is the component of the power management subsystem responsible for deciding which power state the system should transition to. The policy manager can only choose between states that have been defined for the platform. Other constraints placed upon the decision may include locks disallowing certain power states, or various kinds of minimum and maximum latency values, depending on the policy.

More details on the states definition can be found in the `zephyr,power-state` binding documentation.

Residency Under the residency policy, the system will enter the power state which offers the highest power savings, with the constraint that the sum of the minimum residency value (see `zephyr,power-state`) and the latency to exit the mode must be less than or equal to the system idle time duration scheduled by the kernel.

Thus the core logic can be summarized with the following expression:

```
if (time_to_next_scheduled_event >= (state.min_residency_us + state.exit_latency))
↪{
   return state
}
```

Application The application defines the power management policy by implementing the `pm_policy_next_state()` function. In this policy, the application is free to decide which power state the system should transition to based on the remaining time until the next scheduled timeout.

An example of an application that defines its own policy can be found in tests/subsys/pm/power_mgmt/.

Custom ticks hook In addition to the kernel tick expiration and the PM policy event list, applications and SoC-specific code may need to derive the next wake-up time from proprietary or hardware-specific data sources. Examples include hardware registers, binary-only third-party modules, or complex/legacy data structures that are not practical to model as standard PM policy events.

When `CONFIG_PM_CUSTOM_TICKS_HOOK` is enabled, the PM core calls the optional function `pm_policy_next_custom_ticks()` during `pm_system_suspend()`. The hook returns the ticks to the next custom event, or `K_TICKS_FOREVER` if no custom event needs to be considered.

The PM core then selects the earliest expiration among:

- kernel tick expiration,
- PM policy event list ticks (from `pm_policy_next_event_ticks()`),
- the custom ticks value (from `pm_policy_next_custom_ticks()`).

This mechanism allows boards and applications to integrate additional timing sources into the system power management decisions without modifying the PM event list infrastructure.

Policy and Power States The power management subsystem allows different Zephyr components and applications to configure the policy manager to block the system from transitioning into certain power states. This can be used by devices when executing tasks in background to prevent the system from going to a specific state where it would lose context. See `pm_policy_state_lock_get()`.

Examples Some helpful examples showing different power management features:

- samples/boards/st/power_mgmt/blinky/
- samples/boards/espressif/deep_sleep/
- samples/subsys/pm/device_pm/
- tests/subsys/pm/power_mgmt/
- tests/subsys/pm/power_mgmt_soc/
- tests/subsys/pm/power_states_api/

Device Power Management

Introduction Device Power Management in Zephyr is a feature which presents mechanisms to coherently affect the control of power management actions to be taken by device drivers. This control is based on unambiguous expectations which could be set by any component of the system, and on power-related dependencies that devices may have on each other.

Zephyr supports two methods of device power management:

- *Device Runtime Power Management*
- *System-Managed Device Power Management*

Device Runtime Power Management Device runtime power management involves coordinated interaction between device drivers, subsystems, and applications. While device drivers play a crucial role in directly controlling the power state of devices, the decision to suspend or resume a device can also be influenced by higher layers of the software stack.

Each layer—device drivers, subsystems, and applications—can operate independently without needing to know about the specifics of the other layers because the subsystem uses reference count to check when it needs to suspend or resume a device.

- **Device drivers** are responsible for managing the power state of devices. They interact directly with the hardware to put devices into low-power states (suspend) when they are not in use, and bring them back (resume) when needed. Drivers should use the *device runtime power management APIs* provided by Zephyr to control the power state of devices.
- **Subsystems**, such as sensors, file systems, and network, can also influence device power management. Subsystems may have better knowledge about the overall system state and workload, allowing them to make informed decisions about when to suspend or resume devices. For example, a networking subsystem may decide to keep a network interface powered on if it expects network activity in the near future.
- **Applications** running on Zephyr can impact device power management as well. An application may have specific requirements regarding device usage and power consumption. For example, an application that streams data over a network may need to keep the network interface powered on continuously.

Coordination between device drivers, subsystems, and applications is key to efficient device power management. For example, a device driver may not know that a subsystem will perform a series of sequential operations that require a device to remain powered on. In such cases, the subsystem can use device runtime power management to ensure that the device remains in an active state until the operations are complete.

When using this Device Runtime Power Management, the System Power Management subsystem is able to change power states quickly because it does not need to spend time suspending and resuming devices that are runtime enabled.

For more information, see *Device Runtime Power Management*.

System-Managed Device Power Management The system managed device power management (PM) framework is a method where devices are suspended along with the system entering a CPU (or SoC) power state. It can be enabled by setting `CONFIG_PM_DEVICE_SYSTEM_MANAGED`. When using this method, device power management is mostly done inside `pm_system_suspend()`.

If a decision to enter a CPU lower power state is made, the power management subsystem will check if the selected low power state triggers device power management and then suspend devices before changing state. The subsystem takes care of suspending devices following their initialization order, ensuring that possible dependencies between them are satisfied. As soon as the CPU wakes up from a sleep state, devices are resumed in the opposite order that they were suspended.

The decision about suspending devices when entering a low power state is done based on the state and if it has set the property `zephyr,pm-device-disabled`. Here is an example of a target with two low power states with only one triggering device power management:

```
/* Node in a DTS file */
cpus {
     power-states {
             state0: state0 {
                     compatible = "zephyr,power-state";
                     power-state-name = "standby";
                     min-residency-us = <5000>;
                     exit-latency-us = <240>;
                     zephyr,pm-device-disabled;
             };
             state1: state1 {
                     compatible = "zephyr,power-state";
                     power-state-name = "suspend-to-ram";
                     min-residency-us = <8000>;
                     exit-latency-us = <360>;
             };
     };
};
```

Note

When using *System Power Management*, device transitions can be run from the idle thread. As functions in this context cannot block, transitions that intend to use blocking APIs **must** check whether they can do so with `k_can_yield()`.

This method of device power management can be useful in the following scenarios:

- Systems with no device requiring any blocking operations when suspending and resuming. This implementation is reasonably simpler than device runtime power management.
- For devices that can not make any power management decision and have to be always active. For example a firmware using Zephyr that is controlled by an external entity (e.g Host CPU). In this scenario, some devices have to be always active and should be suspended together with the SoC when requested by this external entity.

It is important to emphasize that this method has drawbacks (see above) and *Device Runtime Power Management* is the **preferred** method for implementing device power management.

> **Note**
>
> When using this method of device power management, the CPU will not enter a low-power state if a device cannot be suspended. For example, if a device returns an error such as `-EBUSY` in response to the `PM_DEVICE_ACTION_SUSPEND` action, indicating it is in the middle of a transaction that cannot be interrupted. Another condition that prevents the CPU from entering a low-power state is if the option `CONFIG_PM_NEED_ALL_DEVICES_IDLE` is set and a device is marked as busy.

> **Note**
>
> Devices are suspended only when the last active core is entering a low power state and devices are resumed by the first core that becomes active.

Device Power Management States The power management subsystem defines device states in `pm_device_state`. This method is used to track power states of a particular device. It is important to emphasize that, although the state is tracked by the subsystem, it is the responsibility of each device driver to handle device actions(`pm_device_action`) which change device state.

Device drivers implement the `pm_device_action_cb_t()` hook internally which receives the `pm_device_action` for the device driver to handle. If the `CONFIG_PM_DEVICE` option is selected, the device drivers implementations of the hooks are exposed to the PM subsystem, enabling runtime power management of the devices.

`pm_device_action` actions have direct and unambiguous relationships with `pm_device_state` states:

As mentioned above, device drivers do not directly change between these states. This is entirely done by the power management subsystem. Instead, drivers are responsible for implementing any hardware-specific tasks needed to handle state changes.

Device Model with Device Power Management Support Drivers initialize devices using macros. See *Device Driver Model* for details on how these macros are used. A driver which implements device power management support must provide these macros with arguments that describe its power management implementation.

Use `PM_DEVICE_DEFINE` or `PM_DEVICE_DT_DEFINE` to define the power management resources required by a driver. These macros allocate the driver-specific context which is required by the power management subsystem.

Drivers can use `PM_DEVICE_GET` or `PM_DEVICE_DT_GET` to get a pointer to this context. These pointers should be passed to `DEVICE_DEFINE` or `DEVICE_DT_DEFINE` to initialize the power management field in each `device`.

The following example code shows how to implement device power management support in a device driver. Note that return values are explicitly ignored for brevity, in real drivers they must be handled.

```
#include <zephyr/pm/device.h>
#include <zephyr/pm/device_runtime.h>

#define DT_DRV_COMPAT dummy_device

struct dummy_driver_data {
        struct gpio_callback int_pin_callback;
        const struct device *dev;
};
```

(continues on next page)

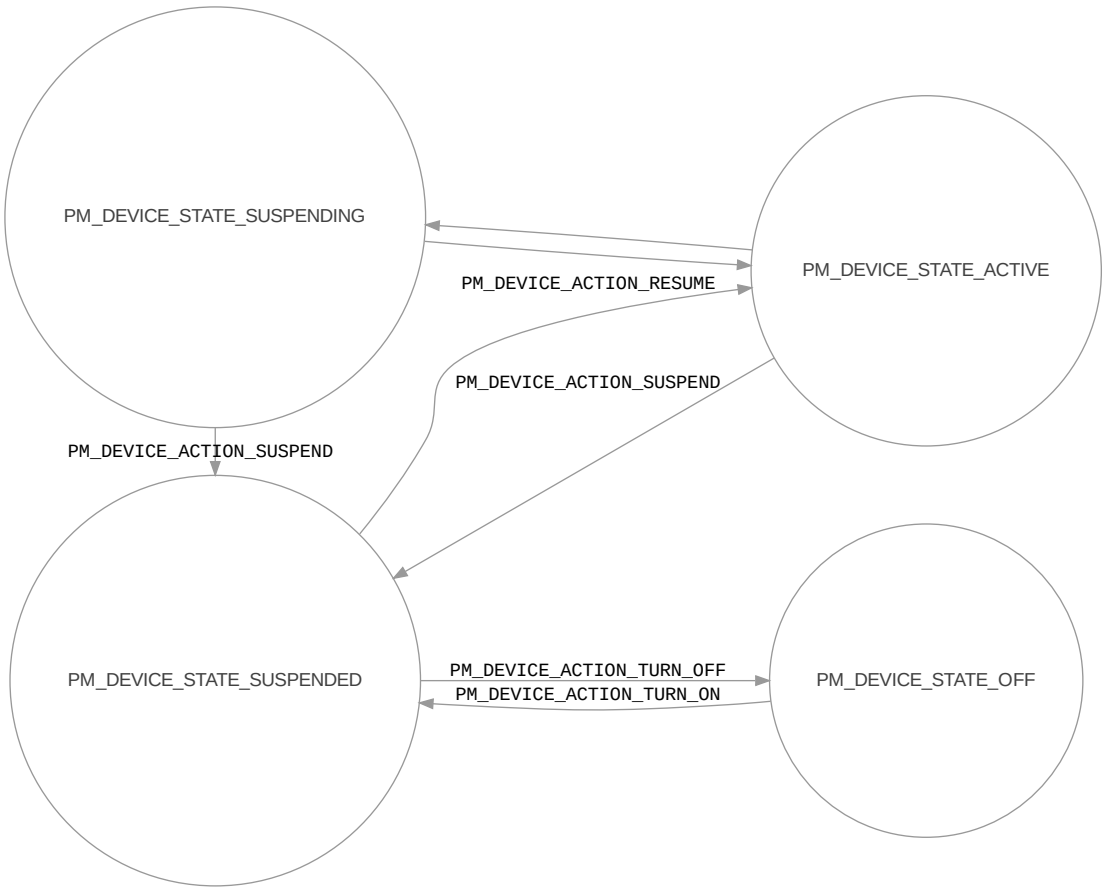


Fig. 40: Device actions x states

(continued from previous page)

```
struct dummy_driver_config {
        const struct device *bus;
        const struct gpio_dt_spec int_gpio;
        const struct gpio_dt_spec enable_pin;
};

static void dummy_driver_int_pin_handler(const struct device *dev,
                                         struct gpio_callback *cb,
                                         uint32_t pins)
{
        struct dummy_driver_data *dev_data =
                CONTAINER_OF(cb, struct dummy_driver_data, int_pin_callback);
        const struct device *dev = dev_data->dev;
        const struct dummy_driver_config *dev_config = dev->config;

        /* ... */
}

static int dummy_driver_pm_suspend(const struct device *dev)
{
        struct dummy_driver_data *dev_data = dev->data;
        const struct dummy_driver_config *config = dev->config;

        /* Request devices needed by device */
        (void)pm_device_runtime_get(config->enable_pin.port);

        /* Disable and remove interrupt pin interrupt */
        (void)gpio_pin_interrupt_configure_dt(&config->int_gpio,␣
↪GPIO_INT_DISABLED);
        (void)gpio_remove_callback(config->int_pin.port, &data->int_pin_callback);

        /* Disable the device. In this case, we use the enable pin */
        (void)gpio_pin_set_dt(&config->enable_pin, 0);

        /* Release devices currently not needed by device */
        (void)pm_device_runtime_put(config->enable_pin.port);
        (void)pm_device_runtime_put(config->int_pin.port);

        /*
         * Note that we now have suspended the device and released all the
         * devices this device depends on. We are ready for the power
         * domain being suspended, the device being resumed again, or the
         * device driver being deinitialized.
         */

        return 0;
}

static int dummy_driver_pm_resume(const struct device *dev)
{
        struct dummy_driver_data *dev_data = dev->data;
        const struct dummy_driver_config *config = dev->config;

        /* Request devices needed by device */
        (void)pm_device_runtime_get(config->enable_pin.port);
```

(continues on next page)

(continued from previous page)

```
        (void)pm_device_runtime_get(config->int_pin.port);
        (void)pm_device_runtime_get(config->bus);

        /* Enable the device. In this case, we use the enable pin */
        (void)gpio_pin_set_dt(&config->enable_pin, 1);

        /*
         * Write initial commands to device, in this case configuring
         * the device's interrupt output pin using the bus
         */

        /* ... */

        /* Add and enable interrupt pin interrupt */
        (void)gpio_add_callback(config->int_pin.port, &data->int_pin_callback);
        (void)gpio_pin_interrupt_configure_dt(&config->int_gpio,␣
↪GPIO_INT_EDGE_TO_ACTIVE);

        /*
         * Release devices currently not needed by device. In this case, we
         * are releasing the bus and the enable pin.
         *
         * The device driver would keep the bus ACTIVE while the device is
         * ACTIVE in cases of high throughput or unsolicitet data on the
         * bus, to avoid inefficient RESUME/SUSPEND cycles of the bus
         * for every transaction, and allowing reception of unsolicited
         * data on buses like UART.
         */
        (void)pm_device_runtime_put(config->bus);
        (void)pm_device_runtime_put(config->enable_pin.port);

        /*
         * Note that the interrupt pin's port is kept resumed as it
         * it needs to service the GPIO interrupt we enabled.
         */

        return 0;
}

static int dummy_driver_pm_turn_off(const struct device *dev)
{
        const struct dummy_driver_config *config = dev->config;

        /* Request devices needed for configuring device */
        (void)pm_device_runtime_get(config->enable_pin.port);

        /*
         * We prepare the device for being powered off. In this case, we
         * have an active low enable pin, which could back power the device
         * once the power domain is suspended, so we configure it as
         * disconnected if supported, input otherwise.
         */
        if (gpio_pin_configure_dt(&config->enable_pin, GPIO_DISCONNECTED)) {
                (void)gpio_pin_configure_dt(&config->enable_pin, GPIO_INPUT);
        }
```

(continues on next page)

(continued from previous page)

```
        /* Release devices needed for configuring device */
        (void)pm_device_runtime_put(config->enable_pin.port);

        /*
         * We have now prepared the device for being powered off and have
         * released all the devices this device depends on. We assume that
         * the enable pin will retain its configuration, even as we have
         * released the enable pin's port.
         */

         return 0;
}

static int dummy_driver_pm_turn_on(const struct device *dev)
{
        const struct dummy_driver_config *config = dev->config;

        /* Request devices needed for configuring device */
        (void)pm_device_runtime_get(config->enable_pin.port);
        (void)pm_device_runtime_get(config->int_gpio.port);

        /*
         * We ensure the device is suspended, and if possible in its reset
         * state. In this case we are using an enable pin, for other devices
         * we may need to reset them by toggling a reset pin, using an SoC
         * reset controller, or writing a reset command to them using their
         * bus.
         */
        (void)gpio_pin_configure_dt(&config->enable_pin, GPIO_OUTPUT_INACTIVE);

        /* We configure pins for suspended */
        (void)gpio_pin_configure_dt(&config->int_gpio, GPIO_INPUT);

        /* Release devices needed for configuring device */
        (void)pm_device_runtime_put(config->int_gpio.port);
        (void)pm_device_runtime_put(config->enable_pin.port);

        return 0;
}

static int dummy_driver_pm_action(const struct device *dev,
                                  enum pm_device_action action)
{
        int ret;

        switch (action) {
        case PM_DEVICE_ACTION_SUSPEND:
                ret = dummy_driver_pm_suspend(dev);
                break;
        case PM_DEVICE_ACTION_RESUME:
                ret = dummy_driver_pm_resume(dev);
                break;
        case PM_DEVICE_ACTION_TURN_OFF:
                ret = dummy_driver_pm_turn_off(dev);
                break;
        case PM_DEVICE_ACTION_TURN_ON:
```

(continues on next page)

(continued from previous page)

```
                ret = dummy_driver_pm_turn_on(dev);
                break;
        default:
                ret = -EINVAL;
                break;
        }

        return ret;
}

static int dummy_init(const struct device *dev)
{
        struct dummy_driver_data *dev_data = dev->data;
        const struct dummy_driver_config *dev_config = dev->config;

        /*
         * We must ensure all devices we depend on, excluding a potential
         * power domain, are initialized.
         *
         * If CONFIG_PM_DEVICE=n, this also ensures the devices are ACTIVE.
         */
        if (!device_is_ready(dev_config->bus) ||
            !gpio_is_ready_dt(&dev_config->int_pin) ||
            !gpio_is_ready_dt(&dev_config->enable_pin)) {
                return -ENODEV;
        }

        /* We then initialize the device driver data structure */
        gpio_init_callback(&dev_data->int_pin_callback,
                           dummy_driver_int_pin_handler,
                           BIT(dev_config->int_pin.pin));

        dev_data->dev = dev;

       /*
        * This call must be the last call of the device init function.
        * It will initialize the device's PM_DEVICE context and use the
        * dummy_driver_pm_action callback to initialize the device into
        * the appropriate state.
         *
         * On success, :c:func:`pm_device_driver_init` returns 0.
         * If the callback fails (for example on ``PM_DEVICE_ACTION_RESUME``),
         * the error is propagated and the device is not marked ``ACTIVE``.
        */
       return pm_device_driver_init(dev, dummy_driver_pm_action);
}

#ifdef CONFIG_DEVICE_DEINIT_SUPPORT
static int dummy_deinit(const struct device *dev)
{
        int ret;

        /*
         * This call must be the first call of the device deinit function.
         * It will use the dummy_driver_pm_action callback to move the
         * device into, or verify the device is already in, an appropriate
```

(continues on next page)

(continued from previous page)

```
         * state for deinitialization, and deinitialize the device's
         * PM_DEVICE context.
         */
        ret = pm_device_driver_deinit(dev, dummy_driver_pm_action);
        if (ret) {
                return ret;
        }

        /*
         * The device is now either SUSPENDED or OFF, all the devices this
         * device depends on have been released, and devices with persistent
         * configurations like GPIO pins have been configured to match the
         * device state.
         *
         * The device will be left in this state until a new "owner" takes
         * over.
         */

        /*
         * If we had allocated memory, DMA channels or other resources, we would
         * release them here.
         */

        return ret;
}
#endif

static struct dummy_driver_data data0;

static struct dummy_driver_config config0 = {
        .bus = DEVICE_DT_GET(DT_INST_PARENT(0)),
        .int_pin = GPIO_DT_SPEC_INST_GET(0, int_gpios),
        .enable_pin = GPIO_DT_SPEC_INST_GET(0, enable_gpios),
};

/* Define the device's PM DEVICE context */
PM_DEVICE_DT_INST_DEFINE(0, dummy_driver_pm_action);

/* Define the device, pointing to the device's PM DEVICE context */
DEVICE_DT_INST_DEINIT_DEFINE(
        0,
        &dummy_init,
        &dummy_deinit,
        PM_DEVICE_DT_INST_GET(0),
        &data0,
        &config0,
        POST_KERNEL,
        CONFIG_KERNEL_INIT_PRIORITY_DEFAULT,
        NULL
);
```

Device Model with Partial Device Power Management Support If `CONFIG_PM_DEVICE` is not enabled, the device power state is tied to the devices initialization state.

Once a device is initialized, the device driver PM action hook is used to move the device to the expected initial state through calling `pm_device_driver_init()`. Following the De-

vice actions x states graph and the definition of the OFF state, this results in a call to PM_DEVICE_ACTION_TURN_ON followed by PM_DEVICE_ACTION_RESUME.

If the driver's PM hook returns an error, initialization fails and the device is left in its previous state (for devices that support PM this is typically SUSPENDED or OFF).

Given power domains and buses are "just devices", every power domain and bus will be resumed before its child devices as they are initialized according to the devicetree dependency ordinals. Every device is assumed to be powered, and the devices a device depends on are assumed to be ACTIVE, when device is initialized.

Once a device is deinitialized, the device driver PM action hook is used to move the device to the SUSPENDED state through calling pm_device_driver_deinit(). Following the Device actions x states, and assuming power domains are "always on" this results in a call to PM_DEVICE_ACTION_SUSPEND.

Shell Commands Power management actions can be triggered from shell commands for testing purposes. To do that, enable the CONFIG_PM_DEVICE_SHELL option and issue a pm command on a device from the shell, for example:

```
uart:~$ device list
- buttons (active)
uart:~$ pm suspend buttons
uart:~$ device list
devices:
- buttons (suspended)
```

To print the power management state of a device, enable CONFIG_DEVICE_SHELL and use the device list command, for example:

```
uart:~$ device list
devices:
- i2c@40003000 (active)
- buttons (active, usage=1)
- leds (READY)
```

In this case, leds does not support PM, i2c supports PM with manual suspend and resume actions and it's currently active, buttons supports runtime PM and it's currently active with one user.

Busy Status Indication When the system is idle and the SoC is going to sleep, the power management subsystem can suspend devices, as described in *System-Managed Device Power Management*. This can cause device hardware to lose some states. Suspending a device which is in the middle of a hardware transaction, such as writing to a flash memory, may lead to undefined behavior or inconsistent states. This API guards such transactions by indicating to the kernel that the device is in the middle of an operation and should not be suspended.

When pm_device_busy_set() is called, the device is marked as busy and the system will not do power management on it. After the device is no longer doing an operation and can be suspended, it should call pm_device_busy_clear().

Device Power Management X System Power Management When managing power in embedded systems, it's crucial to understand the interplay between device power state and the overall system power state. Some devices may have dependencies on the system power state. For example, certain low-power states of the SoC might not supply power to peripheral devices, leading to problems if the device is in the middle of an operation. Proper coordination is essential to maintain system stability and data integrity.

To avoid this sort of problem, devices must *get and release lock* power states that cause power loss during an operation.

Zephyr provides a mechanism for devices to declare which power states cause power loss and an API that automatically get and put lock on them. This feature is enabled setting `CONFIG_PM_POLICY_DEVICE_CONSTRAINTS` to y.

Once this feature is enabled, devices must declare in devicetree which states cause power loss. In the following example, device `test_dev` says that power states `state1` and `state2` cause power loss.

```
power-states {
        state0: state0 {
                compatible = "zephyr,power-state";
                power-state-name = "suspend-to-idle";
                min-residency-us = <10000>;
                exit-latency-us = <100>;
        };

        state1: state1 {
                compatible = "zephyr,power-state";
                power-state-name = "standby";
                min-residency-us = <20000>;
                exit-latency-us = <200>;
        };

        state2: state2 {
                compatible = "zephyr,power-state";
                power-state-name = "suspend-to-ram";
                min-residency-us = <50000>;
                exit-latency-us = <500>;
        };

        state3: state3 {
                compatible = "zephyr,power-state";
                power-state-name = "suspend-to-ram";
                status = "disabled";
        };
};

test_dev: test_dev {
        compatible = "test-device-pm";
        status = "okay";
        zephyr,disabling-power-states = <&state1 &state2>;
};
```

After that devices can lock these state calling `pm_policy_device_power_lock_get()` and release with `pm_policy_device_power_lock_put()`. For example:

```
static void timer_expire_cb(struct k_timer *timer)
{
        struct test_driver_data *data = k_timer_user_data_get(timer);

        data->ongoing = false;
        k_timer_stop(timer);
        pm_policy_device_power_lock_put(data->self);
}

void test_driver_async_operation(const struct device *dev)
{
        struct test_driver_data *data = dev->data;
```

(continues on next page)

(continued from previous page)

```
        data->ongoing = true;
        pm_policy_device_power_lock_get(dev);

        /** Lets set a timer big enough to ensure that any deep
         *  sleep state would be suitable but constraints will
         *  make only state0 (suspend-to-idle) will be used.
         */
        k_timer_start(&data->timer, K_MSEC(500), K_NO_WAIT);
}
```

The same pattern applies to device paths that can produce *zero-latency interrupts*. A zero-latency ISR must not call PM APIs or other kernel APIs. If the ISR depends on resources that are unavailable in some system power states, the normal driver path that arms the interrupt source can hold the device power policy lock while that path is active. The driver should mask or disable the interrupt source before releasing the lock when the path is no longer active.

Wakeup capability Some devices are capable of waking the system up from a sleep state. When a device has such capability, applications can enable or disable this feature on a device dynamically using `pm_device_wakeup_enable()`.

This property can be set on device declaring the property `wakeup-source` in the device node in devicetree. For example, this devicetree fragment sets the `gpio0` device as a "wakeup" source:

```
gpio0: gpio@40022000 {
        compatible = "ti,cc13xx-cc26xx-gpio";
        reg = <0x40022000 0x400>;
        interrupts = <0 0>;
        status = "disabled";
        label = "GPIO_0";
        gpio-controller;
        wakeup-source;
        #gpio-cells = <2>;
};
```

By default, "wakeup" capable devices do not have this functionality enabled during the device initialization. Applications can enable this functionality later calling `pm_device_wakeup_enable()`.

> **Note**
>
> This property is **only** used by the system power management to identify devices that should not be suspended. It is responsibility of driver or the application to do any additional configuration required by the device to support it.

Examples Some helpful examples showing device power management features:

- samples/subsys/pm/device_pm/
- tests/subsys/pm/power_mgmt/
- tests/subsys/pm/device_wakeup_api/
- tests/subsys/pm/device_driver_init/

Device Runtime Power Management

Introduction The device runtime power management (PM) framework is an active power management mechanism which reduces the overall system power consumption by suspending the devices which are idle or not used independently of the system state. It can be enabled by setting `CONFIG_PM_DEVICE_RUNTIME`. In this model the device driver is responsible to indicate when it needs the device and when it does not. This information is used to determine when to suspend or resume a device based on usage count.

When device runtime power management is enabled on a device, its state will be initially set to a `PM_DEVICE_STATE_SUSPENDED` indicating it is not used. On the first device request, it will be resumed and so put into the `PM_DEVICE_STATE_ACTIVE` state. The device will remain in this state until it is no longer used. At this point, the device will be suspended until the next device request. If the suspension is performed synchronously the device will be immediately put into the `PM_DEVICE_STATE_SUSPENDED` state, whereas if it is performed asynchronously, it will be put into the `PM_DEVICE_STATE_SUSPENDING` state first and then into the `PM_DEVICE_STATE_SUSPENDED` state when the action is run.

For devices on a power domain (via the devicetree 'power-domains' property), device runtime power management automatically attempts to request and release the dependent domain in response to `pm_device_runtime_get()` and `pm_device_runtime_put()` calls on the child device.

For the previous to automatically control the power domain state, device runtime PM must be enabled on the power domain device. To enable device runtime PM globally, enable `CONFIG_PM_DEVICE_RUNTIME_DEFAULT_ENABLE`. To enable device runtime PM only for select devices, set the `zephyr,pm-device-runtime-auto` devicetree property, or use `pm_device_runtime_enable()` for the select devices.

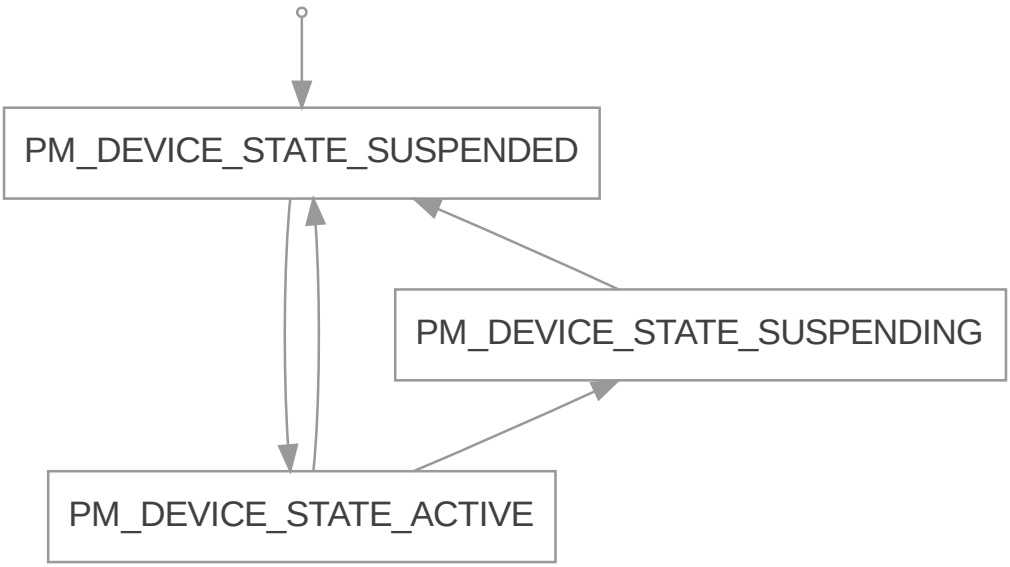


Fig. 41: Device states and transitions

The device runtime power management framework has been designed to minimize devices power consumption with minimal application work. Device drivers are responsible for indicating when they need the device to be operational and when they do not. Therefore, applications can not manually suspend or resume a device. An application can, however, decide when to disable or enable runtime power management for a device. This can be useful, for example, if

an application wants a particular device to be always active.

Design principles When runtime PM is enabled on a device it will no longer be resumed or suspended during system power transitions. Instead, the device is fully responsible to indicate when it needs a device and when it does not. The device runtime PM API uses reference counting to keep track of device's usage. This allows the API to determine when a device needs to be resumed or suspended. The API uses the *get* and *put* terminology to indicate when a device is needed or not, respectively. This mechanism plays a key role when we account for device dependencies. For example, if a bus device is used by multiple sensors, we can keep the bus active until the last sensor has finished using it.

> **Note**
>
> As of today, the device runtime power management API does not manage device dependencies. This effectively means that, if a device depends on other devices to operate (e.g. a sensor may depend on a bus device), the bus will be resumed and suspended on every transaction. In general, it is more efficient to keep parent devices active when their children are used, since the children may perform multiple transactions in a short period of time. Until this feature is added, devices can manually *get* or *put* their dependencies.

The `pm_device_runtime_get()` function can be used by a device driver to indicate it *needs* the device to be active or operational. This function will increase device usage count and resume the device if necessary. Similarly, the `pm_device_runtime_put()` function can be used to indicate that the device is no longer needed. This function will decrease the device usage count and suspend the device if necessary. It is worth to note that in both cases, the operation is carried out synchronously. The sequence diagram shown below illustrates how a device can use this API and the expected sequence of events.

The synchronous model is as simple as it gets. However, it may introduce unnecessary delays since the application will not get the operation result until the device is suspended (in case device is no longer used). It will likely not be a problem if the operation is fast, e.g. a register toggle. However, the situation will not be the same if suspension involves sending packets through a slow bus. For this reason the device drivers can also make use of the `pm_device_runtime_put_async()` function. This function will schedule the suspend operation, again, if device is no longer used.

By default, runtime PM operations are offloaded to the system work queue. However, device drivers must not perform any blocking operations during suspend, as this can stall the system work queue and negatively impact system responsiveness.

To address this, applications can configure runtime PM to use a dedicated work queue by enabling `CONFIG_PM_DEVICE_RUNTIME_USE_DEDICATED_WQ`.

If blocking behavior is required—for example, when accessing a slow peripheral or waiting for a bus transaction—the PM subsystem work queue must be used instead. Drivers that require this behavior can explicitly request it by enabling `CONFIG_PM_DEVICE_DRIVER_NEEDS_DEDICATED_WQ`.

For targets with constrained resources that do not need asynchronous operations, this functionality can be disabled altogether by de-selecting `CONFIG_PM_DEVICE_RUNTIME_ASYNC`, reducing memory usage and system complexity.

Implementation guidelines In a first place, a device driver needs to implement the PM action callback used by the PM subsystem to suspend or resume devices.

```
static int mydev_pm_action(const struct device *dev,
                           enum pm_device_action action)
{
    switch (action) {
```

(continues on next page)

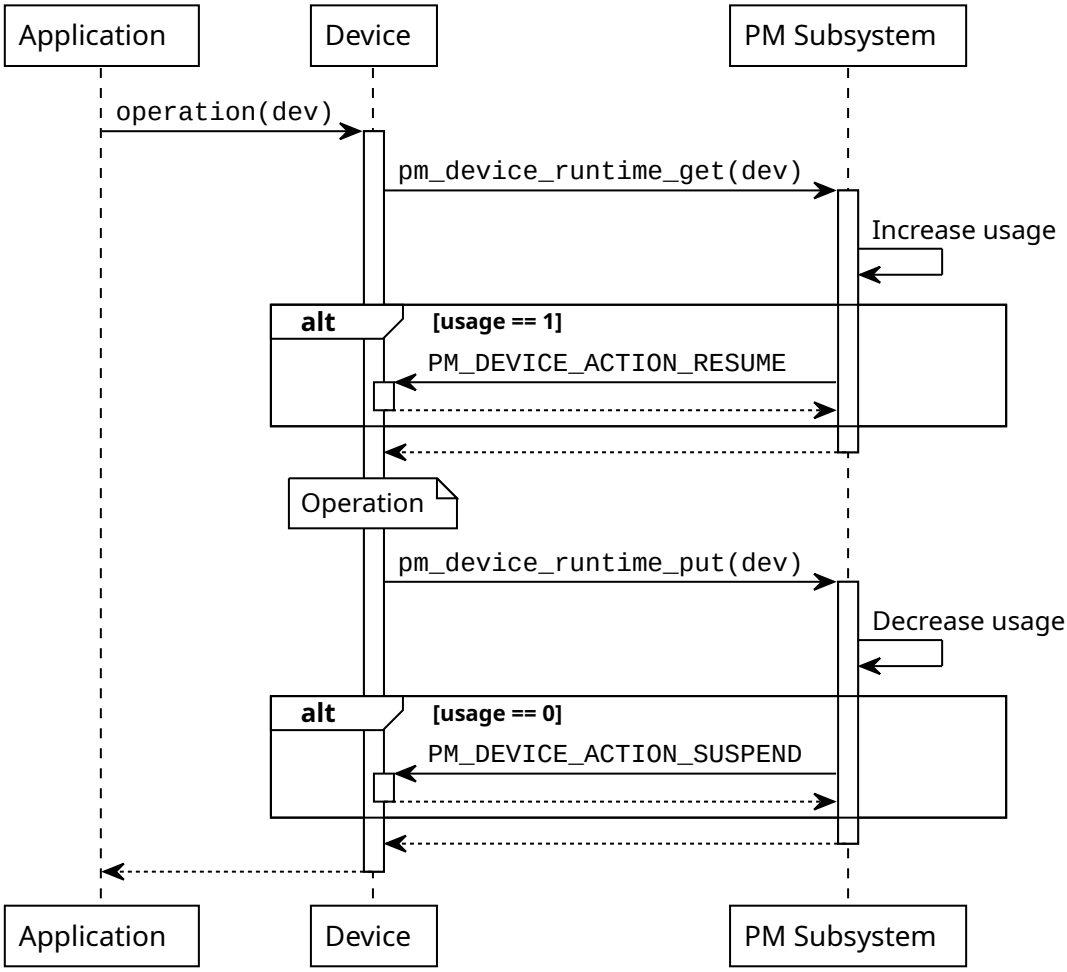


Fig. 42: Synchronous operation on a single device

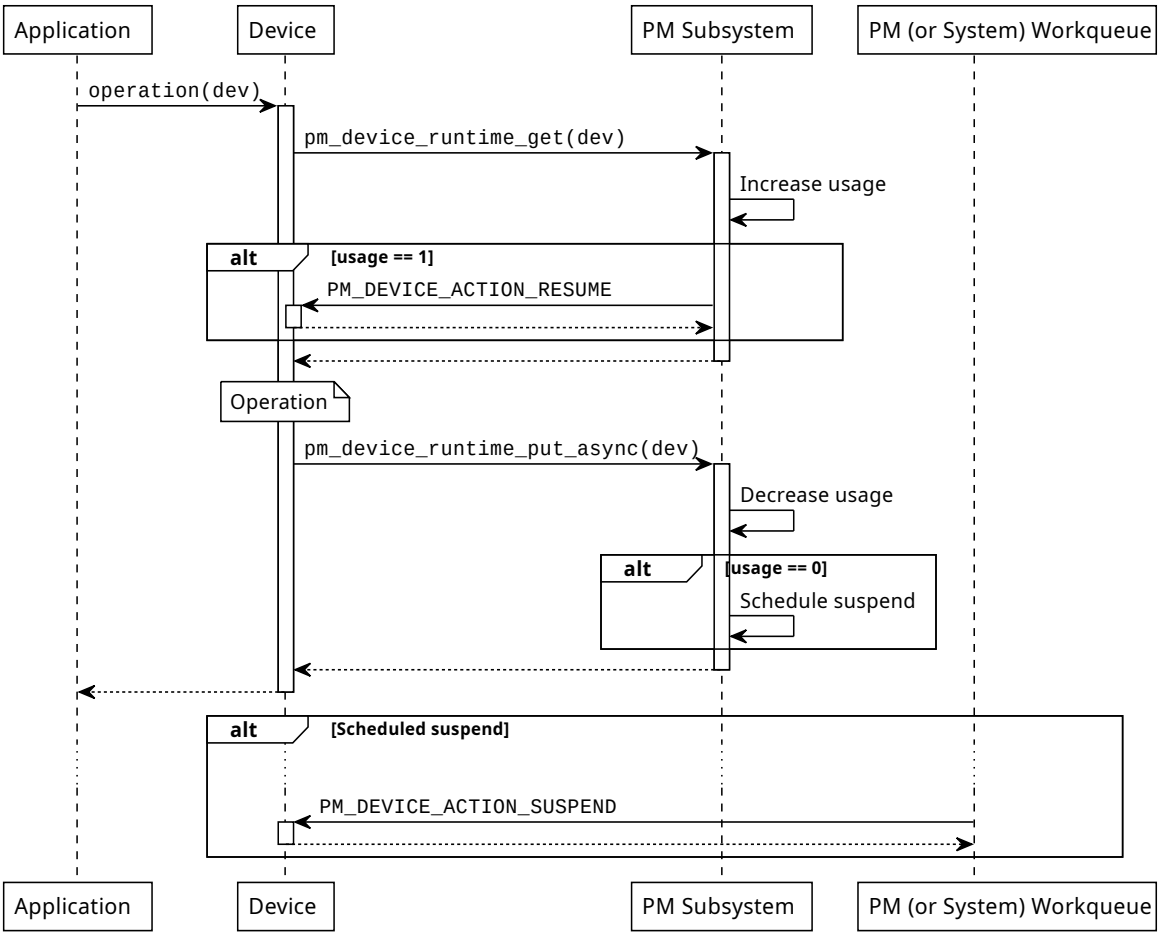


Fig. 43: Asynchronous operation on a single device

(continued from previous page)

```
    case PM_DEVICE_ACTION_SUSPEND:
        /* suspend the device */
        ...
        break;
    case PM_DEVICE_ACTION_RESUME:
        /* resume the device */
        ...
        break;
    default:
        return -ENOTSUP;
    }

    return 0;
}
```

The PM action callback calls are serialized by the PM subsystem, therefore, no special synchronization is required.

To enable device runtime power management on a device, the driver needs to call `pm_device_runtime_enable()` at initialization time. Note that this function will suspend the device if its state is `PM_DEVICE_STATE_ACTIVE`. In case the device is physically suspended, the init function should call `pm_device_init_suspended()` before calling `pm_device_runtime_enable()`.

```
/* device driver initialization function */
static int mydev_init(const struct device *dev)
{
    int ret;
    ...

    /* OPTIONAL: mark device as suspended if it is physically suspended */
    pm_device_init_suspended(dev);

    /* enable device runtime power management */
    ret = pm_device_runtime_enable(dev);
    if (ret < 0) {
        return ret;
    }
}
```

Device runtime power management can also be automatically enabled on a device instance by adding the `zephyr,pm-device-runtime-auto` flag onto the corresponding devicetree node. If enabled, `pm_device_runtime_enable()` is called immediately after the `init` function of the device runs and returns successfully.

```
foo {
    /* ... */
    zephyr,pm-device-runtime-auto;
};
```

Assuming an example device driver that implements an `operation` API call, the *get* and *put* operations could be carried out as follows:

```
static int mydev_operation(const struct device *dev)
{
    int ret;

    /* "get" device (increases usage count, resumes device if suspended) */
```

(continues on next page)

(continued from previous page)

```
    ret = pm_device_runtime_get(dev);
    if (ret < 0) {
        return ret;
    }

    /* do something with the device */
    ...

    /* "put" device (decreases usage count, suspends device if no more users) */
    return pm_device_runtime_put(dev);
}
```

In case the suspend operation is *slow*, the device driver can use the asynchronous API:

```
static int mydev_operation(const struct device *dev)
{
    int ret;

    /* "get" device (increases usage count, resumes device if suspended) */
    ret = pm_device_runtime_get(dev);
    if (ret < 0) {
        return ret;
    }

    /* do something with the device */
    ...

    /* "put" device (decreases usage count, schedule suspend if no more users) */
    return pm_device_runtime_put_async(dev, K_NO_WAIT);
}
```

Examples Some helpful examples showing device runtime power management features:

- tests/subsys/pm/device_runtime_api/
- tests/subsys/pm/device_power_domains/
- tests/subsys/pm/power_domain/

Power Domain

Introduction The Zephyr power domain abstraction is designed to support groupings of devices powered by a common source to be notified of power source state changes in a generic fashion. Application code that is using device A does not need to know that device B is on the same power domain and should also be configured into a low power state.

Power domains are optional on Zephyr, to enable this feature the option `CONFIG_PM_DEVICE_POWER_DOMAIN` has to be set.

When a power domain turns itself on or off, it is the responsibility of the power domain to notify all devices using it through their power management callback called with `PM_DEVICE_ACTION_TURN_ON` or `PM_DEVICE_ACTION_TURN_OFF` respectively. This work flow is illustrated in the diagram below.

Internal Power Domains Most of the devices in an SoC have independent power control that can be turned on or off to reduce power consumption. But there is a significant amount of static current leakage that can't be controlled only using device power management. To solve this problem, SoCs normally are divided into several regions grouping devices that are generally

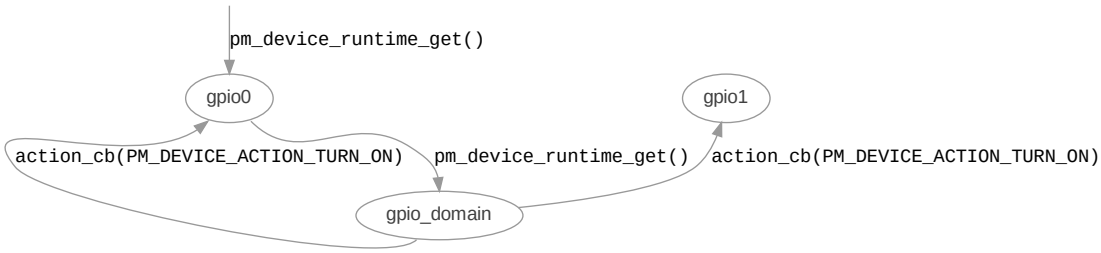


Fig. 44: Power domain work flow

used together, so that an unused region can be completely powered off to eliminate this leakage. These regions are called "power domains", can be present in a hierarchy and can be nested.

External Power Domains Devices external to a SoC can be powered from sources other than the main power source of the SoC. These external sources are typically a switch, a regulator, or a dedicated power IC. Multiple devices can be powered from the same source, and this grouping of devices is typically called a "power domain".

Placing devices on power domains can be done for a variety of reasons, including to enable devices with high power consumption in low power mode to be completely turned off when not in use.

Implementation guidelines In a first place, a device that acts as a power domain needs to declare compatible with `power-domain`. Taking *Power domain work flow* as example, the following code defines a domain called `gpio_domain`.

```
gpio_domain: gpio_domain@4 {
        compatible = "power-domain";
        ...
};
```

A power domain needs to implement the PM action callback used by the PM subsystem to turn devices on and off.

```
static int mydomain_pm_action(const struct device *dev,
                              enum pm_device_action action)
{
    switch (action) {
    case PM_DEVICE_ACTION_RESUME:
        /* resume the domain */
        ...
        /* notify children domain is now powered */
        pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_ON, NULL);
        break;
    case PM_DEVICE_ACTION_SUSPEND:
        /* notify children domain is going down */
        pm_device_children_action_run(dev, PM_DEVICE_ACTION_TURN_OFF, NULL);
        /* suspend the domain */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_ON:
```

(continues on next page)

(continued from previous page)

```
        /* turn on the domain (e.g. setup control pins to disabled) */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_OFF:
        /* turn off the domain (e.g. reset control pins to default state) */
        ...
        break;
    default:
        return -ENOTSUP;
    }

    return 0;
}
```

Devices belonging to this device can be declared referring it in the `power-domain` node's property. The example below declares devices `gpio0` and `gpio1` belonging to domain `gpio_domain`.

```
&gpio0 {
        compatible = "zephyr,gpio-emul";
        gpio-controller;
        power-domains = <&gpio_domain>;
};

&gpio1 {
        compatible = "zephyr,gpio-emul";
        gpio-controller;
        power-domains = <&gpio_domain>;
};
```

All devices under a domain will be notified when the domain changes state. These notifications are sent as actions in the device PM action callback and can be used by them to do any additional work required. They can safely be ignored though.

```
static int mydev_pm_action(const struct device *dev,
                           enum pm_device_action *action)
{
    switch (action) {
    case PM_DEVICE_ACTION_SUSPEND:
        /* suspend the device */
        ...
        break;
    case PM_DEVICE_ACTION_RESUME:
        /* resume the device */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_ON:
        /* configure the device into low power mode */
        ...
        break;
    case PM_DEVICE_ACTION_TURN_OFF:
        /* prepare the device for power down */
        ...
        break;
    default:
        return -ENOTSUP;
    }
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

Note

It is responsibility of driver or the application to set the domain as "wakeup" source if a device depending on it is used as "wakeup" source.

Examples Some helpful examples showing power domain features:

- tests/subsys/pm/device_power_domains/
- tests/subsys/pm/power_domain/

Power Management APIs

System PM APIs

System

States

States

Related code samples

- NXP MCX suspend-to-RAMSuspend an NXP MCXA/MCXN SoC to RAM and resume on an LPTMR timer (or button) wakeup delivered through the WUU wakeup controller.

Policy

Policy

Related code samples

- PM policy latencyConstrain allowed power states using maximum tolerable exit latency

Hooks

Hooks

Device PM APIs

Device

Related code samples

- Device Idle Power ManagementUse device runtime power management with a parent-child device hierarchy.
- Serial wakeupWake up on serial activity on STM32.
- System OffUse deep sleep on Nordic platforms.

Device Runtime PM APIs
Device Runtime

Related code samples

- Device Idle Power ManagementUse device runtime power management with a parent-child device hierarchy.
- Suspend to RAMUse suspend to RAM low power mode on STM32.

4.7.2 CPU Frequency Scaling

CPU Frequency Scaling Policies

On-Demand CPU Frequency Scaling Policy The On-Demand policy evaluates the current CPU load using the *CPU Load*, and compares it to the trigger threshold defined by the SoC P-state definition.

The On-Demand policy will iterate through the defined P-states and select the first P-state for which the CPU load is greater than or equal to the defined threshold.

If no P-state matches (i.e., the CPU load is below all thresholds), the policy will select the last P-state in the soc_pstates array (the lowest performance state). This is intrinsic to the policy: P-states must be defined in decreasing threshold order in devicetree, and the last P-state will be used for loads below all thresholds.

For an example of the on-demand policy refer to the cpu_freq_on_demand sample.

This policy is reactive. Frequency adjustments occur only after a change in system load has been observed, so it cannot anticipate sudden high loads. The policy has no notion of task deadlines and should not be considered as a real-time policy.

Pressure based CPU Frequency Scaling Policy The pressure policy evaluates the current pressure of the ready queue in order to inform system P-state transitions.

Thread pressure is calculated by the following formula:

$$P_{sys} = \frac{\sum_{t \in R}(P_{min} - prio_t + 1)}{\sum_{t \in T}(P_{min} - prio_t + 1)} \times 100$$

where

- R is the set of runnable (queued) threads
- T is the set of all threads considered for pressure
- $w_t = P_{min} - \text{prio}_t + 1$ is the weight of thread t
- P_{min} is the configured minimum priority to consider (the highest value numerically)

This produces a normalized system pressure between 0 and 100 which is then used to select an appropriate P-state as defined by the SoC or overlay file.

Once the normalized system pressure is calculated, it is treated as the system 'load' and the policy will then iterate through the available P-states of the SoC and select the first P-state for which the normalized pressure is greater than or equal to the defined threshold.

If no P-state matches (i.e., the normalized pressure is below all thresholds), the policy will select the last P-state in the soc_pstates array (the lowest performance state).

The user may tune the responsiveness of this policy by adjusting the `CONFIG_CPU_FREQ_POLICY_PRESSURE_LOWEST_PRIO` option which should be set the priority of the lowest priority thread in the system (which numerically would be the highest number). The

pressure policy will ignore threads of lower priority than this option, and due to the formulae above, will change the perceived impact of a high-priority thread running.

For an example of the pressure policy, refer to the cpu_freq_pressure sample.

This policy attempts to be proactive in evaluating queued tasks and adjusting the clock frequency ahead of their execution time, although this policy makes no guarantees on performance or determinism of threads achieving their deadlines and avoiding starvation.

Note that the `CONFIG_CPU_FREQ_POLICY_PRESSURE_LOWEST_PRIO` includes `CONFIG_TRACING` introduces overhead to context switching.

Timing Noise CPU Frequency Scaling Policy

Overview The timing noise policy is a CPU frequency scaling policy that periodically selects a random performance state (P-state). The intent is to inject timing variability by jittering the CPU clock, which can disrupt naive cycle-count or wall-clock measurements that assume a stable frequency.

This is a randomized P-state policy. It is not a general side-channel countermeasure.

Primary Mitigation Constant-time and constant-flow implementations that have been reviewed for side-channel leakage remain the primary mitigation against timing attacks. This policy does not replace those practices. At best, it may complicate a particular class of measurement.

Attacker Model This policy considers an adversary that:

- Observes relative execution time via cycle counters, wall-clock timers, or similar software-visible timestamps
- Relies on a stable CPU frequency so that small differences in instruction count or data-dependent paths remain distinguishable across trials

Randomized P-state selection does not defeat such an attack. Given enough samples, an attacker can often average through the noise. The policy only makes those measurements less convenient and less repeatable.

Use Cases This policy may be useful when an application wants additional timing variability as a secondary layer on top of already sound software. It should be treated as a way to disrupt certain analyses, not as a security solution on its own.

It is a poor fit for:

- Hard real-time systems that need deterministic execution time
- Safety-related systems where timing predictability is part of the safety case
- Latency-sensitive workloads that cannot tolerate unexpected slow P-states
- Strict power-budget designs, since frequent P-state changes may increase average power consumption

Configuration Enable the CPU frequency subsystem and select the timing noise policy:

```
CONFIG_CPU_FREQ=y
CONFIG_CPU_FREQ_POLICY_TIMING_NOISE=y
```

The policy uses the standard CPU frequency subsystem update interval, configured with:

```
CONFIG_CPU_FREQ_INTERVAL_MS=<interval>
```

Smaller update intervals increase timing variability, but also increase the number of frequency transitions.

Random Number Generation The policy selects performance states using `sys_rand32_get()`. Selecting `CONFIG_CPU_FREQ_POLICY_TIMING_NOISE` enables the entropy driver on platforms with a hardware entropy source. Prefer routing random draws through that driver when a TRNG is available:

```
CONFIG_ENTROPY_DEVICE_RANDOM_GENERATOR=y
```

Limitations

- It does not eliminate timing side channels
- It cannot compensate for fundamentally insecure software
- It may reduce overall system performance
- It may increase power consumption due to frequent performance state changes
- It offers no guarantees against a determined or well-instrumented attacker

For an example of the timing noise policy, refer to the cpu_freq_timing_noise sample.

CPU Frequency Thermal Cap

The CPU frequency thermal cap is an optional constraint layer for the CPU Frequency Scaling subsystem. The active policy selects the requested P-state, and the thermal cap limits the highest-performance P-state allowed when the configured temperature trip points are active.

This keeps performance demand and thermal mitigation separate:

- the CPU frequency policy selects the requested P-state according to its own metrics and thresholds.
- the thermal cap clamps that request to the highest-performance P-state currently allowed by temperature.

After clamping, the resulting P-state is passed to the SoC P-state driver.

Devicetree Enable the thermal cap by adding a `zephyr,cpu-freq-thermal-cap` node and enabling `CONFIG_CPU_FREQ_THERMAL_CAP`.

Example:

```
cpu_freq_thermal_cap: cpu_freq_thermal_cap {
        compatible = "zephyr,cpu-freq-thermal-cap";
        sensor = <&temp0>;
        sensor-channel = "die-temp";
        polling-delay-ms = <1000>;
        trip-active-polling-delay-ms = <100>;

        trip_0 {
                temperature-millicelsius = <85000>;
                hysteresis-millicelsius = <5000>;
                cap-pstate = <&pstate_1>;
        };

        trip_1 {
                temperature-millicelsius = <95000>;
                hysteresis-millicelsius = <3000>;
```

(continues on next page)

(continued from previous page)

```
                cap-pstate = <&pstate_2>;
        };
};
```

When `trip_0` is active, CPU frequency requests are capped to `pstate_1` or lower. When `trip_1` is active, requests are capped to `pstate_2` or lower. A cap does not force the CPU to run at that P-state; a policy request for a lower-performance P-state is left unchanged.

Thermal cap constraints use the CPU frequency P-state index order. Lower index P-states are higher-performance, higher index P-states are lower-performance and more restrictive. Each `cap-pstate` must reference a P-state in that ordered table by phandle.

Runtime behavior Temperature sampling is done from a delayable work item because sensor drivers may use blocking operations. The CPU frequency timer reads only the cached cap when applying a policy result.

If temperature sampling fails repeatedly, the cap applies the lowest-performance P-state (highest index in the table) as a fail-safe constraint.

Overview

The CPU Frequency Scaling subsystem in Zephyr provides a framework for SoC's to dynamically adjust their processor frequency based on a monitored metric and performance state (P-state) policy algorithm.

Design Goals

The CPU Frequency Scaling subsystem aims to provide a framework that allows for any policy algorithm to work with any P-state driver and allows for each policy to make use of one, or many, metrics to determine an optimal CPU frequency. The subsystem should be flexible enough to allow for SoC vendors to define custom P-states, thresholds and metrics.

P-state Policies

A P-state policy is an algorithm that determines what the optimal P-state is for the CPU based on the metrics it consumes and the thresholds defined per P-state. A policy can consume one, or many, metrics to determine the optimal CPU frequency based on the desired statistics of the system.

See *policies* for a list of standard policies.

Metrics

A P-state policy should include one or more metrics to base decisions. Examples of metrics could include percent CPU load, SoC temperature, etc.

For an example of a metric in use, see the *on_demand* policy.

Thermal Cap

The optional *CPU Frequency Thermal Cap* constrains the highest-performance P-state allowed by the active policy based on temperature trip points. It is a constraint layer, not a P-state policy, used to reduce excess heat generation and protect the SoC from thermal stress.

P-state Drivers

A SoC supporting the CPU Frequency Scaling subsystem must implement a P-state driver that implements `cpu_freq_pstate_set()` which applies the passed in `p_state` to the CPU when called.

A SoC must also provide the available P-states in devicetree by having a `zephyr,pstate` compatible node. The SoC may also define its own P-state binding, which extends `zephyr,pstate` to include additional properties that may be used by the SoC's P-state driver.

Usage considerations

The CPU Frequency Scaling subsystem is designed to work on both UP and SMP system. On SMP systems, it is assumed by default that each of the CPUs are clocked at the same rate. Thus, should one CPU undergo a P-state transition, then all other CPUs will also undergo the same P-state transition. This can be overridden by the SoC by enabling the `CONFIG_CPU_FREQ_PER_CPU_SCALING` configuration option to allow each CPU to be clocked independently.

The SoC supporting CPU Frequency Scaling must uphold Zephyr's requirement that the system timer frequency remains steady over the lifetime of the program. See *Kernel Timing* for more information.

The CPU Frequency Scaling subsystem runs as a handler function to a `k_timer`, which means it runs in interrupt context (IRQ). The SoC P-state driver must ensure that its implementation of `cpu_freq_pstate_set()` is IRQ context safe. If a P-state transition cannot be completed reasonably in an IRQ context, it is recommended that the P-state driver of the SoC implements its task as a workqueue item.

4.7.3 Power off

System power off

Related code samples

- Bluetooth Low Energy Power Management on STM32WBPerform Bluetooth LE operations with Zephyr power management enabled on STM32WB.
- GPIO as wake-up pinUse a GPIO pin as wake-up pin.
- NXP MCX system offUse system off (Deep Power Down) on NXP MCXA/MCXN SoCs and wake through the WUU wakeup controller.
- RT1060 System OffUse soft-off on MIMXRT1060-EVK.
- RT595 System OffUse soft-off on MIMXRT595-EVK.
- RT700 System OffUse system off on MIMXRT700-EVK.
- System OffUse deep sleep on Nordic platforms.
- System Off (MCXE31B standby)Use standby (the only low-power mode) on FRDM-MCXE31B and wake via an RTC alarm.

4.7.4 Resource Management

There are various situations where it's necessary to coordinate resource use at runtime among multiple clients. These include power rails, clocks, other peripherals, and binary device power management. The complexity of properly managing multiple consumers of a device in a multi-threaded system, especially when transitions may be asynchronous, suggests that a shared implementation is desirable.

Zephyr provides managers for several coordination policies. These managers are embedded into services that use them for specific functions.

- *On-Off Manager*

On-Off Manager

An on-off manager supports an arbitrary number of clients of a service which has a binary state. Example applications are power rails, clocks, and binary device power management.

The manager has the following properties:

- The stable states are off, on, and error. The service always begins in the off state. The service may also be in a transition to a given state.
- The core operations are request (add a dependency) and release (remove a dependency). Supporting operations are reset (to clear an error state) and cancel (to reclaim client data from an in-progress transition). The service manages the state based on calls to functions that initiate these operations.
- The service transitions from off to on when first client request is received.
- The service transitions from on to off when last client release is received.
- Each service configuration provides functions that implement the transition from off to on, from on to off, and optionally from an error state to off. Transitions must be invokable from both thread and interrupt context.
- The request and reset operations are asynchronous using *Asynchronous Notifications*. Both operations may be cancelled, but cancellation has no effect on the in-progress transition.
- Requests to turn on may be queued while a transition to off is in progress: when the service has turned off successfully it will be immediately turned on again (where context allows) and waiting clients notified when the start completes.

Requests are reference counted, but not tracked. That means clients are responsible for recording whether their requests were accepted, and for initiating a release only if they have previously successfully completed a request. Improper use of the API can cause an active client to be shut out, and the manager does not maintain a record of specific clients that have been granted a request.

Failures in executing a transition are recorded and inhibit further requests or releases until the manager is reset. Pending requests are notified (and cancelled) when errors are discovered.

Transition operation completion notifications are provided through *Asynchronous Notifications*.

Clients and other components interested in tracking all service state changes, including when a service begins turning off or enters an error state, can be informed of state transitions by registering a monitor with onoff_monitor_register(). Notification of changes are provided before issuing completion notifications associated with the new state.

Note

A generic API may be implemented by multiple drivers where the common case is asynchronous. The on-off client structure may be an appropriate solution for the generic API. Where drivers that can guarantee synchronous context-independent transitions a driver may use `onoff_sync_service` and its supporting API rather than `onoff_manager`, with only a small reduction in functionality (primarily no support for the monitor API).

On-Off Service APIs

Security

4.8 Security

This section covers services for protecting a system and its data: cryptographic primitives, authentication and identity, trusted execution, and token handling.

4.8.1 Cryptography

The crypto section contains information regarding the cryptographic primitives supported by the Zephyr kernel. Use the information to understand the principles behind the operation of the different algorithms and how they were implemented.

The following crypto libraries have been included:

PSA Crypto

Overview The PSA (Platform Security Architecture) Crypto API offers a portable programming interface for cryptographic operations and key storage across a wide range of hardware. It is designed to be user-friendly while still providing access to the low-level primitives essential for modern cryptography.

It is created and maintained by Arm. Arm developed the PSA as a comprehensive security framework to address the increasing security needs of connected devices.

In Zephyr, the PSA Crypto API is implemented using Mbed TLS, an open-source cryptographic library that provides the underlying cryptographic functions.

Design Goals The interface is suitable for a vast range of devices: from special-purpose cryptographic processors that process data with a built-in key, to constrained devices running custom application code, such as microcontrollers, and multi-application devices, such as servers. It follows the principle of cryptographic agility.

Algorithm Flexibility
The PSA Crypto API supports a wide range of cryptographic algorithms, allowing developers to switch between different cryptographic methods as needed. This flexibility is crucial for maintaining security as new algorithms emerge and existing ones become obsolete.

Key Management
The PSA Crypto API includes robust key management features that support the creation, storage, and use of cryptographic keys in a secure and flexible manner. It uses opaque key identifiers, which allows for easy key replacement and updates without exposing key material.

Implementation Independence
The PSA Crypto API abstracts the underlying cryptographic library, meaning that the specific implementation can be changed without affecting the application code. This abstraction supports cryptographic agility by enabling the use of different cryptographic libraries or hardware accelerators as needed.

Future-Proofing
By adhering to cryptographic agility, PSA Crypto ensures that applications can quickly adapt to new cryptographic standards and practices, enhancing long-term security and compliance.

Examples of Applications

Network Security (TLS)
The API provides all of the cryptographic primitives needed to establish TLS connections.

Secure Storage
The API provides all primitives related to storage encryption, block or file-based, with master encryption keys stored inside a key store.

Network Credentials
The API provides network credential management inside a key store, for example, for X.509-based authentication or pre-shared keys on enterprise networks.

Device Pairing
The API provides support for key agreement protocols that are often used for secure pairing of devices over wireless channels. For example, the pairing of an NFC token or a Bluetooth device might use key agreement protocols upon first use.

Secure Boot
The API provides primitives for use during firmware integrity and authenticity validation, during a secure or trusted boot process.

Attestation
The API provides primitives used in attestation activities. Attestation is the ability for a device to sign an array of bytes with a device private key and return the result to the caller. There are several use cases; ranging from attestation of the device state, to the ability to generate a key pair and prove that it has been generated inside a secure key store. The API provides access to the algorithms commonly used for attestation.

Factory Provisioning
Most IoT devices receive a unique identity during the factory provisioning process, or once they have been deployed to the field. This API provides the APIs necessary for populating a device with keys that represent that identity.

Usage considerations

Always check for errors
Most functions in the PSA Crypto API can return errors. All functions that can fail have the return type `psa_status_t`. A few functions cannot fail, and thus, return void or some other type.

If an error occurs, unless otherwise specified, the content of the output parameters is undefined and must not be used.

Some common causes of errors include:

- In implementations where the keys are stored and processed in a separate environment from the application, all functions that need to access the cryptography processing environment might fail due to an error in the communication between the two environments.
- If an algorithm is implemented with a hardware accelerator, which is logically separate from the application processor, the accelerator might fail, even when the application processor keeps running normally.
- Most functions might fail due to a lack of resources. However, some implementations guarantee that certain functions always have sufficient memory.
- All functions that access persistent keys might fail due to a storage failure.
- All functions that require randomness might fail due to a lack of entropy. Implementations are encouraged to seed the random generator with sufficient entropy during the execution of `psa_crypto_init()`. However, some security standards require periodic reseeding from a hardware random generator, which can fail.

Shared memory and concurrency
Some environments allow applications to be multithreaded, while others do not. In some environments, applications can share memory with a different security context. In environments with multithreaded applications or shared memory, applications must be written carefully to avoid data corruption or leakage. This specification requires the application to obey certain constraints.

In general, the PSA Crypto API allows either one writer or any number of simultaneous readers, on any given object. In other words, if two or more calls access the same object

concurrently, then the behavior is only well-defined if all the calls are only reading from the object and do not modify it. Read accesses include reading memory by input parameters and reading keystore content by using a key. For more details, refer to Concurrent calls

If an application shares memory with another security context, it can pass shared memory blocks as input buffers or output buffers, but not as non-buffer parameters. For more details, refer to Stability of parameters.

Cleaning up after use
To minimize impact if the system is compromised, it is recommended that applications wipe all sensitive data from memory when it is no longer used. That way, only data that is currently in use can be leaked, and past data is not compromised.

Wiping sensitive data includes:

- Clearing temporary buffers in the stack or on the heap.
- Aborting operations if they will not be finished.
- Destroying keys that are no longer used.

References

- PSA Crypto
- Mbed TLS

Random Number Generation

Overview The random API subsystem provides random number generation APIs in both cryptographically and non-cryptographically secure instances. Which random API to use is based on the cryptographic requirements of the random number. The non-cryptographic APIs will return random values much faster if non-cryptographic values are needed.

- Use `sys_rand_get()` and related functions for non-cryptographic purposes such as randomizing delays, shuffling data, or general-purpose randomization. These functions are faster but not suitable for security applications.
- Use `sys_csrand_get()` for cryptographic purposes such as generating encryption keys, nonces, initialization vectors, or any data that must be unpredictable to an attacker.

Warning

Never use the non-cryptographic random functions (`sys_rand_get()`, `sys_rand8_get()`, `sys_rand16_get()`, `sys_rand32_get()`, `sys_rand64_get()`) for security-sensitive operations. These functions do not provide cryptographically secure random numbers.

API Usage

Non-Cryptographic Random Numbers The following functions provide non-cryptographically secure random numbers:

- `sys_rand_get()` - Fill a buffer with random bytes
- `sys_rand8_get()` - Get a random 8-bit value
- `sys_rand16_get()` - Get a random 16-bit value
- `sys_rand32_get()` - Get a random 32-bit value
- `sys_rand64_get()` - Get a random 64-bit value

Example usage:

```c
#include <zephyr/random/random.h>

void example_random_usage(void)
{
    uint8_t buffer[16];
    uint32_t random_value;

    /* Fill buffer with random bytes */
    sys_rand_get(buffer, sizeof(buffer));

    /* Get a single random 32-bit value */
    random_value = sys_rand32_get();
}
```

Cryptographically Secure Random Numbers For cryptographic purposes, use `sys_csrand_get()`:

```c
#include <zephyr/random/random.h>

int generate_encryption_key(uint8_t *key, size_t key_len)
{
    int ret;

    ret = sys_csrand_get(key, key_len);
    if (ret != 0) {
        /* Handle error - entropy source may have failed */
        return ret;
    }

    return 0;
}
```

> **Note**
>
> `sys_csrand_get()` returns 0 on success, or `-EIO` if the entropy source fails. Always check the return value when generating cryptographic material.

Kconfig Options All configuration options can be found in subsys/random/Kconfig.

General Options

`CONFIG_TEST_RANDOM_GENERATOR`
: For testing, this option allows a non-random number generator to be used and permits random number APIs to return values that are not truly random.

> **Warning**
>
> This option is intended only for testing on platforms that lack hardware entropy support. The random numbers generated when this option is enabled are predictable and not suitable for any security-sensitive use.

Non-Cryptographic Random Generator Selection

`CONFIG_ENTROPY_DEVICE_RANDOM_GENERATOR`
: Uses the hardware entropy driver directly to generate random numbers. This provides

high-quality random numbers but may be slower than pseudo-random alternatives. Select this option when your hardware entropy source has sufficient performance for your application's needs.

`CONFIG_XOSHIRO_RANDOM_GENERATOR`
Uses the Xoshiro128++ pseudo-random number generator, seeded from the hardware entropy source. This is a fast, general-purpose PRNG with a 128-bit state that passes all standard randomness tests.

This is the recommended choice for most applications that need fast non-cryptographic random numbers.

`CONFIG_TIMER_RANDOM_GENERATOR`
Uses the system timer to generate pseudo-random numbers. This generator produces predictable values and is intended **only for testing** on platforms without hardware entropy support.

Requires `CONFIG_TEST_RANDOM_GENERATOR` to be enabled.

Cryptographically Secure Random Generator Selection The `CSPRNG_GENERATOR_CHOICE` kconfig choice selects the source for cryptographically secure random number generation.

To override the default in a board or SoC configuration file:

```
choice CSPRNG_GENERATOR_CHOICE
        default PSA_CSPRNG_GENERATOR
endchoice
```

Available generators:

`CONFIG_HARDWARE_DEVICE_CS_GENERATOR`
Uses the hardware entropy driver directly as the source for cryptographically secure random numbers. Select this when your hardware random number generator is certified or validated for cryptographic use.

`CONFIG_PSA_CSPRNG_GENERATOR`
enables a cryptographically secure random number generator that uses the PSA Crypto API. This CSPRNG implementation leverages the underlying PSA Crypto library's secure random number generation capabilities, which may use hardware entropy sources when available. The PSA CSPRNG provides cryptographically secure random numbers suitable for security-sensitive applications.

`CONFIG_TEST_CSPRNG_GENERATOR`
Routes calls to `sys_csrand_get()` through `sys_rand_get()`. This allows testing libraries that require CSPRNG on platforms without hardware entropy support.

Requires `CONFIG_TEST_RANDOM_GENERATOR` to be enabled.

> **Warning**
>
> This option provides **no cryptographic security**. Use only for testing.

Devicetree Configuration The random subsystem uses the `zephyr,entropy` chosen node to identify the hardware entropy device:

```
/ {
    chosen {
        zephyr,entropy = &rng;
    };
};
```

Ensure your board's devicetree correctly specifies this chosen node if your platform has a hardware random number generator.

API Reference
Random Function APIs

Related code samples

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- Microsoft Azure IoT Hub MQTTConnect to Azure IoT Hub and publish messages using MQTT.

Crypto APIs

Overview

API Reference

Generic API for crypto drivers
Crypto

Related code samples

- CryptoUse the crypto APIs to perform various encryption/decryption operations.

Ciphers API
Cipher

4.8.2 Authentication

The authentication subsystem provides security-token and credential-based authentication services for Zephyr applications.

FIDO2 Authenticator

Overview The FIDO2 authenticator subsystem implements the FIDO2 CTAP2 Specification (Client to Authenticator Protocol), allowing a Zephyr device to act as a hardware security key for passwordless authentication. The subsystem can be enabled with the `CONFIG_FIDO2` option.

FIDO2 security keys are used with the WebAuthn Specification web standard. A relying party (website or service) interacts with the authenticator through a client (browser or OS platform) to register and verify user credentials. The authenticator performs cryptographic operations using on-device keys that never leave the hardware.

The subsystem currently supports the following CTAP2 commands:

- `authenticatorMakeCredential`
- `authenticatorGetAssertion`
- `authenticatorGetInfo`
- `authenticatorClientPIN`
- `authenticatorGetNextAssertion`
- `authenticatorSelection`

Architecture The subsystem is organized into pluggable backend components, each selectable at build time via Kconfig:

Transport
Handles wire-protocol communication between the host and the authenticator. Transports are registered using the `FIDO2_TRANSPORT_DEFINE` macro and are iterated at startup. Available transports:

- **USB HID (CTAPHID)** — `CONFIG_FIDO2_TRANSPORT_USB_HID`
- **Bluetooth LE (CTAPBLE)** — `CONFIG_FIDO2_TRANSPORT_BLE`

User Presence (UP)
Confirms that a human is physically present. Backends are selected via `CONFIG_FIDO2_UP_BACKEND`:

- **Input device** — `CONFIG_FIDO2_UP_INPUT`
- **Always approve** — `CONFIG_FIDO2_UP_ALWAYS`
- **Custom** — `CONFIG_FIDO2_UP_CUSTOM` (application-provided)

Credential Storage
Persists discoverable (resident) credentials. Backends are selected via `CONFIG_FIDO2_STORAGE_BACKEND`:

- **Settings subsystem** — `CONFIG_FIDO2_STORAGE_SETTINGS`
- **None** — `CONFIG_FIDO2_STORAGE_NONE` (non-discoverable credentials only)

Attestation
Signs newly created credentials to prove their origin. Backends are selected via `CONFIG_FIDO2_ATTESTATION_BACKEND`:

- **Self attestation** — `CONFIG_FIDO2_ATTESTATION_SELF` (default)
- **None** — `CONFIG_FIDO2_ATTESTATION_NONE`
- **Custom** — `CONFIG_FIDO2_ATTESTATION_CUSTOM` (application-provided)

Usage To use the FIDO2 subsystem, include the main header:

```
#include <zephyr/authentication/fido2/fido2.h>
```

Basic Initialization At least one transport must be enabled for the authenticator to communicate with a host.

See fido2 for a complete initialization sequence.

Runtime State Monitoring The subsystem exposes a runtime state callback that applications can use to drive status indicators such as LEDs:

```
#include <zephyr/authentication/fido2/fido2.h>

static void on_state_change(enum fido2_runtime_state state, void *user_data)
{
    switch (state) {
    case FIDO2_RUNTIME_STATE_IDLE:
        /* LED off */
        break;
    case FIDO2_RUNTIME_STATE_WAITING_USER_PRESENCE:
        /* Blink LED */
        break;
```

(continues on next page)

(continued from previous page)

```
    case FIDO2_RUNTIME_STATE_PROCESSING:
        /* LED on solid */
        break;
    default:
        break;
    }
}

fido2_set_state_callback(on_state_change, NULL);
```

Extensions CTAP2 extensions are not implemented yet. The following Kconfig options exist for future implementation:

- **credProtect** — `CONFIG_FIDO2_EXT_CRED_PROTECT`
- **hmac-secret** — `CONFIG_FIDO2_EXT_HMAC_SECRET`
- **largeBlobKey** — `CONFIG_FIDO2_EXT_LARGE_BLOB_KEY`
- **credBlob** — `CONFIG_FIDO2_EXT_CRED_BLOB`
- **thirdPartyPayment** — `CONFIG_FIDO2_EXT_THIRD_PARTY_PAYMENT`

References

- FIDO2 CTAP2 Specification
- WebAuthn Specification
- FIDO Alliance

API Reference
FIDO2

Related code samples

- FIDO2 AuthenticatorImplement a FIDO2/CTAP2 hardware security key.

4.8.3 JSON Web Token (JWT)

Overview

JSON Web Tokens (JWT) are an open, industry standard (**RFC 7519**) method for representing claims securely between two parties. Although JWT is fairly flexible, this API is limited to creating the simplistic tokens needed to authenticate with the Google Core IoT infrastructure.

At a high level, a JWT is just a signed blob of JSON that a client can present as a token instead of sending, for example, their username/password every time.

Usage To use the JWT API, include the header file:

```
#include <zephyr/data/jwt.h>
```

Generating a JWT The JWT subsystem provides a lightweight, builder-based API for constructing JSON Web Tokens (JWT). It allows for the creation of tokens where the payload (claims) contains: expiration time, issued-at time, and audience. The token is then signed using a provided private key.

```
#include <zephyr/data/jwt.h>

struct jwt_builder builder;
char buffer[1024];
int ret;

/* Initialize the builder */
ret = jwt_init_builder(&builder, buffer, sizeof(buffer));
if (ret < 0) {
    /* Handle error */
}

/* Add payload: expiration, issued at, audience */
ret = jwt_add_payload(&builder, 1767221999, 1764605987, "project-id");
if (ret < 0) {
    /* Handle error */
}

/* Sign the token using a DER-encoded private key */
ret = jwt_sign(&builder, private_key_der, private_key_der_len);
if (ret < 0) {
    /* Handle error */
}

/*
 * buffer now contains the JWT; it can be passed to a third-party service that␣
↪will be able
 * to validate it against the public key associated with `private_key_der`.
 */
```

Configuration

Related configuration options:

- CONFIG_JWT
- CONFIG_JWT_.*

API Reference

JSON Web Token (JWT)

4.8.4 Trusted Firmware-M (TF-M)

Trusted Firmware-M Overview

Trusted Firmware-M (TF-M) is a reference implementation of the Platform Security Architecture (PSA) IoT Security Framework. It defines and implements an architecture and a set of software components that aim to address some of the main security concerns in IoT products.

Zephyr RTOS has been PSA Certified since Zephyr 2.0.0 with TF-M 1.0, and is currently integrated with TF-M 2.1.0.

What Does TF-M Offer? Through a set of secure services and by design, TF-M provides:

- Isolation of secure and non-secure resources
- Embedded-appropriate crypto

- Management of device secrets (keys, etc.)
- Firmware verification (and encryption)
- Protected off-chip data storage and retrieval
- Proof of device identity (device attestation)
- Audit logging

Build System Integration When using TF-M with a supported platform, TF-M will be automatically built and link in the background as part of the standard Zephyr build process. This build process makes a number of assumptions about how TF-M is being used, and has certain implications about what the Zephyr application image can and can not do:

- The secure processing environment (secure boot and TF-M) starts first
- Resource allocation for Zephyr relies on choices made in the secure image.

Architecture Overview A TF-M application will, generally, have the following three parts, from most to least trusted, left-to-right, with code execution happening in the same order (secure boot > secure image > ns image).

While the secure bootloader is optional, it is enabled by default, and secure boot is an important part of providing a secure solution:

```
+-------------------------------------+           +--------------+
| Secure Processing Environment (SPE) |           |     NSPE     |
| +----------++---------------------+ |           | +----------+ |
| |          ||                     | |           | |          | |
| | bl2.bin  ||  tfm_s_signed.bin   | |           | |zephyr.bin| |
| |          ||                     | | <- PSA -> | |          | |
| |  Secure  || Trusted Firmware-M  | |    APIs   | |  Zephyr  | |
| |   Boot   ||   (Secure Image)    | |           | |(NS Image)| |
| |          ||                     | |           | |          | |
| +----------++---------------------+ |           | +----------+ |
+-------------------------------------+           +--------------+
```

Communication between the (Zephyr) Non-Secure Processing Environment (NSPE) and the (TF-M) Secure Processing Environment image happens based on a set of PSA APIs, and normally makes use of an IPC mechanism that is included as part of the TF-M build, and implemented in Zephyr (see modules/trusted-firmware-m/interface).

Root of Trust (RoT) Architecture TF-M is based upon a **Root of Trust (RoT)** architecture. This allows for hierarchies of trust from most, to less, to least trusted, providing a sound foundation upon which to build or access trusted services and resources.

The benefit of this approach is that less trusted components are prevented from accessing or compromising more critical parts of the system, and error conditions in less trusted environments won't corrupt more trusted, isolated resources.

The following RoT hierarchy is defined for TF-M, from most to least trusted:

- PSA Root of Trust (**PRoT**), which consists of:
  - PSA Immutable Root of Trust: secure boot
  - PSA Updateable Root of Trust: most trusted secure services
- Application Root of Trust (**ARoT**): isolated secure services

The **PSA Immutable Root of Trust** is the most trusted piece of code in the system, to which subsequent Roots of Trust are anchored. In TF-M, this is the secure boot image, which verifies that

the secure and non-secure images are valid, have not been tampered with, and come from a reliable source. The secure bootloader also verifies new images during the firmware update process, thanks to the public signing key(s) built into it. As the name implies, this image is **immutable**.

The **PSA Updateable Root of Trust** implements the most trusted secure services and components in TF-M, such as the Secure Partition Manager (SPM), and shared secure services like PSA Crypto, Internal Trusted Storage (ITS), etc. Services in the PSA Updateable Root of Trust have access to other resources in the same Root of Trust.

The **Application Root of Trust** is a reduced-privilege area in the secure processing environment which, depending on the isolation level chosen when building TF-M, has limited access to the PRoT, or even other ARoT services at the highest isolation levels. Some standard services exist in the ARoT, such as Protected Storage (PS), and generally custom secure services that you implement should be placed in the ARoT, unless a compelling reason is present to place them in the PRoT.

These divisions are distinct from the **untrusted code**, which runs in the non-secure environment, and has the least privilege in the system. This is the Zephyr application image in this case.

Isolation Levels At present, there are three distinct **isolation levels** defined in TF-M, with increasingly rigid boundaries between regions. The isolation level used will depend on your security requirements, and the system resources available to you.

- **Isolation Level 1** is the lowest isolation level, and the only major boundary is between the secure and non-secure processing environment, usually by means of Arm TrustZone on Armv8-M processors. There is no distinction here between the PSA Updateable Root of Trust (PRoT) and the Application Root of Trust (ARoT). They execute at the same privilege level. This isolation level will lead to the smallest combined application images.
- **Isolation Level 2** builds upon level one by introducing a distinction between the PSA Updateable Root of Trust and the Application Root of Trust, where ARoT services have limited access to PRoT services, and can only communicate with them through public APIs exposed by the PRoT services. ARoT services, however, are not strictly isolated from one another.
- **Isolation Level 3** is the highest isolation level, and builds upon level 2 by isolating ARoT services from each other, so that each ARoT is essentially silo'ed from other services. This provides the highest level of isolation, but also comes at the cost of additional overhead and code duplication between services.

The current isolation level can be checked via `CONFIG_TFM_ISOLATION_LEVEL`.

Secure Boot The default secure bootloader in TF-M is based on MCUBoot, and is referred to as `BL2` in TF-M (for the second-stage bootloader, potentially after a HW-based bootloader on the secure MCU, etc.).

All images in TF-M are hashed and signed, with the hash and signature verified by MCUBoot during the firmware update process.

Some key features of MCUBoot as used in TF-M are:

- Public signing key(s) are baked into the bootloader
- S and NS images can be signed using different keys
- Firmware images can optionally be encrypted
- Client software is responsible for writing a new image to the secondary slot
- By default, uses static flash layout of two identically-sized memory regions
- Optional security counter for rollback protection

When dealing with (optionally) encrypted images:

- Only the payload is encrypted (header, TLVs are plain text)

- Hashing and signing are applied over the un-encrypted data
- Uses `AES-CTR-128` or `AES-CTR-256` for encryption
- Encryption key randomized every encryption cycle (via `imgtool`)
- The `AES-CTR` key is included in the image and can be encrypted using:
  - `RSA-OAEP`
  - `AES-KW` (128 or 256 bits depending on the `AES-CTR` key length)
  - `ECIES-P256`
  - `ECIES-X25519`

Key config properties to control secure boot in Zephyr are:

- `CONFIG_TFM_BL2` toggles the bootloader (default = `y`).
- `CONFIG_TFM_KEY_FILE_S` overrides the secure signing key.
- `CONFIG_TFM_KEY_FILE_NS` overrides the non-secure signing key.

Secure Processing Environment Once the secure bootloader has finished executing, a TF-M based secure image will begin execution in the **secure processing environment**. This is where our device will be initially configured, and any secure services will be initialised.

Note that the starting state of our device is controlled by the secure firmware, meaning that when the non-secure Zephyr application starts, peripherals may not be in the HW-default reset state. In case of doubts, be sure to consult the board support packages in TF-M, available in the `platform/ext/target/` folder of the TF-M module (which is in `modules/tee/tf-m/trusted-firmware-m/` within a default Zephyr west workspace.)

Secure Services As of TF-M 1.8.0, the following secure services are generally available (although vendor support may vary):

- Crypto
- Firmware Update (FWU)
- Initial Attestation
- Platform
- Secure Storage, which has two parts:
  - Internal Trusted Storage (ITS)
  - Protected Storage (PS)

A template also exists for creating your own custom services.

For full details on these services, and their exposed APIs, please consult the TF-M Documentation.

Key Management and Derivation Key and secret management is a critical part of any secure device. You need to ensure that key material is available to regions that require it, but not to anything else, and that it is stored securely in a way that makes it difficult to tamper with or maliciously access.

The **Internal Trusted Storage** service in TF-M is used by the **PSA Crypto** service (which itself makes use of mbedtls) to store keys, and ensure that private keys are only ever accessible to the secure processing environment. Crypto operations that make use of key material, such as when signing payloads or when decrypting sensitive data, all take place via key handles. At no point should the key material ever be exposed to the NS environment.

One exception is that private keys can be provisioned into the secure processing environment as a one-way operation, such as during a factory provisioning process, but even this should be

avoided where possible, and a request should be made to the SPE (via the PSA Crypto service) to generate a new private key itself, and the public key for that can be requested during provisioning and logged in the factory. This ensures the private key material is never exposed, or even known during the provisioning phase.

TF-M also makes extensive use of the **Hardware Unique Key (HUK)**, which every TF-M device must provide. This device-unique key is used by the **Protected Storage** service, for example, to encrypt information stored in external memory. For example, this ensures that the contents of flash memory can't be decrypted if they are removed and placed on a new device, since each device has its own unique HUK used while encrypting the memory contents the first time.

HUKs provide an additional advantage for developers, in that they can be used to derive new keys, and the **derived keys** don't need to be stored since they can be regenerated from the HUK at startup, using an additional salt/seed value (depending on the key derivation algorithm used). This removes the storage issue and a frequent attack vector. The HUK itself it usually highly protected in secure devices, and inaccessible directly by users.

`TFM_CRYPTO_ALG_HUK_DERIVATION` identifies the default key derivation algorithm used if a software implementation is used. The current default algorithm is `HKDF` (**RFC 5869**) with a SHA-256 hash. Other hardware implementations may be available on some platforms.

Non-Secure Processing Environment Zephyr is used for the NSPE, using a board that is supported by TF-M where the `CONFIG_BUILD_WITH_TFM` flag has been enabled.

Generally, you simply need to select the `*/ns` board target of a valid board (for example `mps2/an521/cpu0/ns`), which will configure your Zephyr application to run in the NSPE, correctly build and link it with the TF-M secure images, sign the secure and non-secure images, and merge the three binaries into a single `tfm_merged.hex` file. The *west flash* command will flash `tfm_merged.hex` by default in this configuration.

At present, Zephyr can not be configured to be used as the secure processing environment.

TF-M Requirements

The following are some of the boards that can be used with TF-M:

Board	NSPE board name
mps2_an521_board	`mps2/an521/cpu0/ns` (qemu supported)
mps3	• `mps3/corstone300/fvp/ns` (armfvp supported) • `mps3/corstone310/fvp/ns` (armfvp supported)
mps4	• `mps4/corstone315/fvp/ns` (armfvp supported) • `mps4/corstone320/fvp/ns` (armfvp supported)
bl5340_dvk	`bl5340_dvk/nrf5340/cpuapp/ns`
lpcxpresso55s69	`lpcxpresso55s69_ns`
nrf9160dk_nrf9160	`nrf9160dk/nrf9160/ns`
nrf5340dk	`nrf5340dk/nrf5340/cpuapp/ns`
b_u585i_iot02a	`b_u585i_iot02a/stm32u585xx/ns`
nucleo_l552ze_q	`nucleo_l552ze_q/stm32l552xx/ns`
stm32l562e_dk	`stm32l562e_dk/stm32l562xx/ns`
v2m_musca_b1	`v2m_musca_b1/musca_b1/ns`

To make sure TF-M is supported for a board in its output, check that `CONFIG_TRUSTED_EXECUTION_NONSECURE` is set to `y` in that board's default configuration.

Software Requirements The Python modules required when building TF-M binaries are listed in the TF-M repository under `tools/requirements.txt`.

You can install them via:

```
$ pip3 install -r "$(west list trusted-firmware-m -f '{abspath}')/
↪tools/requirements.txt"
```

They are used by TF-M's signing utility to prepare firmware images for validation by the bootloader.

Part of the process of generating binaries for QEMU and merging signed secure and non-secure binaries on certain platforms also requires the use of the `srec_cat` utility.

This can be installed on Linux via:

```
$ sudo apt-get install srecord
```

And on OS X via:

```
$ brew install srecord
```

For Windows-based systems, please make sure you have a copy of the utility available on your system path. See, for example: SRecord for Windows

TF-M Build System

When building a valid `_ns` board target, TF-M will be built in the background, and linked with the Zephyr non-secure application. No knowledge of TF-M's build system is required in most cases, and the following will build a TF-M and Zephyr image pair, and run it in qemu with no additional steps required:

```
$ west build -p auto -b mps2/an521/cpu0/ns␣
↪samples/tfm_integration/psa_protected_storage/ -t run
```

The outputs and certain key steps in this build process are described here, however, since you will need to understand and interact with the outputs, and deal with signing the secure and non-secure images before deploying them.

Images Created by the TF-M Build The TF-M build system creates the following executable files:

- tfm_s - TF-M secure firmware
- tfm_ns - TF-M non-secure app (only used by regression tests).
- bl2 - TF-M MCUboot, if enabled

For each of these, it creates .bin, .hex, .elf, and .axf files.

The TF-M build system also creates signed variants of tfm_s and tfm_ns, and a file which combines them:

- tfm_s_signed
- tfm_ns_signed
- tfm_s_ns_signed

For each of these, only .bin files are created.

The TF-M non-secure app is discarded in favor of Zephyr non-secure app except when running the TF-M regression test suite.

The Zephyr build system usually signs both tfm_s and the Zephyr non-secure app itself. See below for details.

The 'tfm' target contains properties for all these paths. For example, the following will resolve to `<path>/tfm_s.hex`:

```
$<TARGET_PROPERTY:tfm,TFM_S_HEX_FILE>
```

See the top level CMakeLists.txt file in the tfm module for an overview of all the properties.

Signing Images When `CONFIG_TFM_BL2` is set to `y`, TF-M uses a secure bootloader (BL2) and firmware images must be signed with a private key. The firmware image is validated by the bootloader during updates using the corresponding public key, which is stored inside the secure bootloader firmware image.

By default, `<tfm-dir>/bl2/ext/mcuboot/root-rsa-3072.pem` is used to sign secure images, and `<tfm-dir>/bl2/ext/mcuboot/root-rsa-3072_1.pem` is used to sign non-secure images. These default .pem keys can (and **should**) be overridden using the `CONFIG_TFM_KEY_FILE_S` and `CONFIG_TFM_KEY_FILE_NS` config flags.

To satisfy PSA Certified Level 1 requirements, **You MUST replace the default .pem file with a new key pair!**

To generate a new public/private key pair, run the following commands:

```
$ imgtool keygen -k root-rsa-3072_s.pem -t rsa-3072
$ imgtool keygen -k root-rsa-3072_ns.pem -t rsa-3072
```

You can then place the new .pem files in an alternate location, such as your Zephyr application folder, and reference them in the `prj.conf` file via the `CONFIG_TFM_KEY_FILE_S` and `CONFIG_TFM_KEY_FILE_NS` config flags.

> **Warning**
>
> Be sure to keep your private key file in a safe, reliable location! If you lose this key file, you will be unable to sign any future firmware images, and it will no longer be possible to update your devices in the field!

After the built-in signing script has run, it creates a `tfm_merged.hex` (and `tfm_merged.bin`) file that contains all three binaries: bl2, tfm_s, and the zephyr app. These files can then be flashed to your development board or run in QEMU.

Output Files Upon completion of the Zephyr TF-M build, the following output files exist:

Table 55: TF-M Output Files

Filename	Created From	Bootloader Flags	Usage
`tfm_s_signed.{hex/bin}`	TF-M Secure	Signed	OTA Upgrades (`CONFIG_TFM_MCUBOOT_IMAGE_NUMBER` == 2)
`zephyr_ns_signed.{hex/bin}`	Zephyr Nonsecure	Signed	OTA Upgrades (`CONFIG_TFM_MCUBOOT_IMAGE_NUMBER` == 2)
`tfm_s_zephyr_ns_sig{hex/bin}`	TF-M Secure, Zephyr Nonsecure	Signed	OTA Upgrades (`CONFIG_TFM_MCUBOOT_IMAGE_NUMBER` == 1)
`tfm_merged.{hex/bin}`	Bootloader, TF-M Secure, Zephyr Nonsecure	Signed, Confirmed	Production Programming, flashed by `west flash`

Custom CMake arguments When building a Zephyr application with TF-M it might be necessary to control the CMake arguments passed to the TF-M build.

Zephyr TF-M build offers several Kconfig options for controlling the build, but doesn't cover every CMake argument supported by the TF-M build system.

The `TFM_CMAKE_OPTIONS` property on the `zephyr_property_target` can be used to pass custom CMake arguments to the TF-M build system.

To pass the CMake argument `-DFOO=bar` to the TF-M build system, place the following CMake snippet in your CMakeLists.txt file.

```
set_property(TARGET zephyr_property_target
             APPEND PROPERTY TFM_CMAKE_OPTIONS
             -DFOO=bar
)
```

Note

The `TFM_CMAKE_OPTIONS` is a list so it is possible to append multiple options. Also CMake generator expressions are supported, such as `$<1:-DFOO=bar>`

Since `TFM_CMAKE_OPTIONS` is a list argument it will be expanded before it is passed to the TF-M build system. Options that have list arguments must therefore be properly escaped to avoid being expanded as a list.

```
set_property(TARGET zephyr_property_target
             APPEND PROPERTY TFM_CMAKE_OPTIONS
             -DFOO="bar\\\;baz"
)
```

Footprint and Memory Usage The build system offers targets to view and analyse RAM and ROM usage in generated images. The tools run on the final images and give information about size of symbols and code being used in both RAM and ROM. For more information on these tools look here: *Footprint and Memory Usage*

Use the `tfm_ram_report` to get the RAM report for TF-M secure firmware (tfm_s).

Using west:

```
# From the root of the zephyr repository
west build -b mps2/an521/cpu0/ns samples/hello_world
west build -t tfm_ram_report
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2/an521/cpu0/ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild tfm_ram_report
```

Use the `tfm_rom_report` to get the ROM report for TF-M secure firmware (tfm_s).

Using west:

```
# From the root of the zephyr repository
west build -b mps2/an521/cpu0/ns samples/hello_world
west build -t tfm_rom_report
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2/an521/cpu0/ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild tfm_rom_report
```

Use the `bl2_ram_report` to get the RAM report for TF-M MCUboot, if enabled.

Using west:

```
# From the root of the zephyr repository
west build -b mps2/an521/cpu0/ns samples/hello_world
west build -t bl2_ram_report
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2/an521/cpu0/ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild bl2_ram_report
```

Use the `bl2_rom_report` to get the ROM report for TF-M MCUboot, if enabled.

Using west:

```
# From the root of the zephyr repository
west build -b mps2/an521/cpu0/ns samples/hello_world
west build -t bl2_rom_report
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=mps2/an521/cpu0/ns samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild bl2_rom_report
```

Trusted Firmware-M Integration

The Trusted Firmware-M (TF-M) section contains information about the integration between TF-M and Zephyr RTOS. Use this information to help understand how to integrate TF-M with Zephyr for Cortex-M platforms and make use of its secure run-time services in Zephyr applications.

Board Definitions TF-M will be built for the secure processing environment along with Zephyr if the `CONFIG_BUILD_WITH_TFM` flag is set to `y`.

Generally, this value should never be set at the application level, however, and all config flags required for TF-M should be set in a board variant with the `_ns` suffix.

This board variant must define an appropriate flash, SRAM and peripheral configuration that takes into account the initialisation process in the secure processing environment. `CONFIG_TFM_BOARD` must also be set via modules/trusted-firmware-m/Kconfig.tfm to the board name that TF-M expects for this target, so that it knows which target to build for the secure processing environment.

Example: mps2/an521/cpu0/ns The mps2/an521/cpu0 board target is a dual-core Arm Cortex-M33 evaluation board that generates a secure Zephyr binary.

The optional mps2/an521/cpu0/ns board target, however, sets these additional kconfig flags that indicate that Zephyr should be built as a non-secure image, linked with TF-M as an external project, and optionally the secure bootloader:

- CONFIG_TRUSTED_EXECUTION_NONSECURE y
- CONFIG_ARM_TRUSTZONE_M y

Comparing the boards/arm/mps2/mps2_an521_cpu0.dts and boards/arm/mps2/mps2_an521_cpu0_ns.dts files, we can see that the ns version defines offsets in flash and SRAM memory, which leave the required space for TF-M and the secure bootloader:

```
reserved-memory {
            #address-cells = <1>;
            #size-cells = <1>;
            ranges;

            /* The memory regions defined below must match what the TF-M
             * project has defined for that board - a single image boot is
             * assumed. Please see the memory layout in:
             *␣
↪https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/platform/ext/target/mps2/an521
             */

            code: memory@100000 {
                    reg = <0x00100000 DT_SIZE_K(512)>;
            };

            ram: memory@28100000 {
                    reg = <0x28100000 DT_SIZE_M(1)>;
            };
    };
```

This reserves 1 MB of code memory and 1 MB of RAM for secure boot and TF-M, such that our non-secure Zephyr application code will start at 0x10000, with RAM at 0x28100000. 512 KB code memory is available for the NS zephyr image, along with 1 MB of RAM.

This matches the flash memory layout we see in flash_layout.h in TF-M:

```
* 0x0000_0000 BL2 - MCUBoot (0.5 MB)
* 0x0008_0000 Secure image     primary slot (0.5 MB)
* 0x0010_0000 Non-secure image primary slot (0.5 MB)
* 0x0018_0000 Secure image     secondary slot (0.5 MB)
* 0x0020_0000 Non-secure image secondary slot (0.5 MB)
* 0x0028_0000 Scratch area (0.5 MB)
* 0x0030_0000 Protected Storage Area (20 KB)
* 0x0030_5000 Internal Trusted Storage Area (16 KB)
* 0x0030_9000 NV counters area (4 KB)
* 0x0030_A000 Unused (984 KB)
```

mps2/an521 will be passed in to Tf-M as the board target, specified via CONFIG_TFM_BOARD.

Test Suites

TF-M includes two sets of test suites:

- tf-m-tests - Standard TF-M specific regression tests
- psa-arch-tests - Test suites for specific PSA APIs (secure storage, etc.)

These test suites can be run from Zephyr via an appropriate sample application in the samples/tfm_integration folder.

TF-M Regression Tests The regression test suite can be run via the tests/modules/tfm/regression test.

It tests various services and communication mechanisms across the NS/S boundary via the PSA APIs. They provide a useful sanity check for proper integration between the NS RTOS (Zephyr in this case) and the secure application (TF-M).

PSA Arch Tests The PSA Arch Test suite, available via tfm_psa_test, contains a number of test suites that can be used to validate that PSA API specifications are being followed by the secure application, TF-M being an implementation of the Platform Security Architecture (PSA).

Only one of these suites can be run at a time, with the available test suites described via `CONFIG_TFM_PSA_TEST_*` KConfig flags:

Purpose The output of these test suites is required to obtain PSA Certification for your specific board, RTOS (Zephyr here), and PSA implementation (TF-M in this case).

They also provide a useful test case to validate any PRs that make meaningful changes to TF-M, such as enabling a new TF-M board target, or making changes to the core TF-M module(s). They should generally be run as a coherence check before publishing a new PR for new board support, etc.

4.8.5 UUID

Overview

Universally Unique Identifiers (UUID), also known as Globally Unique IDentifiers (GUIDs), are 128-bit identifiers intended to guarantee uniqueness across space and time. They are defined by **RFC 9562**.

The UUID subsystem provides utility functions for the generation, parsing, and manipulation of UUIDs. It supports generating version 4 (random) and version 5 (name-based) UUIDs.

Usage To use the UUID API, include the header file:

```
#include <zephyr/sys/uuid.h>
```

Generating a UUIDv4 UUIDv4 is based on random numbers (hence requires an entropy generator), and can be generated using the `uuid_generate_v4()` function. `CONFIG_UUID_V4` must also be enabled.

```
struct uuid my_uuid;
char uuid_str[UUID_STR_LEN];

int ret = uuid_generate_v4(&my_uuid);

if (ret != 0) {
    printk("Failed to generate UUID v4 (err %d)\n", ret);
    return ret;
}

ret = uuid_to_string(&my_uuid, uuid_str);
if (ret != 0) {
    printk("Failed to convert UUID to string (err %d)\n", ret);
```

(continues on next page)

(continued from previous page)

```
    return ret;
}

 printk("UUID: %s\n", uuid_str);
```

Generating a UUIDv5 UUIDv5 is generated from a namespace UUID and a name (binary data), and can be generated using the `uuid_generate_v5()` function. `CONFIG_UUID_V5` must also be enabled.

UUIDv5 is deterministic: the same namespace and name will always result in the same UUID.

```
struct uuid ns_uuid;
struct uuid my_uuid;
char uuid_str[UUID_STR_LEN];

/* Well-known namespace ID for URLs (RFC 9562) */
uuid_from_string("6ba7b811-9dad-11d1-80b4-00c04fd430c8", &ns_uuid);

const char *name = "https://zephyrproject.org/";
int ret = uuid_generate_v5(&ns_uuid, name, strlen(name), &my_uuid);

if (ret != 0) {
    printk("Failed to generate UUID v5 (err %d)\n", ret);
    return ret;
}

ret = uuid_to_string(&my_uuid, uuid_str);
if (ret != 0) {
    printk("Failed to convert UUID to string (err %d)\n", ret);
    return ret;
}

/* outputs: "UUID: abdb72b5-b342-5882-925b-4ce0e3cb4790" */
printk("UUID: %s\n", uuid_str);
```

Configuration

Related configuration options:

- `CONFIG_UUID`
- `CONFIG_UUID_V4`
- `CONFIG_UUID_V5`
- `CONFIG_UUID_BASE64`

API Reference

UUID

Device Management

4.9 Device Management

This section covers services for managing a device throughout its lifecycle, such as firmware updates and querying metadata embedded in the firmware image.

4.9.1 MCUmgr

Overview

The management subsystem allows remote management of Zephyr-enabled devices. The following management operations are available:

- *OS management*
- *Image management*
- *Statistics management*
- *Settings (config) management*
- *File system management*
- *Shell management*
- *Enumeration management*
- *Transport management*
- *Zephyr basic management*

The management subsystem is based on the Simple Management Protocol (SMP) provided by MCUmgr, an open source project that provides a management subsystem that is portable across multiple real-time operating systems.

The management subsystem is located in subsys/mgmt/ inside of the Zephyr tree.

Additionally, there is a sample server that provides management functionality over Bluetooth LE and serial.

Supported transports

MCUmgr supports a wide range of transports in-tree of Zephyr, additional transports can be added and submitted upstream or used out-of-tree in downstream projects without needing to fork the Zephyr repository.

The following transports are supported in Zephyr:

Name	Kconfigs	Extra details	Mode support		*Transport group management support*	
			Serv	Clier	Incoming	Outgoing
UART	`CONFIG_MCUMGR_TRANSPORT_UART`	*UART SMP over console encoding*	✓		✓	✓
Raw UART	`CONFIG_MCUMGR_TRANSPORT_RAW_UART`		✓		✓	✓
Shell	`CONFIG_MCUMGR_TRANSPORT_SHELL`	*UART SMP over console encoding*	✓		✓	×
Bluetooth	`CONFIG_MCUMGR_TRANSPORT_BT`	*Bluetooth Low Energy*	✓		✓	✓
UDP (IPv4)	`CONFIG_MCUMGR_TRANSPORT_UDP` and `CONFIG_MCUMGR_TRANSPORT_UDP_IPV4`	Optional DTLS support	✓		×	×
UDP (IPv6)	`CONFIG_MCUMGR_TRANSPORT_UDP` and `CONFIG_MCUMGR_TRANSPORT_UDP_IPV6`	Optional DTLS support	✓		×	×
LoRaWAN	`CONFIG_MCUMGR_TRANSPORT_LORAWAN`		✓		×	×
SPI	`CONFIG_MCUMGR_TRANSPORT_SPI`		✓		×	×
The following are designed for usage in tests:						
Dummy	`CONFIG_MCUMGR_TRANSPORT_DUMMY`	Virtual	✓		✓	✓
Raw Dummy	`CONFIG_MCUMGR_TRANSPORT_RAW_DUMMY`	*UART SMP over console encoding*, virtual	✓		✓	✓

Tools/libraries

There are various tools and libraries available which enable usage of MCUmgr functionality on a device which are listed below. Note that these tools are not part of or related to the Zephyr project.

Table 56: Tools and Libraries for MCUmgr

Name	OS support	Transports	Groups									Type	Language
			OS	IMG	Stat	Settings	FS	Shel	Enur	Transport	Zephyr		
AuTerm	Windows, Linux, macOS	Serial, Serial (raw), Bluetooth, UDP, LoRaWAN	✓	✓	✓	✓	✓	✓	✓	×	✓	App	C++ (Qt)
mcumgr-client	Windows, Linux, macOS	Serial	×	✓	×	×	×	×	×	×	×	App	Rust
mcumgr-web	Windows, Linux, macOS	Bluetooth	×	✓	×	×	×	×	×	×	×	Web (chrome only)	Javascript
mcumgr-mac	macOS	Bluetooth	✓	✓	×	×	×	×	×	×	×	App	Rust
nRF Connect Device Manager: Android and iOS	iOS, Android	Bluetooth	✓	✓	✓	✓	✓	✓	×	×	✓	Library, App	Java, Kotlin, Swift
smp	Windows, Linux, macOS, iOS, Android	N/A	✓	✓	✓	✓	✓	✓	✓	×	✓	Library	Python
smpclient	Windows, Linux, macOS	Serial, Bluetooth, UDP	✓	✓	✓	✓	✓	✓	✓	×	✓	Library	Python
smpmgr	Windows, Linux, macOS	Serial, Bluetooth, UDP	✓	✓	✓	×	✓	✓	✓	×	×	App	Python
mcumgr-toolkit (CLI: mcumgrctl)	Windows, Linux, macOS	Serial, Bluetooth, UDP	✓	✓	✓	✓	✓	✓	✓	×	✓	Library, App	Rust, Python
Zephyr MCUmgr client (in-tree)	Linux, Zephyr	Serial, Serial (raw)	✓	✓	×	×	×	×	×	×	×	Library	C

Note that a tick for a particular group indicates basic support for that group in the code, it is possible that not all commands/features of a group are supported by the implementation.

J-Link Virtual MSD Interaction Note

On boards where a J-Link OB is present which has both CDC and MSC (virtual Mass Storage Device, also known as drag-and-drop) support, the MSD functionality can prevent MCUmgr commands over the CDC UART port from working due to how USB endpoints are configured in the J-Link firmware (for example on the nrf52840dk board) because of limiting the maximum packet size (most likely to occur when using image management commands for updating firmware). This issue can be resolved by disabling MSD functionality on the J-Link device, follow the instructions on nordic_segger_msd to disable MSD support.

Bootloader Integration

The *Device Firmware Upgrade* subsystem integrates the management subsystem with the bootloader, providing the ability to send and upgrade a Zephyr image to a device.

Currently only the MCUboot bootloader is supported. See *MCUboot* for more information.

Discord channel

Developers welcome!

- Discord mcumgr channel: https://discord.com/invite/Ck7jw53nU2

API Reference

Management

4.9.2 MCUmgr handlers

Overview

MCUmgr functions by having group handlers which identify a group of functions relating to a specific management area, which is addressed with a 16-bit identification value, `mcumgr_group_t` contains the management groups available in Zephyr with their corresponding group ID values. The group ID is included in SMP headers to identify which group a command belongs to, there is also an 8-bit command ID which identifies the function of that group to execute - see *SMP Protocol Specification* for details on the SMP protocol and header. There can only be one registered group per unique ID.

Implementation

MCUmgr handlers can be added externally by application code or by module code, they do not have to reside in the upstream Zephyr tree to be usable. The first step to creating a handler is to create the folder structure for it, the typical Zephyr MCUmgr group layout is as follows:

```
<dir>/grp/<grp_name>_mgmt/
├── CMakeLists.txt
├── Kconfig
├── include
├─── <grp_name>_mgmt.h
├─── <grp_name>_mgmt_callbacks.h
├── src
└─── <grp_name>_mgmt.c
```

Note that the header files in upstream Zephyr MCUmgr handlers reside in the `zephyr/include/zephyr/mgmt/mcumgr/grp/<grp_name>_mgmt` directory to allow the files to be globally included by applications.

Initial header <grp_name>_mgmt.h The purpose of the header file is to provide defines which can be used by the MCUmgr handler itself and application code, e.g. to reference the command IDs for executing functions. An example file would look similar to:

```
/*
 * Copyright (c) 2023 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#ifndef H_EXAMPLE_MGMT_
#define H_EXAMPLE_MGMT_

#ifdef __cplusplus
extern "C" {
#endif

/**
 * Group ID for example management group.
 */
#define MGMT_GROUP_ID_EXAMPLE  MGMT_GROUP_ID_PERUSER

/**
 * Command IDs for example management group.
 */
#define EXAMPLE_MGMT_ID_TEST   0
#define EXAMPLE_MGMT_ID_OTHER  1

/**
 * Command result codes for example management group.
 */
enum example_mgmt_err_code_t {
        /** No error, this is implied if there is no ret value in the response */
        EXAMPLE_MGMT_ERR_OK = 0,

        /** Unknown error occurred. */
        EXAMPLE_MGMT_ERR_UNKNOWN,

        /** The provided value is not wanted at this time. */
        EXAMPLE_MGMT_ERR_NOT_WANTED,

        /** The provided value was rejected by a hook. */
        EXAMPLE_MGMT_ERR_REJECTED_BY_HOOK,
};

#ifdef __cplusplus
}
#endif

#endif
```

This provides the defines for 2 command `test` and `other` and sets up the SMP version 2 error responses (which have unique error codes per group as opposed to the legacy SMP version 1 error responses that return a `mcumgr_err_t` - there should always be an OK error code with the value 0 and an unknown error code with the value 1. The above example then adds an error code

of `not wanted` with value 2. In addition, the group ID is set to be `MGMT_GROUP_ID_PERUSER`, which is the start group ID for user-defined groups, note that group IDs need to be unique so other custom groups should use different values, a central index header file (as upstream Zephyr has) can be used to distribute group IDs more easily.

Initial header <grp_name>_mgmt_callbacks.h The purpose of the header file is to provide defines which can be used by the MCUmgr handler itself and application code, e.g. to reference the command IDs for executing functions. An example file would look similar to:

```
/*
 * Copyright (c) 2023 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#ifndef H_MCUMGR_EXAMPLE_MGMT_CALLBACKS_
#define H_MCUMGR_EXAMPLE_MGMT_CALLBACKS_
#include <stdint.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>

#ifdef __cplusplus
extern "C" {
#endif

/* This is the event ID for the example group */
#define MGMT_EVT_GRP_EXAMPLE MGMT_EVT_GRP_USER_CUSTOM_START

/* MGMT event opcodes for example management group */
enum example_mgmt_group_events {
        /* Callback when the other command is received, data is␣
↪example_mgmt_other_data */
        MGMT_EVT_OP_EXAMPLE_OTHER = MGMT_DEF_EVT_OP_ID(MGMT_EVT_GRP_EXAMPLE, 0),

        /* Used to enable all smp_group events */
        MGMT_EVT_OP_EXAMPLE_MGMT_ALL  = MGMT_DEF_EVT_OP_ALL(MGMT_EVT_GRP_EXAMPLE),
};

/* Structure provided in the #MGMT_EVT_OP_EXAMPLE_OTHER notification callback */
struct example_mgmt_other_data {
        /* Contains the user supplied value */
        uint32_t user_value;
};

#ifdef __cplusplus
}
#endif

#endif
```

This sets up a single event which application (or module) code can register for to receive a callback when the function handler is executed, which allows the flow of the handler to be changed (i.e. to return an error instead of continuing). The event group ID is set to `MGMT_EVT_GRP_USER_CUSTOM_START`, which is the start event ID for user-defined groups, note that event IDs need to be unique so other custom groups should use different values, a central index header file (as upstream Zephyr has) can be used to distribute event IDs more easily.

Initial source <grp_name>_mgmt.c The purpose of this source file is to handle the incoming MCUmgr commands, provide responses, and register the transport with MCUmgr so that commands will be sent to it. An example file would look similar to:

```c
/*
 * Copyright (c) 2023 Nordic Semiconductor ASA
 *
 * SPDX-License-Identifier: Apache-2.0
 */
#include <zephyr/kernel.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/smp/smp.h>
#include <zephyr/mgmt/mcumgr/mgmt/handlers.h>
/* The below should be updated with the real name of the file */
#include <example_mgmt.h>
#include <zephyr/logging/log.h>
#include <assert.h>
#include <limits.h>
#include <string.h>
#include <stdio.h>
#include <zcbor_common.h>
#include <zcbor_decode.h>
#include <zcbor_encode.h>
#include <mgmt/mcumgr/util/zcbor_bulk.h>

#if defined(CONFIG_MCUMGR_MGMT_NOTIFICATION_HOOKS)
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>
#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
/* The below should be updated with the real name of the file */
#include <example_mgmt_callbacks.h>
#endif
#endif

LOG_MODULE_REGISTER(mcumgr_example_grp, CONFIG_MCUMGR_GRP_EXAMPLE_LOG_LEVEL);
/* Example function with "read" command support, requires both parameters are␣
↪supplied */
static int example_mgmt_test(struct smp_streamer *ctxt)
{
        uint32_t uint_value = 0;
        zcbor_state_t *zse = ctxt->writer->zs;
        zcbor_state_t *zsd = ctxt->reader->zs;
        bool ok;
        struct zcbor_string string_value = { 0 };
        size_t decoded;
        struct zcbor_map_decode_key_val example_test_decode[] = {
                ZCBOR_MAP_DECODE_KEY_DECODER("uint_key", zcbor_uint32_decode, &
↪uint_value),
                ZCBOR_MAP_DECODE_KEY_DECODER("string_key", zcbor_tstr_decode, &
↪string_value),
        };

        LOG_DBG("Example test function called");

        ok = zcbor_map_decode_bulk(zsd, example_test_decode,␣
↪ARRAY_SIZE(example_test_decode),
                                   &decoded) == 0;
        /* Check that both parameters were supplied and that the value of
↪"string_key" is not
         * empty
         */
        if (!ok || string_value.len == 0 || !zcbor_map_decode_bulk_key_found(
```

(continues on next page)

(continued from previous page)

```
                                example_test_decode, ARRAY_SIZE(example_test_decode), "uint_key")) {
                        return MGMT_ERR_EINVAL;
        }

        /* If the value of "uint_key" is over 50, return an error of "not wanted" */
        if (uint_value > 50) {
                        ok = smp_add_cmd_err(zse, MGMT_GROUP_ID_EXAMPLE, EXAMPLE_MGMT_ERR_NOT_WANTED);
                        goto end;
        }

        /* Otherwise, return an integer value of 4691 */
        ok = zcbor_tstr_put_lit(zse, "return_int") &&
             zcbor_int32_put(zse, 4691);

end:
        /* If "ok" is false, then there was an error processing the output cbor message, which
         * likely indicates a lack of available memory
         */
        return (ok ? MGMT_ERR_EOK : MGMT_ERR_EMSGSIZE);
}

/* Example function with "write" command support */
static int example_mgmt_other(struct smp_streamer *ctxt)
{
        uint32_t user_value = 0;
        zcbor_state_t *zse = ctxt->writer->zs;
        zcbor_state_t *zsd = ctxt->reader->zs;
        bool ok;
        size_t decoded;
        struct zcbor_map_decode_key_val example_other_decode[] = {
                        ZCBOR_MAP_DECODE_KEY_DECODER("user_value", zcbor_uint32_decode, &user_value),
        };

#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
        struct example_mgmt_other_data other_data;
        enum mgmt_cb_return status;
        int32_t err_rc;
        uint16_t err_group;
#endif

        LOG_DBG("Example other function called");

        ok = zcbor_map_decode_bulk(zsd, example_other_decode, ARRAY_SIZE(example_other_decode),
                                        &decoded) == 0;

        /* The supplied value is optional, therefore do not return an error if it was not
         * provided
         */
        if (!ok) {
```

(continues on next page)

(continued from previous page)

```
                return MGMT_ERR_EINVAL;
        }

#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
        /* Send request to application to check what to do */
        other_data.user_value = user_value;
        status = mgmt_callback_notify(MGMT_EVT_OP_EXAMPLE_OTHER, &other_data, sizeof(other_data),
                                      &err_rc, &err_group);
        if (status != MGMT_CB_OK) {
                /* If a callback returned an RC error, exit out, if it returned a group error
                 * code, add the error code to the response and return to the calling function to
                 * have it sent back to the client
                 */
                if (status == MGMT_CB_ERROR_RC) {
                        return err_rc;
                }

                ok = smp_add_cmd_err(zse, err_group, (uint16_t)err_rc);
                goto end;
        }
#endif
        /* Return some dummy data to the client */
        ok = zcbor_tstr_put_lit(zse, "return_string") &&
             zcbor_tstr_put_lit(zse, "some dummy data!");

#if defined(CONFIG_MCUMGR_GRP_EXAMPLE_OTHER_HOOK)
end:
#endif
        /* If "ok" is false, then there was an error processing the output cbor message, which
         * likely indicates a lack of available memory
         */
        return (ok ? MGMT_ERR_EOK : MGMT_ERR_EMSGSIZE);
}

#ifdef CONFIG_MCUMGR_SMP_SUPPORT_ORIGINAL_PROTOCOL
/* This is a lookup function that converts from SMP version 2 group error codes to legacy
 * MCUmgr error codes, it is only included if support for the original protocol is enabled.
 * Note that in SMP version 2, MCUmgr error codes can still be returned, but are to be used
 * only for general SMP/MCUmgr errors. The success/OK error code is not used in translation
 * functions as it is automatically handled by the base SMP code.
 */
static int example_mgmt_translate_error_code(uint16_t err)
{
        int rc;

        switch (err) {
        case EXAMPLE_MGMT_ERR_NOT_WANTED:
                rc = MGMT_ERR_ENOENT;
```

(continues on next page)

(continued from previous page)

```
                break;

        case EXAMPLE_MGMT_ERR_REJECTED_BY_HOOK:
                rc = MGMT_ERR_EBADSTATE;
                break;

        case EXAMPLE_MGMT_ERR_UNKNOWN:
        default:
                rc = MGMT_ERR_EUNKNOWN;
        }

        return rc;
}
#endif

static const struct mgmt_handler example_mgmt_handlers[] = {
        [EXAMPLE_MGMT_ID_TEST] = {
                .mh_read = example_mgmt_test,
                .mh_write = NULL,
        },
        [EXAMPLE_MGMT_ID_OTHER] = {
                .mh_read = NULL,
                .mh_write = example_mgmt_other,
        },
};

static struct mgmt_group example_mgmt_group = {
        .mg_handlers = example_mgmt_handlers,
        .mg_handlers_count = ARRAY_SIZE(example_mgmt_handlers),
        .mg_group_id = MGMT_GROUP_ID_EXAMPLE,
#ifdef CONFIG_MCUMGR_SMP_SUPPORT_ORIGINAL_PROTOCOL
        .mg_translate_error = example_mgmt_translate_error_code,
#endif
};

static void example_mgmt_register_group(void)
{
        /* This function is called during system init before main() is invoked, if the
         * handler needs to set anything up before it can be used, it should do it here.
         * This register the group so that clients can call the function handlers.
         */
        mgmt_register_group(&example_mgmt_group);
}

MCUMGR_HANDLER_DEFINE(example_mgmt, example_mgmt_register_group);
```

The above code creates 2 function handlers, `test` which supports read requests and takes 2 required parameters, and `other` which supports write requests and takes 1 optional parameter, this function handler has an optional notification callback feature that allows other parts of the code to listen for the event and take any required actions that are necessary or prevent further execution of the function by returning an error, further details on MCUmgr callback functionality can be found on *MCUmgr Callbacks*.

Note that other code referencing callbacks for custom MCUmgr handlers needs to include both the base Zephyr callback include file and the custom handler callback file, only in-tree Zephyr

handler headers are included when including the upstream Zephyr callback header file.

Initial Kconfig The purpose of the Kconfig file is to provide options which users can enable or change relating to the functionality of the handler being implemented. An example file would look similar to:

```
# Copyright Nordic Semiconductor ASA 2023. All rights reserved.
# SPDX-License-Identifier: Apache-2.0
# The Kconfig file is dedicated to example management group of
# of MCUmgr subsystem and provides Kconfig options to configure
# group commands behaviour and other aspects.
#
# Options defined in this file should be prefixed:
#  MCUMGR_GRP_EXAMPLE_ -- general group options;
#
# When adding Kconfig options, that control the same feature,
# try to group them together by the same stem after prefix.
if MCUMGR

menuconfig MCUMGR_GRP_EXAMPLE_APP
        bool "MCUmgr handlers for example management (app)"
        select MCUMGR_SMP_CBOR_MIN_DECODING_LEVEL_2
        default y
        help
          Enables MCUmgr handlers for example management. This demonstrates the
          file at application-level.

if MCUMGR_GRP_EXAMPLE_APP
config MCUMGR_GRP_EXAMPLE_OTHER_HOOK
        bool "Other hook"
        depends on MCUMGR_MGMT_NOTIFICATION_HOOKS
        help
          Allows applications to receive callback when the "other" example
          management function is called

module = MCUMGR_GRP_EXAMPLE
module-str = mcumgr_grp_example
source "subsys/logging/Kconfig.template.log_config"

endif

endif

source "Kconfig.zephyr"
```

Initial CMakeLists.txt The CMakeLists.txt file is used by the build system to setup files to compile, include directories to add and specify options that can be changed. A basic file only need to include the source files if the Kconfig options are enabled. An example file would look similar to:

Zephyr module

```
# Copyright (c) 2023 Nordic Semiconductor ASA
# SPDX-License-Identifier: Apache-2.0

if(CONFIG_MCUMGR_GRP_EXAMPLE_MODULE)
  zephyr_library()
```

(continues on next page)

(continued from previous page)

```
  # The below should be updated with the real name of the file
  zephyr_library_sources(src/example_mgmt.c)
  zephyr_include_directories(include)
endif()
```

Application

```
if(CONFIG_MCUMGR_GRP_EXAMPLE_APP)
  target_sources(app PRIVATE example_as_module/src/example_mgmt.c)
  zephyr_include_directories(example_as_module/include)
endif()
```

Including from application

Application-specific MCUmgr handlers can be added by creating/editing application build files. Example modifications are shown below.

Example CMakeLists.txt The application `CMakeLists.txt` file can load the CMake file for the example MCUmgr handler by adding the following:

```
add_subdirectory(mcumgr/grp/<grp_name>)
```

Example Kconfig The application Kconfig file can include the Kconfig file for the example MCUmgr handler by adding the following to the `Kconfig` file in the application directory (or creating it if it does not exist):

```
rsource "mcumgr/grp/<grp_name>/Kconfig"

# Include Zephyr's Kconfig
source "Kconfig.zephyr"
```

Including from Zephyr Module

Zephyr *Modules (External projects)* can be used to add custom MCUmgr handlers to multiple different applications without needing to duplicate the code in each application's source tree, see *Module yaml file description* for details on how to set up the module files. Example files are shown below.

Example zephyr/module.yml This is an example file which can be used to load the Kconfig and CMake files from the root of the module directory, and would be placed at `zephyr/module.yml`:

```
build:
  kconfig: Kconfig
  cmake: .
```

Example CMakeLists.txt This is an example CMakeLists.txt file which loads the CMake file for the example MCUmgr handler, and would be placed at `CMakeLists.txt`:

```
add_subdirectory(mcumgr/grp/<grp_name>)
```

Example Kconfig This is an example Kconfig file which loads the Kconfig file for the example MCUmgr handler, and would be placed at `Kconfig`:

```
rsource "mcumgr/grp/<grp_name>/Kconfig"
```

Demonstration handler

There is a demonstration project which includes configuration for both application and zephyr module-MCUmgr handlers which can be used as a basis for created your own in tests/subsys/mgmt/mcumgr/handler_demo/.

4.9.3 MCUmgr Callbacks

Overview

MCUmgr has a customisable callback/notification system that allows application (and module) code to receive callbacks for MCUmgr events that they are interested in and react to them or return a status code to the calling function that provides control over if the action should be allowed or not. An example of this is with the fs_mgmt group, whereby file access can be gated, the callback allows the application to inspect the request path and allow or deny access to said file, or it can rewrite the provided path to a different path for transparent file redirection support.

Implementation

Enabling The base callback/notification system can be enabled using `CONFIG_MCUMGR_MGMT_NOTIFICATION_HOOKS` which will compile the registration and notification system into the code. This will not provide any callbacks by default as the callbacks that are supported by a build must also be selected by enabling the Kconfig's for the required callbacks (see *Events* for further details). A callback function with the `mgmt_cb` type definition can then be declared and registered by calling `mgmt_callback_register()` for the desired event inside of a `mgmt_callback` structure. Handlers are called in the order that they were registered.

With the system enabled, a basic handler can be set up and defined in application code as per:

```
#include <zephyr/kernel.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>

struct mgmt_callback my_callback;

enum mgmt_cb_return my_function(uint32_t event, enum mgmt_cb_return prev_status,
                                int32_t *rc, uint16_t *group, bool *abort_more,
                                void *data, size_t data_size)
{
    if (event == MGMT_EVT_OP_CMD_DONE) {
        /* This is the event we registered for */
    }

    /* Return OK status code to continue with acceptance to underlying handler */
    return MGMT_CB_OK;
}

int main()
{
    my_callback.callback = my_function;
    my_callback.event_id = MGMT_EVT_OP_CMD_DONE;
    mgmt_callback_register(&my_callback);
}
```

This code registers a handler for the `MGMT_EVT_OP_CMD_DONE` event, which will be called after a

MCUmgr command has been processed and output generated, note that this requires that CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS be enabled to receive this callback.

Multiple callbacks can be setup to use a single function as a common callback, and many different functions can be used for each event by registering each group once, or all notifications for a whole group can be enabled by using one of the MGMT_EVT_OP_*_ALL events, alternatively a handler can setup for every notification by using MGMT_EVT_OP_ALL. When setting up handlers, events can be combined that are in the same group only, for example 5 img_mgmt callbacks can be setup with a single registration call, but to also setup a callback for an os_mgmt callback, this must be done as a separate registration. Group IDs are numerical increments, event IDs are bitmask values, hence the restriction.

As an example, the following registration is allowed, which will register for 3 SMP events with a single callback function in a single registration:

```
my_callback.callback = my_function;
my_callback.event_id = (MGMT_EVT_OP_CMD_RECV |
                        MGMT_EVT_OP_CMD_STATUS |
                        MGMT_EVT_OP_CMD_DONE);
mgmt_callback_register(&my_callback);
```

The following code is not allowed, and will cause undefined operation, because it mixes the IMG management group with the OS management group whereby the group is **not** a bitmask value, only the event is:

```
my_callback.callback = my_function;
my_callback.event_id = (MGMT_EVT_OP_IMG_MGMT_DFU_STARTED |
                        MGMT_EVT_OP_OS_MGMT_RESET);
mgmt_callback_register(&my_callback);
```

Events Events can be selected by enabling their corresponding Kconfig option:

- **CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS**
  MCUmgr command status (MGMT_EVT_OP_CMD_RECV, MGMT_EVT_OP_CMD_STATUS, MGMT_EVT_OP_CMD_DONE)
- **CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK**
  fs_mgmt file access (MGMT_EVT_OP_FS_MGMT_FILE_ACCESS)
- **CONFIG_MCUMGR_GRP_IMG_UPLOAD_CHECK_HOOK**
  img_mgmt upload check (MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK)
- **CONFIG_MCUMGR_GRP_IMG_STATUS_HOOKS**
  img_mgmt upload status (MGMT_EVT_OP_IMG_MGMT_DFU_STOPPED, MGMT_EVT_OP_IMG_MGMT_DFU_STARTED, MGMT_EVT_OP_IMG_MGMT_DFU_PENDING, MGMT_EVT_OP_IMG_MGMT_DFU_CONFIRMED)
- **CONFIG_MCUMGR_GRP_OS_RESET_HOOK**
  os_mgmt reset check (MGMT_EVT_OP_OS_MGMT_RESET)
- **CONFIG_MCUMGR_GRP_SETTINGS_ACCESS_HOOK**
  settings_mgmt access (MGMT_EVT_OP_SETTINGS_MGMT_ACCESS)

Actions Some callbacks expect a return status to either allow or disallow an operation, an example is the fs_mgmt access hook which allows for access to files to be allowed or denied. With these handlers, the first non-OK error code returned by a handler will be returned to the MCUmgr client.

An example of selectively denying file access:

```
#include <zephyr/kernel.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>
#include <string.h>

struct mgmt_callback my_callback;

enum mgmt_cb_return my_function(uint32_t event, enum mgmt_cb_return prev_status,
                                int32_t *rc, uint16_t *group, bool *abort_more,
                                void *data, size_t data_size)
{
    /* Only run this handler if a previous handler has not failed */
    if (event == MGMT_EVT_OP_FS_MGMT_FILE_ACCESS && prev_status == MGMT_CB_OK) {
        struct fs_mgmt_file_access *fs_data = (struct fs_mgmt_file_access *)data;

        /* Check if this is an upload and deny access if it is, otherwise check
         * the path and deny if is matches a name
         */
        if (fs_data->access == FS_MGMT_FILE_ACCESS_WRITE) {
            /* Return an access denied error code to the client and abort calling
             * further handlers
             */
            *abort_more = true;
            *rc = MGMT_ERR_EACCESSDENIED;

            return MGMT_CB_ERROR_RC;
        } else if (strcmp(fs_data->filename, "/lfs1/false_deny.txt") == 0) {
            /* Return a no entry error code to the client, call additional handlers
             * (which will have failed set to true)
             */
            *rc = MGMT_ERR_ENOENT;

            return MGMT_CB_ERROR_RC;
        }
    }

    /* Return OK status code to continue with acceptance to underlying handler */
    return MGMT_CB_OK;
}

int main()
{
    my_callback.callback = my_function;
    my_callback.event_id = MGMT_EVT_OP_FS_MGMT_FILE_ACCESS;
    mgmt_callback_register(&my_callback);
}
```

This code registers a handler for the `MGMT_EVT_OP_FS_MGMT_FILE_ACCESS` event, which will be called after a fs_mgmt file read/write command has been received to check if access to the file should be allowed or not, note that this requires that `CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK` be enabled to receive this callback. Two types of errors can be returned, the `rc` parameter can be set to an `mcumgr_err_t` error code and `MGMT_CB_ERROR_RC` can be returned, or a group error code (introduced with version 2 of the MCUmgr protocol) can be set by setting the `group` value to the group and `rc` value to the group error code and returning `MGMT_CB_ERROR_ERR`.

MCUmgr Command Callback Usage/Adding New Event Types To add a callback to a MCUmgr command, `mgmt_callback_notify()` can be called with the event ID and, optionally, a data struct

to pass to the callback (which can be modified by handlers). If no data needs to be passed back, NULL can be used instead, and size of the data set to 0.

An example MCUmgr command handler:

```
#include <zephyr/kernel.h>
#include <zcbor_common.h>
#include <zcbor_encode.h>
#include <zephyr/mgmt/mcumgr/smp/smp.h>
#include <zephyr/mgmt/mcumgr/mgmt/mgmt.h>
#include <zephyr/mgmt/mcumgr/mgmt/callbacks.h>

#define MGMT_EVT_GRP_USER_ONE MGMT_EVT_GRP_USER_CUSTOM_START

enum user_one_group_events {
    /** Callback on first post, data is test_struct. */
    MGMT_EVT_OP_USER_ONE_FIRST  = MGMT_DEF_EVT_OP_ID(MGMT_EVT_GRP_USER_ONE, 0),

    /** Callback on second post, data is test_struct. */
    MGMT_EVT_OP_USER_ONE_SECOND = MGMT_DEF_EVT_OP_ID(MGMT_EVT_GRP_USER_ONE, 1),

    /** Used to enable all user_one events. */
    MGMT_EVT_OP_USER_ONE_ALL    = MGMT_DEF_EVT_OP_ALL(MGMT_EVT_GRP_USER_ONE),
};

struct test_struct {
    uint8_t some_value;
};

static int test_command(struct mgmt_ctxt *ctxt)
{
    int rc;
    int err_rc;
    uint16_t err_group;
    zcbor_state_t *zse = ctxt->cnbe->zs;
    bool ok;
    struct test_struct test_data = {
        .some_value = 8,
    };

    rc = mgmt_callback_notify(MGMT_EVT_OP_USER_ONE_FIRST, &test_data,
                              sizeof(test_data), &err_rc, &err_group);

    if (rc != MGMT_CB_OK) {
        /* A handler returned a failure code */
        if (rc == MGMT_CB_ERROR_RC) {
            /* The failure code is the RC value */
            return err_rc;
        }

        /* The failure is a group and ID error value */
        ok = smp_add_cmd_err(zse, err_group, (uint16_t)err_rc);
        goto end;
    }

    /* All handlers returned success codes */
    ok = zcbor_tstr_put_lit(zse, "output_value") &&
         zcbor_int32_put(zse, 1234);
```

(continues on next page)

(continued from previous page)

```
end:
    rc = (ok ? MGMT_ERR_EOK : MGMT_ERR_EMSGSIZE);

    return rc;
}
```

If no response is required for the callback, the function call be called and casted to void.

Migration

If there is existing code using the previous callback system(s) in Zephyr 3.2 or earlier, then it will need to be migrated to the new system. To migrate code, the following callback registration functions will need to be migrated to register for callbacks using `mgmt_callback_register()` (note that `CONFIG_MCUMGR_MGMT_NOTIFICATION_HOOKS` will need to be set to enable the new notification system in addition to any migrations):

- **mgmt_evt**
  Using `MGMT_EVT_OP_CMD_RECV`, `MGMT_EVT_OP_CMD_STATUS`, or `MGMT_EVT_OP_CMD_DONE` as drop-in replacements for events of the same name, where the provided data is `mgmt_evt_op_cmd_arg`. `CONFIG_MCUMGR_SMP_COMMAND_STATUS_HOOKS` needs to be set.
- **fs_mgmt_register_evt_cb**
  Using `MGMT_EVT_OP_FS_MGMT_FILE_ACCESS` where the provided data is `fs_mgmt_file_access`. Instead of returning true to allow the action or false to deny, a MCUmgr result code needs to be returned, `MGMT_ERR_EOK` will allow the action, any other return code will disallow it and return that code to the client (`MGMT_ERR_EACCESSDENIED` can be used for an access denied error). `CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK` needs to be set.
- **img_mgmt_register_callbacks**
  Using `MGMT_EVT_OP_IMG_MGMT_DFU_STARTED` if `dfu_started_cb` was used, `MGMT_EVT_OP_IMG_MGMT_DFU_STOPPED` if `dfu_stopped_cb` was used, `MGMT_EVT_OP_IMG_MGMT_DFU_PENDING` if `dfu_pending_cb` was used or `MGMT_EVT_OP_IMG_MGMT_DFU_CONFIRMED` if `dfu_confirmed_cb` was used. These callbacks do not have any return status. `CONFIG_MCUMGR_GRP_IMG_STATUS_HOOKS` needs to be set.
- **img_mgmt_set_upload_cb**
  Using `MGMT_EVT_OP_IMG_MGMT_DFU_CHUNK` where the provided data is `img_mgmt_upload_check`. Instead of returning true to allow the action or false to deny, a MCUmgr result code needs to be returned, `MGMT_ERR_EOK` will allow the action, any other return code will disallow it and return that code to the client (`MGMT_ERR_EACCESSDENIED` can be used for an access denied error). `CONFIG_MCUMGR_GRP_IMG_UPLOAD_CHECK_HOOK` needs to be set.
- **os_mgmt_register_reset_evt_cb**
  Using `MGMT_EVT_OP_OS_MGMT_RESET`. Instead of returning true to allow the action or false to deny, a MCUmgr result code needs to be returned, `MGMT_ERR_EOK` will allow the action, any other return code will disallow it and return that code to the client (`MGMT_ERR_EACCESSDENIED` can be used for an access denied error). `CONFIG_MCUMGR_GRP_OS_RESET_HOOK` needs to be set.

API Reference

Callbacks

4.9.4 SMP Protocol Specification

This is description of Simple Management Protocol, SMP, that is used by MCUmgr to pass requests to devices and receive responses from them.

SMP is an application layer protocol. The underlying transport layer is not in scope of this documentation.

> **Note**
>
> SMP in this context refers to SMP for MCUmgr (Simple Management Protocol), it is unrelated to SMP in Bluetooth (Security Manager Protocol), but there is an MCUmgr SMP transport for Bluetooth.

Frame: The envelope

Each frame consists of a header and data. The `Data Length` field in the header may be used for reassembly purposes if underlying transport layer supports fragmentation. Frames are encoded in "Big Endian" (Network endianness) when fields are more than one byte long, and takes the following form:

3								2								1								0							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Res			Ver		OP			Flags								Data Length															
Group ID																Sequence Num								Command ID							
Data ...																															

> **Note**
>
> The original specification states that SMP should support receiving both the "Little-endian" and "Big-endian" frames but in reality the MCUmgr library is hardcoded to always treat "Network" side as "Big-endian".

Data is optional and is not present when `Data Length` is zero. The encoding of data depends on the target of group/ID.

A description of the various fields and their meaning:

Field	Description
Res	This is reserved, not-used field and must be always set to 0.
Ver (Version)	This indicates the version of the protocol being used, this should be set to 0b01 to use the newer SMP transport where error codes are more detailed and returned in the map, otherwise left as 0b00 to use the legacy SMP protocol. Versions 0b10 and 0b11 are reserved for future use and should not be used.
OP	mcumgr_op_t, determines whether information is written to a device or requested from it and whether a packet contains request to an SMP server or response from it.
Flags	Reserved for flags; there are no flags defined yet, the field should be set to 0
Data Length	Length of the Data field
Group ID	mcumgr_group_t, see *Management Group ID's* for further details.
Sequence Num	This is a frame sequence number. The number is increased by one with each request frame. The Sequence Num of a response should match the one in the request.
Command ID	This is a command, within Group.
Data	This is data payload of the Data Length size. It is optional as Data Length may be set to zero, which means that no data follows the header.

Note

Contents of Data depends on a value of an OP, a Group ID, and a Command ID.

Management Group ID's The SMP protocol supports predefined common groups and allows user defined groups. The following table presents a list of common groups:

Decimal ID	Group description
0	*Default/OS Management Group*
1	*Application/software image management group*
2	*Statistics management*
3	*Settings (Config) Management Group*
4	Application/system log management (currently not used by Zephyr)
5	Run-time tests (unused by Zephyr)
6	Split image management (unused by Zephyr)
7	Test crashing application (unused by Zephyr)
8	*File system management*
9	*Shell management*
63	*Zephyr Management Group*
64	This is the base group for defining an application specific management groups.

The payload for above groups, except for user groups (64 and above) is always CBOR encoded. The group 64, and above can define their own scheme for data communication.

Minimal response

Regardless of a command issued, as long as there is SMP client on the other side of a request, a response should be issued containing the header followed by CBOR map container. Lack of response is only allowed when there is no SMP service or device is non-responsive.

Minimal response SMP data Minimal response is:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Note that in the case of a successful command, an empty map will be returned (`rc`/`err` is only returned if there is an error condition, therefore if only an empty map is returned or a response lacks these, the request can be considered as being successful. For SMP version 2, errors relating to SMP itself that are not group specific will still be returned as `rc` errors, SMP version 2 clients must therefore be able to handle both types of errors.

Specifications of management groups supported by Zephyr

Default/OS Management Group OS management group defines following commands:

Command ID	Command description
0	Echo
1	Console/Terminal echo control; unimplemented by Zephyr
2	Task Statistics
3	Memory pool statistics
4	Date-time string
5	System reset
6	MCUMGR parameters
7	OS/Application info
8	Bootloader information

Echo command Echo command responses by sending back string that it has received.

Echo request Echo request header fields:

OP	Group ID	Command ID
0 or 2	0	0

CBOR data of request:

```
{
    (str)"d" : (str)
}
```

where:

"d"	string to be replied by echo service.

Echo response Echo response header fields:

OP	Group ID	Command ID	Note
1	0	0	When request OP was 0
3	0	0	When request OP was 2

CBOR data of successful response:

```
{
    (str)"r"        : (str)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"r"	replying echo string.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Task statistics command The command responds with some system statistics.

Task statistics request Task statistics request header fields:

OP	Group ID	Command ID
0	0	2

The command sends an empty CBOR map as data.

Task statistics response Task statistics response header fields:

OP	Group ID	Command ID
1	0	2

CBOR data of successful response:

```
{
    (str)"tasks" : {
        (str)<task_name> : {
            (str)"prio"         : (uint)
            (str)"tid"          : (uint)
            (str)"state"        : (uint)
            (str)"stkuse"       : (uint)
            (str)"stksiz"       : (uint)
            (str)"cswcnt"       : (uint)
            (str)"runtime"      : (uint)
            (str)"last_checkin" : (uint)
            (str)"next_checkin" : (uint)
        }
        ...
    }
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

<task_nam	string identifying task.
"prio"	task priority.
"tid"	numeric task ID.
"state"	numeric task state.
"stkuse"	task's/thread's stack usage.
"stksiz"	task's/thread's stack size.
"cswcnt"	task's/thread's context switches.
"run-time"	task's/thread's runtime in "ticks".
"last_check	set to 0 by Zephyr.
"next_chec	set to 0 by Zephyr.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Note

The unit for "stkuse" and "stksiz" is system dependent and in case of Zephyr this is number of 4 byte words.

Memory pool statistics The command is used to obtain information on memory pools active in running system.

Memory pool statistic request Memory pool statistics request header fields:

OP	Group ID	Command ID
0	0	3

The command sends an empty CBOR map as data.

Memory pool statistics response Memory pool statistics response header fields:

OP	Group ID	Command ID
1	0	3

CBOR data of successful response:

```
{
    (str)"mpools" : {
        (str)<pool_name> : {
            (str)"blksiz"   : (int)
            (str)"nblks"    : (int)
            (str)"nfree"    : (int)
            (str)"min'      : (int)
        }
        ...
    }
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

<pool_na	string representing the pool name, used as a key for dictionary with pool statistics data.
"blk-siz"	size of the memory block in the pool. This value is always 1 in Zephyr, it can be omitted from responses by enabling `CONFIG_MCUMGR_GRP_OS_MPSTAT_ONLY_SUPPORTED_STATS`.
"nblks"	number of blocks in the pool.
"nfree"	number of free blocks.
"min"	lowest number of free blocks the pool reached during run-time.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Date-time command The command allows to obtain string representing current time-date on a device or set a new time to a device. The time format used, by both set and get operations, is:

> "yyyy-MM-dd'T'HH:mm:ss.SSSSSSZZZZZ"

Date-time get The command allows to obtain date-time from a device.

Date-time get request Date-time request header fields:

OP	Group ID	Command ID
0	0	4

The command sends an empty CBOR map as data.

Date-time get response Date-time get response header fields:

OP	Group ID	Command ID
1	0	4

CBOR data of successful response:

```
{
    (str)"datetime" : (str)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"date-time"	String in format: `yyyy-MM-dd'T'HH:mm:ss.SSSSSSZZZZZ`.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Date-time set The command allows to set date-time to a device.

Date-time set request Date-time set request header fields:

OP	Group ID	Command ID
2	0	4

CBOR data of response:

```
{
    (str)"datetime" : (str)
}
```

where:

"datetime"	String in format: `yyyy-MM-dd'T'HH:mm:ss.SSSSSSZZZZZ`.

Date-time set response Date-time set response header fields:

OP	Group ID	Command ID
3	0	4

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

System reset Performs reset of system. The device should issue response before resetting so that the SMP client could receive information that the command has been accepted. By default, this command is accepted in all conditions, however if the `CONFIG_MCUMGR_GRP_OS_RESET_HOOK` is enabled and an application registers a callback, the callback will be called when this command is issued and can be used to perform any necessary tidy operations prior to the module rebooting, or to reject the reset request outright altogether with an error response. For details on this functionality, see *MCUmgr Callbacks*.

System reset request System reset request header fields:

OP	Group ID	Command ID
2	0	5

Normally the command sends an empty CBOR map as data, but if a previous reset attempt has responded with "rc" equal to `MGMT_ERR_EBUSY` then the following map may be sent to force a reset:

```
{
    (opt)"force"       : (int)
    (opt)"boot_mode"   : (uint)
}
```

where:

"force"	Force reset if value > 0, optional if 0.
"boot_mo	Specifies boot mode for image or application mode/selection, requires `CONFIG_MCUMGR_GRP_OS_RESET_BOOT_MODE` to be enabled.

System reset response System reset response header fields

OP	Group ID	Command ID
3	0	5

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

MCUmgr Parameters Used to obtain parameters of mcumgr library.

MCUmgr Parameters Request MCUmgr parameters request header fields:

OP	Group ID	Command ID
0	0	6

The command sends an empty CBOR map as data.

MCUmgr Parameters Response MCUmgr parameters response header fields

OP	Group ID	Command ID
1	0	6

CBOR data of successful response:

```
{
    (str)"buf_size"     : (uint)
    (str)"buf_count"    : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"buf_size"	Single SMP buffer size, this includes SMP header and CBOR payload.
"buf_count	Number of SMP buffers supported.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

OS/Application Info Used to obtain information on running image, similar functionality to the linux uname command, allowing details such as kernel name, kernel version, build date/time, processor type and application-defined details to be returned. This functionality can be enabled with `CONFIG_MCUMGR_GRP_OS_INFO`.

OS/Application Info Request OS/Application info request header fields:

OP	Group ID	Command ID
0	0	7

CBOR data of request:

```
{
    (str,opt)"format"      : (str)
}
```

where:

"format'	Format specifier of returned response, fields are appended in their natural ascending index order, not the order of characters that are received by the command. Format specifiers: * `s` Kernel name * `n` Node name * `r` Kernel release * `v` Kernel version * `b` Build date and time (requires `CONFIG_MCUMGR_GRP_OS_INFO_BUILD_DATE_TIME`) * `m` Machine * `p` Processor * `i` Hardware platform * `o` Operating system * `a` All fields (shorthand for all above options) If this option is not provided, the `s` Kernel name option will be used.

OS/Application Info Response OS/Application info response header fields

OP	Group ID	Command ID
1	0	7

CBOR data of successful response:

```
{
    (str)"output"       : (str)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"output"	Text response including requested parameters.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Bootloader Information Allows retrieving information about the on-board bootloader and its parameters.

Bootloader Information Request Bootloader information request header:

OP	Group ID	Command ID
0	0	8

CBOR data of request:

```
{
    (str,opt)"query"  : (str)
}
```

where:

"que	Is string representing query for parameters, with no restrictions how the query looks like as processing of query is left for bootloader backend. If there is no query, then response will return string identifying the bootloader.

Bootloader Information Response Bootloader information response header:

OP	Group ID	Command ID
1	0	8

In case when no "query" has been provided in request, CBOR data of response:

```
{
    (str)"bootloader"      : (str)
}
```

where:

"bootloader"	String representing bootloader name

In case when "query" is provided:

```
{
    (str,opt)<response>   : ()
    ...
}
```

where:

<response>	Response to "query". This is optional and may be left out in case when query yields no response, SMP version 2 error code of *OS_MGMT_ERR_QUERY_YIELDS_NO_ANSWER* is expected. Response may have more than one parameter reported back or it may be a map, that is dependent on bootloader backednd and query.
...	Parameter characteristic information.

Parameter may be accompanied by additional, parameter specific, information keywords with assigned values.

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Bootloader Information: MCUboot In case when MCUboot is application bootloader, empty request will be responded with:

```
{
    (str)"bootloader"       : (str)"MCUboot"
}
```

Currently "MCUboot" supports querying for mode of operation:

```
{
    (str)"query"            : (str)"mode"
}
```

Response to "mode" is:

```
{
    (str)"mode"                     : (int)
    (str,opt)"no-downgrade"         : (bool)
}
```

where "mode" is one of:

-1	Unknown mode of MCUboot.
0	MCUboot is in single application mode.
1	MCUboot is in swap using scratch partition mode.
2	MCUboot is in overwrite (upgrade-only) mode.
3	MCUboot is in swap without scratch mode.
4	MCUboot is in DirectXIP without revert mode.
5	MCUboot is in DirectXIP with revert mode.
6	MCUboot is in RAM loader mode.

The `no-downgrade` field is a flag, which is always sent when true, indicating that MCUboot has downgrade prevention enabled; downgrade prevention means that if the uploaded image has a lower version than the currently running application, it will not be used for an update by MCUboot.

MCUmgr may reject images with a lower version in this configuration.

Application/software image management group Application/software image management group defines following commands:

Command ID	Command description
0	State of images
1	Image upload
2	File (reserved but not supported by Zephyr)
3	Corelist (reserved but not supported by Zephyr)
4	Coreload (reserved but not supported by Zephyr)
5	Image erase
6	Slot info

Notion of "slots" and "images" in Zephyr The "slot" and "image" definition comes from mcuboot where "image" would consist of two "slots", further named "primary" and "secondary"; the application is supposed to run from the "primary slot" and update is supposed to be uploaded to the "secondary slot"; the mcuboot is responsible in swapping slots on boot. This means that pair of slots is dedicated to single upgradable application. In case of Zephyr this gets a little bit confusing because DTS will use "slot0_partition" and "slot1_partition", as label of `fixed-partitions` or `zephyr,mapped-partition` dedicated to single application, but will name them as "image-0" and "image-1" respectively.

Currently Zephyr supports at most two images, in which case mapping is as follows:

Image	Slot labels	Slot Names
1	"slot0_partition" "slot1_partition"	"image-0" "image-1"
2	"slot2_partition" "slot3_partition"	"image-2" "image-3"

State of images The command is used to set state of images and obtain list of images with their current state.

Get state of images request Get state of images request header fields:

OP	Group ID	Command ID
0	1	0

The command sends an empty CBOR map as data.

Get state of images response Get state of images response header fields:

OP	Group ID	Command ID
1	1	0

> **Note**
>
> Below definition of the response contains "image" field that has been marked as optional(opt): the field may not appear in response when target application does not support more than one image. The field is mandatory when application supports more than one application image to allow identifying which image information is listed.

A response will only contain information for valid images, if an image can not be identified as valid it is simply skipped.

CBOR data of successful response:

```
{
    (str)"images" : [
        {
            (str,opt)"image"        : (uint)
            (str)"slot"             : (uint)
            (str)"version"          : (str)
            (str,opt*)"hash"        : (byte str)
            (str,opt)"bootable"     : (bool)
            (str,opt)"pending"      : (bool)
            (str,opt)"confirmed"    : (bool)
            (str,opt)"active"       : (bool)
            (str,opt)"permanent"    : (bool)
        }
        ...
    ]
    (str,opt)"splitStatus" : (int)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
    (str,opt)"rsn"  : (str)
}
```

where:

"image"	semi-optional image number; the field is not required when only one image is supported by the running application.
"slot	slot number within "image"; each image has two slots : primary (running one) = 0 and secondary (for DFU dual-bank purposes) = 1.
"version"	string representing image version, as set with `imgtool`.
"hash	SHA256 hash of the image header and body. Note that this will not be the same as the SHA256 of the whole file, it is the field in the MCUboot TLV section that contains a hash of the data which is used for signature verification purposes. This field is optional but only optional when using MCUboot's serial recovery feature with one pair of image slots, Kconfig `CONFIG_BOOT_SERIAL_IMG_GRP_HASH` can be disabled to remove support for hashes in this configuration. MCUmgr in applications must support sending hashes. **Note** See `IMAGE_TLV_SHA256` in the MCUboot image format documentation link below.
"bootable"	true if image has bootable flag set; this field does not have to be present if false.
"pending"	true if image is set for next swap; this field does not have to be present if false.
"confirm	true if image has been confirmed; this field does not have to be present if false.
"active"	true if image is currently active application; this field does not have to be present if false.
"permanent	true if image is to stay in primary slot after the next boot; this does not have to be present if false.
"splitStatus"	states whether loader of split image is compatible with application part; this is unused by Zephyr.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.
"rsn"	optional string that clarifies reason for an error; specifically useful when `rc` is `MGMT_ERR_EUNKNOWN`.

Note

For more information on how does image/slots function, please refer to the MCUBoot documentation https://docs.mcuboot.com/design.html#image-slots For information on MCUboot image format, please reset to the MCUboot documentation https://docs.mcuboot.com/design.html#image-format

Set state of image request Set state of image request header fields:

OP	Group ID	Command ID
2	1	0

CBOR data of request:

```
{
    (str,opt)"hash"     : (byte str)
    (str)"confirm"      : (bool)
}
```

If "confirm" is false or not provided, an image with the "hash" will be set for test, which means that it will not be marked as permanent and upon hard reset the previous application will be restored to the primary slot. In case when "confirm" is true, the "hash" is optional as the currently running application will be assumed as target for confirmation.

Set state of image response The response takes the same format as *Get state of images response*

Image upload The image upload command allows to update application image.

Image upload request The image upload request is sent for each chunk of image that is uploaded, until complete image gets uploaded to a device.

Image upload request header fields:

OP	Group ID	Command ID
2	1	1

CBOR data of request:

```
{
    (str,opt)"image"    : (uint)
    (str,opt)"len"      : (uint)
    (str)"off"          : (uint)
    (str,opt)"sha"      : (byte str)
    (str)"data"         : (byte str)
    (str,opt)"upgrade"  : (bool)
}
```

where:

"image"	optional image number, it does not have to appear in request at all, in which case it is assumed to be 0. Should only be present when "off" is 0.
"len"	optional length of an image. Must appear when "off" is 0.
"off"	offset of image chunk the request carries.
"sha	SHA256 hash of an upload; this is used to identify an upload session (e.g. to allow MCUmgr to continue a broken session), and for image verification purposes. This must be a full SHA256 hash of the whole image being uploaded, or not included if the hash is not available (in which case, upload session continuation and image verification functionality will be unavailable). Should only be present when "off" is 0.
"dat	image data to write at provided offset.
"upgrad	optional flag that states that only upgrade should be allowed, so if the version of uploaded software is not higher then already on a device, the image upload will be rejected. Zephyr compares major, minor and revision (x.y.z) by default unless `CONFIG_MCUMGR_GRP_IMG_VERSION_CMP_USE_BUILD_NUMBER` is set, whereby it will compare build numbers too. Should only be present when "off" is 0.

Note

There is no field representing size of chunk that is carried as "data" because that information is embedded within "data" field itself.

Note

It is possible that a server will respond to an upload with "off" of 0, this may happen if an upload on another transport (or outside of MCUmgr entirely) is started, if the device has rebooted or if a packet has been lost. If this happens, a client must re-send all the required and optional fields that it sent in the original first packet so that the upload state can be re-created by the server. If the original fields are not included, the upload will be unable to continue.

The MCUmgr library uses "sha" field to tag ongoing update session, to be able to continue it in case when it gets broken, and for upload verification purposes. If library gets request with "off" equal zero it checks stored "sha" within its state and if it matches it will respond to update client application with offset that it should continue with. If this hash is not available (e.g. because a file is being streamed) then it must not be provided, image verification and upload session continuation features will be unavailable in this case.

Image upload response Image upload response header fields:

OP	Group ID	Command ID
3	1	1

CBOR data of successful response:

```
{
    (str,opt)"off"    : (uint)
    (str,opt)"match"  : (bool)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
    (str,opt)"rsn"  : (str)
}
```

where:

"off"	offset of last successfully written byte of update.
"match"	indicates if the uploaded data successfully matches the provided SHA256 hash or not, only sent in the final packet if `CONFIG_IMG_ENABLE_IMAGE_CHECK` is enabled.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.
"rsn"	optional string that clarifies reason for an error; specifically useful when `rc` is `MGMT_ERR_EUNKNOWN`.

The "off" field is only included in responses to successfully processed requests; if "rc" is negative then "off" may not appear.

Image erase The command is used for erasing image slot on a target device.

> **Note**
>
> This is synchronous command which means that a sender of request will not receive response until the command completes, which can take a long time.

Image erase request Image erase request header fields:

OP	Group ID	Command ID
2	1	5

CBOR data of request:

```
{
    (str,opt)"slot"     : (uint)
}
```

where:

"slot"	optional slot number, it does not have to appear in the request at all, in which case it is assumed to be 1.

Image erase response Image erase response header fields:

OP	Group ID	Command ID
3	1	5

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
```

(continues on next page)

(continued from previous page)

```
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
    (str,opt)"rsn"  : (str)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.
"rsn"	optional string that clarifies reason for an error; specifically useful when `rc` is `MGMT_ERR_EUNKNOWN`.

Note

Response from Zephyr running device may have "rc" value of `MGMT_ERR_EBADSTATE`, which means that the secondary image has been marked for next boot already and may not be erased.

Slot info The command is used for fetching information on slots that are available. This command can be enabled with `CONFIG_MCUMGR_GRP_IMG_SLOT_INFO`.

Slot info request Slot info request header fields:

OP	Group ID	Command ID
0	1	6

The command sends an empty CBOR map as data.

Slot info response Slot info response header fields:

OP	Group ID	Command ID
1	1	6

CBOR data of successful response:

```
{
    (str)"images" : [
        {
            (str)"image"                        : (uint)
```

(continues on next page)

(continued from previous page)

```
            (str)"slots" : [
                {
                    (str)"slot"                 : (uint)
                    (str)"size"                 : (uint)
                    (str,opt)"upload_image_id"  : (uint)
                }
                ...
            ]
            (str,opt)"max_image_size"           : (uint)
        }
        ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"image"	the image number being enumerated.
"slot"	the slot inside the image being enumerated (note: this will be 0 or 1, it is the slot of the image not the physical slot number).
"size"	the size of the slot.
"upload_im	optional field (only present if `CONFIG_MCUMGR_GRP_IMG_DIRECT_UPLOAD` is enabled to allow direct image uploads) which specifies the image ID that can be used by external tools to upload an image to that slot.
"max_ir	optional field (only present if `CONFIG_MCUMGR_GRP_IMG_TOO_LARGE_SYSBUILD` or `CONFIG_MCUMGR_GRP_IMG_TOO_LARGE_BOOTLOADER_INFO` are enabled) which specifies the maximum size of an application that can be uploaded to that image number.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears ifnon-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Statistics management Statistics management allows to obtain data gathered by Statistics subsystem of Zephyr, enabled with `CONFIG_STATS`.

Statistics management group defines commands:

Command ID	Command description
0	Group data
1	List groups

Statistics: group data The command is used to obtain data for group specified by a name. The name is one of group names as registered, with `STATS_INIT_AND_REG` macro or `stats_init_and_reg()` function call, within module that gathers the statistics.

Statistics: group data request Statistics group data request header:

OP	Group ID	Command ID
0	2	0

CBOR data of request:

```
{
    (str)"name" :  (str)
}
```

where:

"name"	group name.

Statistics: group data response Statistics group data response header:

OP	Group ID	Command ID
1	2	0

CBOR data of successful response:

```
{
    (str)"name"     : (str)
    (str)"fields"   : {
        (str)<entry_name> : (uint)
        ...
    }
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"name"	this is name of group the response contains data for.
"fields"	this is map of entries within groups that consists of pairs where the entry name is mapped to value it represents in statistics.
<entry_name>	single entry to value mapping; value is hardcoded to unsigned integer type, in a CBOR meaning.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Statistics: list of groups The command is used to obtain list of groups of statistics that are gathered on a device. This is a list of names as given to groups with `STATS_INIT_AND_REG` macro or `stats_init_and_reg()` function calls, within module that gathers the statistics; this means that this command may be considered optional as it is known during compilation what groups will be included into build and listing them is not needed prior to issuing a query.

Statistics: list of groups request Statistics group list request header:

OP	Group ID	Command ID
0	2	1

The command sends an empty CBOR map as data.

Statistics: list of groups response Statistics group list request header:

OP	Group ID	Command ID
1	2	1

CBOR data of successful response:

```
{
    (str)"stat_list" :  [
        (str)<stat_group_name>, ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
```

(continues on next page)

(continued from previous page)

```
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"stat_list"	array of strings representing group names; this array may be empty if there are no groups.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Settings (Config) Management Group Settings management group (known as Configuration Manager in the original MCUmgr repository) defines the following commands:

Command ID	Command description
0	Read/write setting
1	Delete setting
2	Commit settings
3	Load/Save settings

Note that the Zephyr version adds additional commands and features which are not supported by the original upstream version, however, the original client functionality should work for read/write functionality.

Read/write setting command Read/write setting command allows updating a setting entry on a device or getting the current value of a setting from a device.

Read setting request Read setting request header fields:

OP	Group ID	Command ID
0	3	0

CBOR data of request:

```
{
    (str)"name"         : (str)
    (str,opt)"max_size" : (uint)
}
```

where:

"name"	string of the setting to retrieve
"max_size"	optional maximum size of data to return

Read setting response Read setting response header fields:

OP	Group ID	Command ID
1	3	0

CBOR data of successful response:

```
{
    (str)"val"          : (bstr)
    (str,opt)"max_size" : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"val"	binary string of the returned data, note that the underlying data type cannot be specified through this and must be known by the client.
"max_s	will be set if the maximum supported data size is smaller than the maximum requested data size, and contains the maximum data size which the device supports, equivalent to kconfig:option:*CONFIG_MCUMGR_GRP_SETTINGS_NAME_LEN*.
"err" -> "group	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Write setting request Write setting request header fields:

OP	Group ID	Command ID
2	3	0

CBOR data of request:

```
{
    (str)"name"    : (str)
```

(continues on next page)

(continued from previous page)

```
    (str)"val"      : (bstr)
}
```

where:

"name"	string of the setting to update/set
"val"	value to set the setting to

Write setting response Write setting response header fields:

OP	Group ID	Command ID
3	3	0

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Delete setting command Delete setting command allows deleting a setting on a device.

Delete setting request Delete setting request header fields:

OP	Group ID	Command ID
2	3	1

CBOR data of request:

```
{
    (str)"name"   : (str)
}
```

where:

"name"	string of the setting to delete

Delete setting response Delete setting response header fields:

OP	Group ID	Command ID
3	3	1

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Commit settings command Commit settings command allows committing all settings that have been set but not yet applied on a device.

Commit settings request Commit settings request header fields:

OP	Group ID	Command ID
2	3	2

The command sends an empty CBOR map as data.

Commit settings response Commit settings response header fields:

OP	Group ID	Command ID
3	3	2

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Load/Save settings command Load/Save settings command allows loading/saving all serialized items from/to persistent storage on a device.

Load settings request Load settings request header fields:

OP	Group ID	Command ID
0	3	3

The command sends an empty CBOR map as data.

Load settings response Load settings response header fields:

OP	Group ID	Command ID
1	3	3

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Save settings request Save settings request header fields:

OP	Group ID	Command ID
2	3	3

CBOR data of request:

```
{
    (str,opt)"name" : (str)
}
```

where:

"nan	If provided, contains the settings subtree name to save (can also be a single setting name if `CONFIG_SETTINGS_SAVE_SINGLE_SUBTREE_WITHOUT_MODIFICATION` is enabled), if not then will save all settings.

Save settings response Save settings response header fields:

OP	Group ID	Command ID
3	3	3

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Settings access callback There is a settings access MCUmgr callback available (see *MCUmgr Callbacks* for details on callbacks) which allows for applications/modules to know when settings management commands are used and, optionally, block access (for example through the use of a security mechanism). This callback can be enabled with `CONFIG_MCUMGR_GRP_SETTINGS_ACCESS_HOOK`, registered with the event `MGMT_EVT_OP_SETTINGS_MGMT_ACCESS`, whereby the supplied callback data is `settings_mgmt_access`.

File system management The file system management group provides commands that allow to upload and download files to/from a device.

File management group defines following commands:

Command ID	Command description
0	File download/upload
1	File status
2	File hash/checksum
3	Supported file hash/checksum types
4	File close

File download Command allows to download contents of an existing file from specified path of a target device. Client applications must keep track of data they have already downloaded and where their position in the file is (MCUmgr will cache these also), and issue subsequent requests, with modified offset, to gather the entire file. Request does not carry size of requested chunk, the size is specified by application itself. Note that file handles will remain open for consecutive requests (as long as an idle timeout has not been reached and another transport does not make use of uploading/downloading files using fs_mgmt), but files are not exclusively owned by MCUmgr, for the time of download session, and may change between requests or even be removed.

> **Note**
>
> By default, all file upload/download requests are unconditionally allowed. However, if the Kconfig option `CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK` is enabled, then an application can register a callback handler for `MGMT_EVT_OP_FS_MGMT_FILE_ACCESS` (see *MCUmgr callbacks*), which allows for allowing or declining access to reading/writing a particular file, or for rewriting the path supplied by the client.

File download request File download request header:

OP	Group ID	Command ID
0	8	0

CBOR data of request:

```
{
    (str)"off" :  (uint)
    (str)"name" : (str)
}
```

where:

"off"	offset to start download at
"name"	absolute path to a file

File download response File download response header:

OP	Group ID	Command ID
1	8	0

CBOR data of successful response:

```
{
    (str)"off"      : (uint)
    (str)"data"     : (byte str)
    (str,opt)"len"  : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"off"	offset the response is for.
"data"	chunk of data read from file; it is CBOR encoded stream of bytes with embedded size; "data" appears only in responses where "rc" is 0.
"len"	length of file, this field is only mandatory when "off" is 0.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File upload Allows to upload a file to a specified location. Command will automatically overwrite existing file or create a new one if it does not exist at specified path. The protocol supports stateless upload where each requests carries different chunk of a file and it is client side responsibility to track progress of upload.

Note that file handles will remain open for consecutive requests (as long as an idle timeout has not been reached, but files are not exclusively owned by MCUmgr, for the time of download session, and may change between requests or even be removed. Note that file handles will remain open for consecutive requests (as long as an idle timeout has not been reached and another transport does not make use of uploading/downloading files using fs_mgmt), but files are not exclusively owned by MCUmgr, for the time of download session, and may change between requests or even be removed.

> **Note**
>
> Weirdly, the current Zephyr implementation is half-stateless as is able to hold single upload context, holding information on ongoing upload, that consists of bool flag indicating in-progress upload, last successfully uploaded offset and total length only.

> **Note**
>
> By default, all file upload/download requests are unconditionally allowed. However, if the Kconfig option `CONFIG_MCUMGR_GRP_FS_FILE_ACCESS_HOOK` is enabled, then an application can register a callback handler for `MGMT_EVT_OP_FS_MGMT_FILE_ACCESS` (see *MCUmgr callbacks*), which allows for allowing or declining access to reading/writing a particular file, or for rewriting the path supplied by the client.

File upload request File upload request header:

OP	Group ID	Command ID
2	8	0

CBOR data of request:

```
{
    (str)"off"      : (uint)
    (str)"data"     : (str)
    (str)"name"     : (str)
    (str,opt)"len"  : (uint)
}
```

where:

“off”	offset to start/continue upload at.
“data”	chunk of data to write to the file; it is CBOR encoded with length embedded.
“name”	absolute path to a file.
“len”	length of file, this field is only mandatory when “off” is 0.

File upload response File upload response header:

OP	Group ID	Command ID
3	8	0

CBOR data of successful response:

```
{
    (str)"off"      : (uint)
}
```

In case of error the CBOR data takes the form:

where:

"off"	offset of last successfully written data.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File status Command allows to retrieve status of an existing file from specified path of a target device.

File status request File status request header:

OP	Group ID	Command ID
0	8	1

CBOR data of request:

```
{
    (str)"name" : (str)
}
```

where:

"name"	absolute path to a file.

File status response File status response header:

OP	Group ID	Command ID
1	8	1

CBOR data of successful response:

```
{
    (str)"len"      : (uint)
}
```

In case of error the CBOR data takes form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"len"	length of file (in bytes).
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File hash/checksum Command allows to generate a hash/checksum of an existing file at a specified path on a target device. Note that kernel heap memory is required for buffers to be allocated for this to function, and large stack memory buffers are required for generation of the output hash/checksum. Requires `CONFIG_MCUMGR_GRP_FS_CHECKSUM_HASH` to be enabled for the base functionality, supported hash/checksum are opt-in with `CONFIG_MCUMGR_GRP_FS_CHECKSUM_IEEE_CRC32` or `CONFIG_MCUMGR_GRP_FS_HASH_SHA256`.

File hash/checksum request File hash/checksum request header:

OP	Group ID	Command ID
0	8	2

CBOR data of request:

```
{
    (str)"name"     : (str)
    (str,opt)"type" : (str)
    (str,opt)"off"  : (uint)
    (str,opt)"len"  : (uint)
}
```

where:

"name"	absolute path to a file.
"type"	type of hash/checksum to perform *Hash/checksum types* or omit to use default.
"off"	offset to start hash/checksum calculation at (optional, 0 if not provided).
"len"	maximum length of data to read from file to generate hash/checksum with (optional, full file size if not provided).

Hash/checksum types

String name	Hash/checksum	Byte string	Size (bytes)
crc32	IEEE CRC32 checksum	No	4
sha256	SHA256 (Secure Hash Algorithm)	Yes	32

Note that the default type will be crc32 if it is enabled, or sha256 if crc32 is not enabled.

File hash/checksum response File hash/checksum response header:

OP	Group ID	Command ID
1	8	2

CBOR data of successful response:

```
{
    (str)"type"     : (str)
    (str,opt)"off"  : (uint)
    (str)"len"      : (uint)
    (str)"output"   : (uint or bstr)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"type"	type of hash/checksum that was performed *Hash/checksum types*.
"off"	offset that hash/checksum calculation started at (only present if not 0).
"len"	length of input data used for hash/checksum generation (in bytes).
"output"	output hash/checksum.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Supported file hash/checksum types Command allows listing which hash and checksum types are available on a device. Requires Kconfig `CONFIG_MCUMGR_GRP_FS_CHECKSUM_HASH_SUPPORTED_CMD` to be enabled.

Supported file hash/checksum types request Supported file hash/checksum types request header:

OP	Group ID	Command ID
0	8	3

The command sends empty CBOR map as data.

Supported file hash/checksum types response Supported file hash/checksum types response header:

OP	Group ID	Command ID
1	8	3

CBOR data of successful response:

```
{
    (str)"types" : {
        (str)<hash_checksum_name> : {
            (str)"format"       : (uint)
            (str)"size"         : (uint)
        }
        ...
    }
}
```

In case of error the CBOR data takes form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

<hash_checksur	name of the hash/checksum type *Hash/checksum types*.
"format"	format that the hash/checksum returns where 0 is for numerical and 1 is for byte array.
"size"	size (in bytes) of output hash/checksum response.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

File close Command allows closing any open file handles held by fs_mgmt upload/download requests that might have stalled or be incomplete.

File close request File close request header:

OP	Group ID	Command ID
2	8	4

The command sends empty CBOR map as data.

File close response File close response header:

OP	Group ID	Command ID
3	8	4

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Shell management Shell management allows passing commands to the shell subsystem over the SMP protocol.

Shell management group defines following commands:

Command ID	Command description
0	Shell command line execute

Shell command line execute The command allows to execute command line in a similar way to typing it into a shell, but both a request and a response are transported over SMP.

Shell command line execute request Execute command request header:

OP	Group ID	Command ID
2	9	0

CBOR data of request:

```
{
    (str)"argv"     : [
        (str)<cmd>
        (str,opt)<arg>
        ...
    ]
}
```

where:

"argv"	array consisting of strings representing command and its arguments.
<cmd>	command to be executed.
<arg>	optional arguments to command.

Shell command line execute response Command line execute response header fields:

OP	Group ID	Command ID
3	9	0

CBOR data of successful response:

```
{
    (str)"o"            : (str)
    (str)"ret"          : (int)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1 (and non-group SMP version 2)

```
{
    (str)"rc"       : (int)
}
```

where:

"o"	command output.
"ret"	return code from shell command execution.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Note

In older versions of Zephyr, "rc" was used for both the mcumgr status code and shell command execution return code, this legacy behaviour can be restored by enabling `CONFIG_MCUMGR_GRP_SHELL_LEGACY_RC_RETURN_CODE`

Enumeration Management Group Enumeration management group defines the following commands:

Command ID	Command description
0	Count of supported groups
1	List supported groups
2	Fetch single group ID
3	Details on supported groups

Count of supported groups command Count of supported groups returns the total number of MCUmgr command groups that a device supports.

Count of supported groups request Read setting request header fields:

OP	Group ID	Command ID
0	10	0

The command sends an empty CBOR map as data.

Count of supported groups response Count of supported groups response header fields:

OP	Group ID	Command ID
1	10	0

CBOR data of successful response:

```
{
    (str)"count"        : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"count"	contains the total number of supported MCUmgr groups on the device.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

List supported groups command List supported groups command allows listing the group IDs of supported MCUmgr groups on the device.

List supported groups request List supported groups request header fields:

OP	Group ID	Command ID
0	10	1

The command sends an empty CBOR map as data.

List supported groups response List supported groups response header fields:

OP	Group ID	Command ID
1	10	1

CBOR data of successful response:

```
{
    (str)"groups" : [
        (uint)
        ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"groups"	contains a list of the supported MCUmgr group IDs on the device.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Fetch single group ID command Fetch single group ID command allows listing the group IDs of supported MCUmgr groups on the device, one by one.

Fetch single group ID request Fetch single group ID request header fields:

OP	Group ID	Command ID
0	10	2

CBOR data of request:

```
{
    (str,opt)"index" : (uint)
}
```

where:

"index"	contains the (0-based) index of the group to return information on, can be omitted to return the first group's details.

Fetch single group ID response Fetch single group ID response header fields:

OP	Group ID	Command ID
1	10	2

CBOR data of successful response:

```
{
    (str)"group"    : (uint)
```

(continues on next page)

(continued from previous page)

```
    (str,opt)"end"  : (bool)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"group"	contains the list of the supported MCUmgr group IDs on the device.
"end"	will be set to true if the listed group is the final supported group on the device, otherwise will be omitted.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Details on supported groups command Details on supported groups command allows fetching details on each supported MCUmgr group, such as the name and number of handlers. A device can specify an allow list of groups to return details on or details on all groups can be returned.

This command is optional, it can be enabled using `CONFIG_MCUMGR_GRP_ENUM_DETAILS`. The optional name and number of handlers can be enabled/disabled with `CONFIG_MCUMGR_GRP_ENUM_DETAILS_NAME` and `CONFIG_MCUMGR_GRP_ENUM_DETAILS_HANDLERS`.

Details on supported groups request Details on supported groups request header fields:

OP	Group ID	Command ID
0	10	3

Details on all groups

The command sends an empty CBOR map as data.

Details on specified groups

CBOR data of request:

```
{
    (str)"groups" : [
        (uint)
        ...
    ]
}
```

where:

"groups"	contains a list of the MCUmgr group IDs to fetch details on.

Details on supported groups response Details on supported groups response header fields:

OP	Group ID	Command ID
1	10	3

CBOR data of successful response:

```
{
    (str)"groups" : [
        {
            (str)"group"          : (uint)
            (str,opt)"name"       : (str)
            (str,opt)"handlers"   : (uint)
        }
        ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"group"	the group ID of the MCUmgr command group.
"name"	the name of the MCUmgr command group.
"handlers"	the number of handlers that the MCUmgr command group supports.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Details on supported groups callback There is a details on supported groups MCUmgr callback available (see *MCUmgr Callbacks* for details on callbacks) which allows for applications/modules to add additional fields to this response. This callback can be enabled with `CONFIG_MCUMGR_GRP_ENUM_DETAILS_HOOK`, registered with the event `MGMT_EVT_OP_ENUM_MGMT_DETAILS`, whereby the supplied callback data is `enum_mgmt_detail_output`. Note that `CONFIG_MCUMGR_GRP_ENUM_DETAILS_STATES` will need incrementing by the number of additional extra fields that are added.

Transport Management Group Transport management group defines the following commands:

Command ID	Command description
0	Connect (bridge) transport
1	Disconnect bridged transport
2	Fetch bridge/transport status
6	List transports
7	Details on transport modes
8	Details on transport configuration

Note

Transport management is experimental and subject to change without notice.

Connect (bridge) transport command Bridge the transport which received the MCUmgr packet to another MCUmgr transport.

Connect (bridge) transport request Connect (bridge) transport request header fields:

OP	Group ID	Command ID
2	11	0

CBOR data of request:

```
{
    (str)"transport" : (uint)
    (str,opt)"mode"  : (uint)
    ...
}
```

where:

"transport"	`smp_transport_type` contains the transport type for which to bridge (connect) from the transport to.
"mode"	contains the configuration mode of the transport to use, may be omitted to use the default value of 0.
...	there might be additional fields that the transport requires in order to make a connection, these are not described here as are transport-specific.

Connect (bridge) transport response Connect (bridge) transport response header fields:

OP	Group ID	Command ID
3	11	0

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Disconnect bridged transport command Disconnect the current transport's bridge, or disconnect all transport bridges.

Disconnect bridged transport request Disconnect bridged transport request header fields:

OP	Group ID	Command ID
2	11	1

CBOR data of request:

Disconnect bridge of current transport

The command sends an empty CBOR map as data.

Disconnect an active bridge

```
{
    (str)"transport" : (uint)
}
```

Disconnect all bridges

```
{
    (str)"all" :       (bool)
}
```

where:

"transport"	`smp_transport_type` contains the transport type for which to disconnect the bridge from, this must not be provided if `all` is provided.
"all"	set to true to disconnect all active bridged transports, this must not be provided if `transport` is provided.

Disconnect bridged transport response Disconnect bridged transport response header fields:

OP	Group ID	Command ID
3	11	1

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Fetch bridge/transport status command Return information on active bridges and what the device supports.

Fetch bridge/transport status request Fetch bridge/transport status request header fields:

OP	Group ID	Command ID
0	11	2

The command sends an empty CBOR map as data.

Fetch bridge/transport status response Fetch bridge/transport status response header fields:

OP	Group ID	Command ID
1	11	2

CBOR data of successful response:

```
{
    (str)"supported"      : (uint)
    (str)"active"         : (uint)
    (str,opt)"bridged"    : (bool)
    (str,opt)"transport"  : (uint)
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"supported"	contains how many bridges can be active at a given time.
"active"	contains how many bridges are currently active.
"bridged"	will be present and true if the current transport is bridged, otherwise will be omitted.
"transport"	the transport ID of the MCUmgr transport that the transport is bridged to. Only appears if the transport is bridged.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

List transports command Return information on transports that the device supports.

List transports request List transports request header fields:

OP	Group ID	Command ID
0	11	6

The command sends an empty CBOR map as data.

List transports response List transports response header fields:

OP	Group ID	Command ID
1	11	6

CBOR data of successful response:

```
{
    (str)"transports" : [
        {
            (str)"id"         : (uint)
            (str,opt)"name"   : (str)
        }
        ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"id"	the transport ID of the MCUmgr transport that supports bridging.
"name"	optional name of the MCUmgr transport (if available).
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Details on transport modes command Return information on modes of a transport that the device supports.

Details on transport mode request Details on transport modes request header fields:

OP	Group ID	Command ID
0	11	7

CBOR data of request:

```
{
    (str)"transport" : (uint)
}
```

where:

"transport"	`smp_transport_type` contains the transport type for which to get details on.

Details on transport modes response Details on transport modes response header fields:

OP	Group ID	Command ID
1	11	7

CBOR data of successful response:

```
{
    (str)"modes" : [
        {
            (str)"id"            : (uint)
            (str)"description"   : (str)
            (str,opt)"incoming"  : (bool)
            (str,opt)"outgoing"  : (bool)
        }
        ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"id"	the ID of the transport mode.
"description"	description of the transport mode.
"incoming"	will be set to true if transport mode supports incoming bridge connections.
"outgoing"	will be set to true if transport mode supports outgoing bridge connections.
"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Details on transport configuration command Return information on transport configuration that is supported.

Details on transport configuration request Details on transport configuration request header fields:

OP	Group ID	Command ID
0	11	8

CBOR data of request:

```
{
    (str)"transport" : (uint)
    (str)"mode"      : (uint)
}
```

where:

"transport"	`smp_transport_type` contains the transport type for which to get configuration information for.
"mode"	contains the configuration mode of the transport type which to get configuration information for.

Details on transport configuration response Details on transport configuration response header fields:

OP	Group ID	Command ID
1	11	8

CBOR data of successful response:

```
{
    (str)"configs" : [
        {
            (str)"name"          : (str)
            (str)"type"          : (uint)
            (str,opt)"required"  : (bool)
```

(continues on next page)

(continued from previous page)

```
        }
        ...
    ]
}
```

In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"nar	name of the configuration item.
"typ	**the type of the configuration item, using the following mapping:** • 0: uint • 1: int • 2: bool • 3: string • 4: byte string
"re-quir	will be present and set to true if the parameter is required.
"err -> "gro	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

Zephyr Management Group Zephyr management group defines the following commands:

Command ID	Command description
0	Erase storage

Erase storage command Erase storage command allows clearing the `storage_partition` flash partition on a device, generally this is used when switching to a new application build if the application uses storage that should be cleared (application dependent).

Erase storage request Erase storage request header fields:

OP	Group ID	Command ID
2	63	0

The command sends sends empty CBOR map as data.

Erase storage response Read setting response header fields:

OP	Group ID	Command ID
3	63	0

The command sends an empty CBOR map as data if successful. In case of error the CBOR data takes the form:

SMP version 2

```
{
    (str)"err" : {
        (str)"group"    : (uint)
        (str)"rc"       : (uint)
    }
}
```

SMP version 1

```
{
    (str)"rc"       : (int)
}
```

where:

"err" -> "group"	`mcumgr_group_t` group of the group-based error code. Only appears if an error is returned when using SMP version 2.
"err" -> "rc"	contains the index of the group-based error code. Only appears if non-zero (error condition) when using SMP version 2.
"rc"	`mcumgr_err_t` only appears if non-zero (error condition) when using SMP version 1 or for SMP errors when using SMP version 2.

4.9.5 SMP Transport Specification

The documents specifies information needed for implementing server and client side SMP transports.

Bluetooth Low Energy (LE)

MCUmgr Clients need to use following Bluetooth Characteristics, when implementing SMP client:

- **Service UUID**: `8D53DC1D-1DB7-4CD3-868B-8A527460AA84`
- **Characteristic UUID**: `DA2E7828-FBCE-4E01-AE9E-261174997C48`

All SMP communication utilizes a single GATT characteristic. An SMP request is sent via a GATT Write Without Response command. An SMP response is sent in the form of a GATT Notification

If an SMP request or response is too large to fit in a single GATT command, the sender fragments it across several packets. No additional framing is introduced when a request or response is fragmented; the payload is simply split among several packets. Since GATT guarantees ordered

delivery of packets, the SMP header in the first fragment contains sufficient information for reassembly.

UART/serial and console

SMP protocol specification by MCUmgr subsystem of Zephyr uses basic framing of data to allow multiplexing of UART channel. Multiplexing requires prefixing each frame with two byte marker and terminating it with newline. Currently MCUmgr imposes a 127 byte limit on frame size, although there are no real protocol constraints that require that limit. The limit includes the prefix and the newline character, so the allowed payload size is actually 124 bytes.

Although no such transport exists in Zephyr, it is possible to implement MCUmgr client/server over UART transport that does not have framing at all, or uses hardware serial port control, or other means of framing.

Frame fragmenting SMP protocol over serial is fragmented into frames of up to 127 bytes; each frame consists of two byte start marker, body and terminating newline character.

There are four types of types of frames: initial, partial, partial-final and initial-final; each frame type differs by start marker and/or body contents.

Frame formats Initial frame requires to be followed by optional sequence of partial frames and finally by partial-final frame. Body is always Base64 encoded, so the body size, here described as MTU - 3, is able to actually carry N = (MTU - 3) / 4 * 3 bytes of raw data.

Body of initial frame is preceded by two byte total packet length, encoded in Big Endian, and equals size of a raw body plus two bytes, size of CRC16; this means that actual body size allowed into an initial frame is N - 2.

If a body size is smaller than N - 4, than it is possible to carry entire body with preceding length and following it CRC in a single frame, here called initial-final; for the description of initial-final frame look below.

Initial frame format:

Content	Size	Description
0x06 0x09	2 bytes	Frame start marker
<base64-i>	no more than MTU - 3 bytes	Base64 encoded body
0x0a	1 byte	Frame termination

`<base64-i>` is Base64 encoded body of format:

Content	Size	Description
total length	2 bytes	Big endian 16-bit value representing total length of body + 2 bytes for CRC16; note that size of total length field is not added to total length value.
body	no more than MTU - 5	Raw body data fragment

Initial-final frame format is similar to initial frame format, but differs by `<base64-i>` definition.

`<base64-i>` of initial-final frame, is Base64 encoded data taking form:

Content	Size	Description
total length	2 bytes	Big endian 16-bit value representing total length of body + 2 bytes for CRC16; note that size of total length field is not added to total length value.
body	no more than MTU - 7	Raw body data fragment
crc16	2 bytes	CRC16 of entire packet body, preceding length not included.

Partial frame is continuation after previous initial or other partial frame. Partial frame takes form:

Content	Size	Description
0x04 0x14	2 bytes	Frame start marker
<base64-i>	no more than MTU - 3 bytes	Base64 encoded body
0x0a	1 byte	Frame termination

The `<base64-i>` of partial frame is Base64 encoding of data, taking form:

Content	Size	Description
body	no more than MTU - 3	Raw body data fragment

The `<base64-i>` of partial-final frame is Base64 encoding of data, taking form:

Content	Size	Description
body	no more than MTU - 5	Raw body data fragment
crc16	2 bytes	CRC16 of entire packet body, preceding length not included.

CRC Details The CRC16 included in final type frames is calculated over only raw data and does not include packet length. CRC16 polynomial is 0x1021 and initial value is 0.

Raw UART/serial (without console)

In Zephyr, an alternative transport is available for UART which does not use Base64 as the SMP over console transport does, this allows for smaller code size and faster transfers - but cannot be used on devices which have shell or log output on the same UART or are connected to terminals that are displaying ASCII data, as all communication is done using the raw binary SMP protocol.

To use this protocol, use `CONFIG_MCUMGR_TRANSPORT_RAW_UART`, it requires the MCUmgr UART console driver to be enabled in raw mode.

Timeout Because there is no framing when using this transport, it is possible that there might be invalid data received on the UART which causes the system to wait for e.g. a very long packet which is not going to arrive. To prevent this, a timeout system exists whereby if the full packet is not received within a specific timeframe, the whole receive buffer is cleared. It is recommended that this option should be enabled on UART ports, for transports like USB CDC whereby it is a "virtual" UART and the data can be verified before being passed, this option is not needed. The option can be enabled with `CONFIG_MCUMGR_TRANSPORT_RAW_UART_INPUT_TIMEOUT`, the timeout for this can be set with `CONFIG_MCUMGR_TRANSPORT_RAW_UART_INPUT_TIMEOUT_TIME_MS`.

API Reference

SMP transport

4.9.6 Device Firmware Upgrade

Overview

The Device Firmware Upgrade subsystem provides the necessary frameworks to upgrade the image of a Zephyr-based application at run time. It currently consists of two different modules:

- subsys/dfu/boot/: Interface code to bootloaders
- subsys/dfu/img_util/: Image management code

The DFU subsystem deals with image management, but not with the transport or management protocols themselves required to send the image to the target device. For information on these protocols and frameworks please refer to the *Device Management* section.

Flash Image The flash image API as part of the Device Firmware Upgrade (DFU) subsystem provides an abstraction on top of Flash Stream to simplify writing firmware image chunks to flash.

API Reference
Flash image API

MCUBoot API The MCUboot API is provided to get version information and boot status of application images. It allows to select application image and boot type for the next boot.

API Reference
MCUboot image control API

Bootloaders

MCUboot Zephyr is directly compatible with the open source, cross-RTOS MCUboot boot loader. It interfaces with MCUboot and is aware of the image format required by it, so that Device Firmware Upgrade is available when MCUboot is the boot loader used with Zephyr. The source code itself is hosted in the MCUboot GitHub Project page.

In order to use MCUboot with Zephyr you need to take the following into account:

1. You will need to define the flash partitions required by MCUboot; see *Flash map* for details.
2. You will have to specify your flash partition as the chosen code partition

```
/ {
   chosen {
      zephyr,code-partition = &slot0_partition;
   };
};
```

3. Your application's `.conf` file needs to enable the `CONFIG_BOOTLOADER_MCUBOOT` Kconfig option in order for Zephyr to be built in an MCUboot-compatible manner
4. You need to build and flash MCUboot itself on your device
5. You might need to take precautions to avoid mass erasing the flash and also to flash the Zephyr application image at the correct offset (right after the bootloader)

More detailed information regarding the use of MCUboot with Zephyr can be found in the MCUboot with Zephyr documentation page on the MCUboot website.

4.9.7 Over-the-Air Update

Overview

Over-the-Air (OTA) Update is a method for delivering firmware updates to remote devices using a network connection. Although the name implies a wireless connection, updates received over a wired connection (such as Ethernet) are still commonly referred to as OTA updates. This approach requires server infrastructure to host the firmware binary and implement a method of signaling when an update is available. Security is a concern with OTA updates; firmware binaries should be cryptographically signed and verified before upgrading.

The *Device Firmware Upgrade* section discusses upgrading Zephyr firmware using MCUboot. The same method can be used as part of OTA. The binary is first downloaded into an unoccupied code partition, usually named `slot1_partition`, then upgraded using the *MCUboot* process.

Examples of OTA

Golioth Golioth is an IoT management platform that includes OTA updates. Devices are configured to observe your available firmware revisions on the Golioth Cloud. When a new version is available, the device downloads and flashes the binary. In this implementation, the connection between cloud and device is secured using TLS/DTLS, and the signed firmware binary is confirmed by MCUboot before the upgrade occurs.

1. A working sample can be found on the Golioth Firmware SDK repository
2. The Golioth OTA documentation includes complete information about the versioning process

Plain HTTP download The *Firmware Over-the-Air over HTTP* library fetches a signed image from any HTTP server, writes it into the secondary slot and requests the swap. No management protocol or dedicated update server is needed. MCUboot verifies the image signature before the upgrade occurs.

There is a fota-http sample included in the Zephyr mgmt section.

Eclipse hawkBit™ Eclipse hawkBit™ is an update server framework that uses polling on a REST api to detect firmware updates. When a new update is detected, the binary is downloaded and installed. MCUboot can be used to verify the signature before upgrading the firmware.

There is a hawkbit-api sample included in the Zephyr mgmt section.

UpdateHub UpdateHub is a platform for remotely updating embedded devices. Updates can be manually triggered or monitored via polling. When a new update is detected, the binary is downloaded and installed. MCUboot can be used to verify the signature before upgrading the firmware.

There is an updatehub-fota sample included in the Zephyr mgmt section.

SMP Server A Simple Management Protocol (SMP) server can be used to update firmware via Bluetooth Low Energy (LE) or UDP. *MCUmgr* is used to send a signed firmware binary to the remote device where it is verified by MCUboot before the upgrade occurs.

There is an smp-svr sample included in the Zephyr mgmt section.

Lightweight M2M (LwM2M) The *Lightweight M2M (LwM2M)* protocol includes support for firmware update via `CONFIG_LWM2M_FIRMWARE_UPDATE_OBJ_SUPPORT`. Devices securely connect to an LwM2M server using DTLS. A lwm2m-client sample is available but it does not demonstrate the firmware update feature.

mender-mcu mender-mcu enables robust firmware updates on resource-constrained devices by integrating with Zephyr. It implements an Update Module interface and provides a default Update Module that integrates with MCUboot to provide A/B updates. This allows microcontroller units (MCUs) to perform atomic, fail-safe OTA updates with automatic rollback on failure.

See external_module_mender_mcu for integration details and examples.

Memfault and nRF Cloud powered by Memfault Memfault is a IoT observability platform that includes OTA management. Devices check-in with Memfault's service periodically for an OTA update, and when an update is available, download and install the binary.

See external_module_memfault_firmware_sdk for overall integration details and examples.

4.9.8 Firmware Over-the-Air over HTTP

Overview

The FOTA HTTP library downloads a firmware image from an HTTP server and writes it straight into the MCUboot slot that is not currently running. It then lets the application request the swap, confirm the new image or erase the slot. Unlike *hawkBit or UpdateHub*, no device management protocol is involved: any web server able to serve a file can act as the update source, which makes it a good fit for simple deployments and for bench testing.

The image is streamed to flash through the *Flash Image* as it arrives, so its size is not limited by RAM. The slot is erased progressively during the write, and the MCUboot trailer of the slot is cleared before the first byte lands, so a leftover from a previous image can never be mistaken for a pending swap.

After the download the library reads the MCUboot header back from flash and fails with `-ENOEXEC` when the data is not an image. It does not verify the signature: MCUboot does that before the image is ever booted, and that check decides whether the image runs. An optional SHA-256 comparison against a digest supplied by the caller is available with `CONFIG_FOTA_HTTP_SHA256_CHECK` to catch a corrupted transfer before the reboot. An image rejected by that comparison, or by the downgrade check below, is erased from the slot. After a download that failed part way, `fota_http_apply()` refuses the slot until another download into it succeeds, while the partial data stays in place for a resume.

Update cycle

The typical sequence is:

1. `fota_http_download()` or `fota_http_download_async()` fetches the image into the secondary slot. The optional `fota_http_download_params` members `sha256`, `resume` and the TLS ones only exist when their Kconfig option is enabled.
2. `fota_http_apply()` marks the image for a test boot, or as permanent.
3. The device reboots and MCUboot swaps the image in.
4. On a test boot the application validates itself and calls `fota_http_confirm()`. Without that call, the next reboot makes MCUboot revert to the previous image.

The revert step requires an MCUboot mode with revert support, such as swap-using-move or swap-using-offset. In overwrite-only mode the previous image is gone as soon as the swap happens and confirmation has no effect, so check which mode the board's sysbuild configuration selects.

While a test image runs, the secondary slot holds the image MCUboot would revert to, so `fota_http_download()` refuses with `-EPERM` until `fota_http_confirm()` has been called.

Threads and stack

`fota_http_download()` blocks the calling thread for the whole transfer and needs enough stack for the HTTP client and, with https, the TLS handshake. With `CONFIG_FOTA_HTTP_ASYNC`, `fota_http_download_async()` runs the same transfer on a thread owned by the library, sized by `CONFIG_FOTA_HTTP_THREAD_STACK_SIZE`, and reports the result through a completion callback. The shell command selects the asynchronous variant so the shell stack size does not matter. Applications that only call `fota_http_download()` leave it disabled and do not pay for the thread stack.

Only one download runs at a time. `fota_http_cancel()` aborts the transfer in progress when the next fragment arrives or the request times out.

Redirects and resume

Responses with status 301, 302, 303, 307 or 308 are followed up to `CONFIG_FOTA_HTTP_MAX_REDIRECTS` times. Both absolute and path-only `Location` headers are accepted.

With `CONFIG_FOTA_HTTP_RESUME` the download offset is saved in settings while the transfer runs and when it fails. A later download with the `resume` parameter set continues from that offset with an HTTP `Range` request. The offset is only reused when the URL and image index match the interrupted download, and when the server sent an `ETag` or `Last-Modified` header the request carries `If-Range` with it, so a file that changed on the server is fetched again from the start. A server that ignores the `Range` header and answers with the full file restarts the transfer from zero, and so does a `416 Range Not Satisfiable` answer, which is what a server sends when the file grew shorter than the saved offset. A validator too long to store also drops the saved progress, so the next download starts from the beginning rather than resuming against a file that may have changed. The offset is cleared once the image is complete.

Multiple images

In a layout with more than one updateable image, the `image_index` parameter selects which secondary slot the download goes to. `fota_http_apply()` and `fota_http_confirm()` both take the same index, so a layout that carries firmware for a second device confirms only what this device validated. When several images must run together, download and apply every one of them before rebooting: MCUboot validates the whole set and swaps it in one go, and confirm each one afterwards, because an image left unconfirmed is reverted on its own. `fota_http_confirm_pending()` only reports image 0, so an application that updates several images tracks the rest itself. A downgrade check with `CONFIG_FOTA_HTTP_REJECT_DOWNGRADE` only covers image 0.

With direct-XIP bootloaders the slot an image runs from is fixed at link time, so the device must fetch the image variant built for the free slot. `fota_http_get_download_slot()` reports which slot that is. This only covers single-image layouts: for every image after the first the library always uploads to the odd-numbered slot of the pair, which is correct for the swap and overwrite modes but not for direct-XIP.

Downgrade protection

`CONFIG_FOTA_HTTP_REJECT_DOWNGRADE` compares the version in the downloaded header with the running image and fails the download when it is older, erasing the rejected image so it cannot be applied by mistake. This is a convenience check for the application. An enforced policy belongs in the bootloader, see the MCUboot downgrade prevention and hardware rollback protection options.

TLS

`https` URLs are accepted when `CONFIG_FOTA_HTTP_TLS` is enabled. The CA certificate of the server is looked up under `CONFIG_FOTA_HTTP_TLS_SEC_TAG`, or under the tags passed in

the download parameters, and can be registered with `fota_http_tls_add_ca()` or directly with `tls_credential_add()`. For mutual TLS, register the client certificate and key with `fota_http_tls_add_client_cert()`.

Peer verification is required by default and `CONFIG_FOTA_HTTP_TLS_PEER_VERIFY` should stay at its default in production. The host name from the URL is verified against the certificate. When the URL carries an IP address and the certificate has no matching entry, pass the expected name in the `tls_hostname` parameter.

Certificate validity dates are only enforced when `CONFIG_MBEDTLS_HAVE_TIME_DATE` is enabled, which requires a correct wall clock on the device before the first download. TLS 1.3 can be selected with `CONFIG_FOTA_HTTP_TLS_VERSION_1_3`.

Shell commands

With `CONFIG_FOTA_HTTP_SHELL` enabled, a `fota` command exposes the library from the console:

```
fota download <url> [resume] [image=<n>]   download an image
fota cancel                                abort the download in progress
fota apply [permanent] [image=<n>]         boot the downloaded image on the next␣
↪reset
fota confirm [image]                       confirm the running image
fota status                                show images and confirmation state
fota erase [image]                         erase the secondary slot
```

See the fota-http sample for a complete walkthrough.

API Reference

FOTA HTTP

Related code samples

- Firmware Over-the-Air over HTTPDownload a signed firmware image over HTTP into the MCUboot secondary slot and swap to it.

4.9.9 EC Host Command

Overview

The host command protocol defines the interface for a host, or application processor, to communicate with a target embedded controller (EC). The EC Host command subsystem implements the target side of the protocol, generating responses to commands sent by the host. The host command protocol interface supports multiple versions, but this subsystem implementation only support protocol version 3.

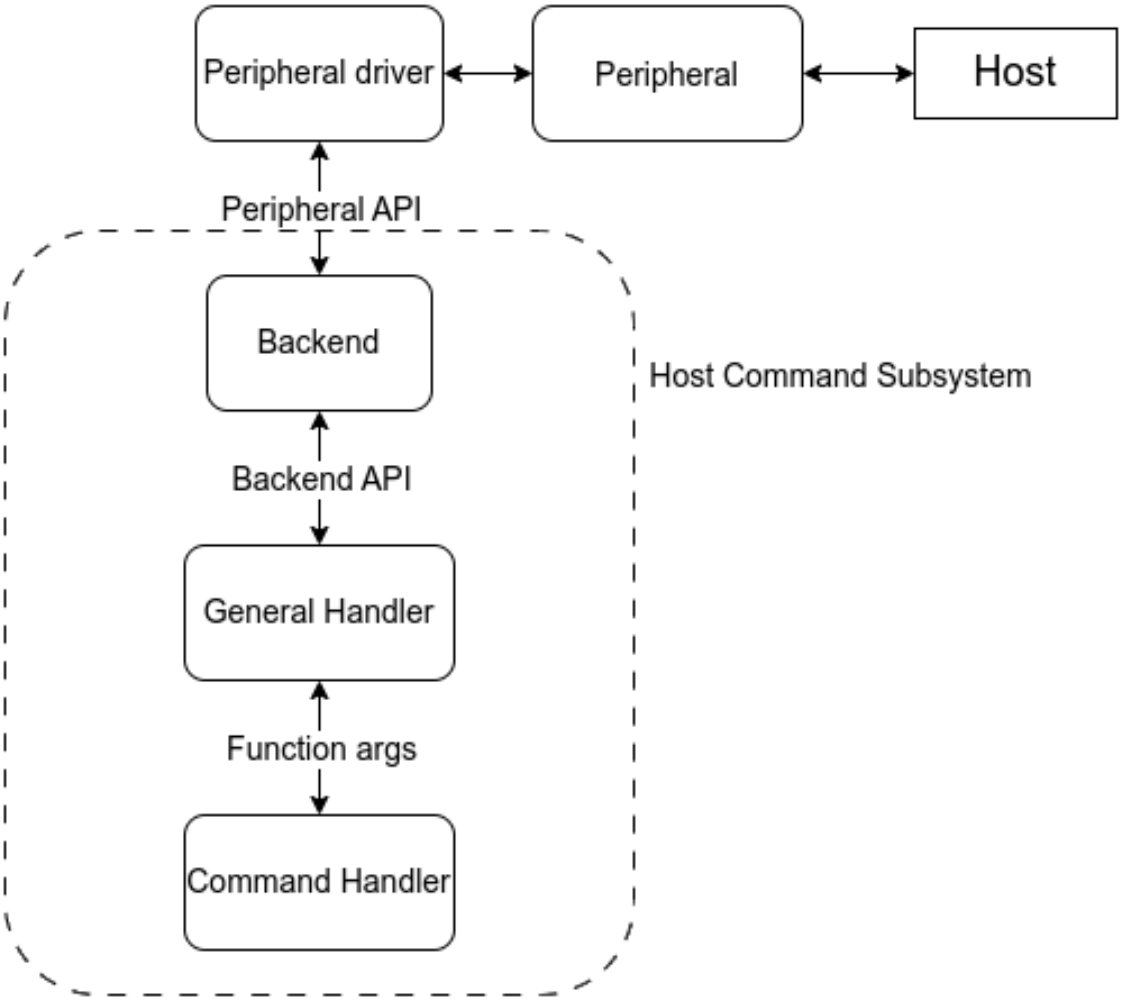
Architecture

The Host Command subsystem contains a few components:

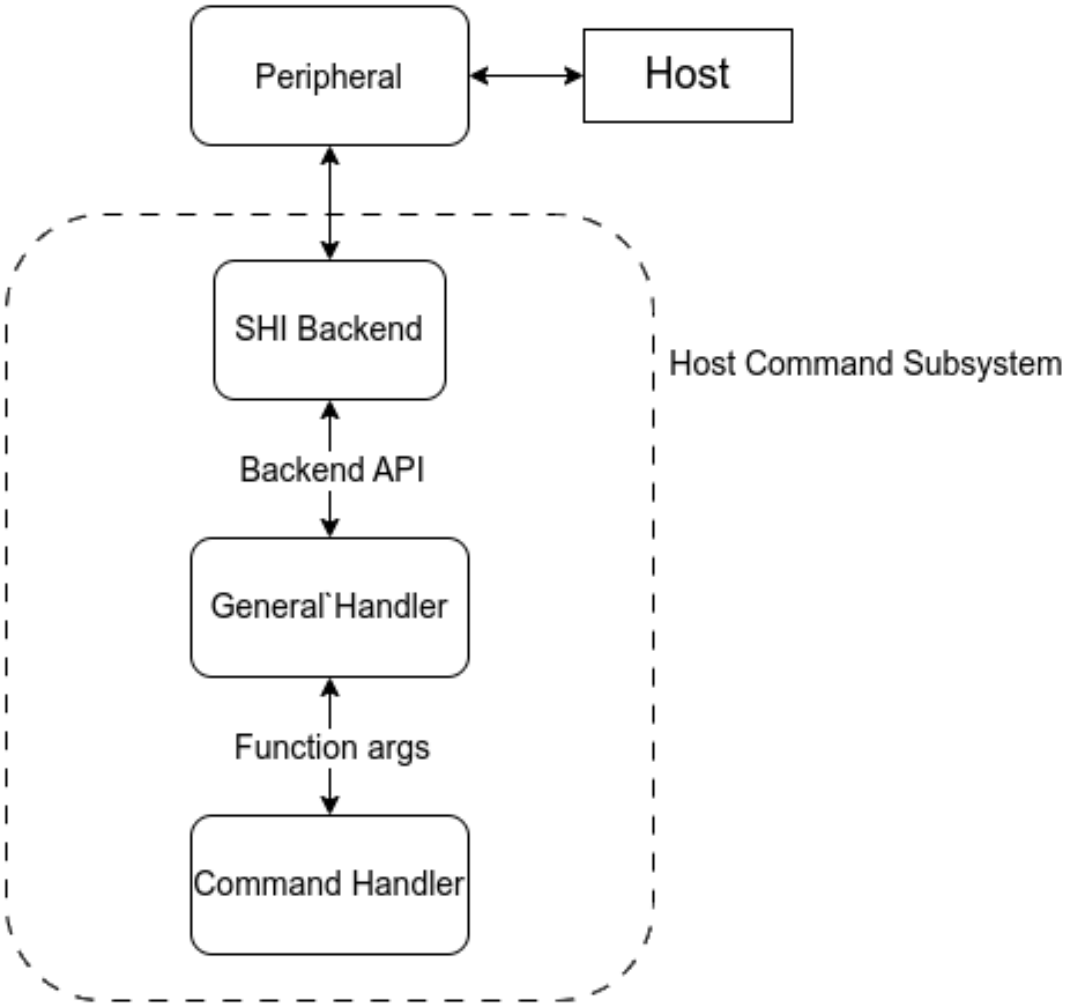
- Backend
- General handler
- Command handler

The backend is a layer between a peripheral driver and the general handler. It is responsible for sending and receiving commands via chosen peripheral.

The general handler validates data from the backend e.g. check sizes, checksum, etc. If the command is valid and the user has provided a handler for a received command id, the command handler is called.



SHI (Serial Host Interface) is different to this because it is used only for communication with a host. SHI does not have API itself, thus the backend and peripheral driver layers are combined into one backend layer.



Another case is SPI. Unfortunately, the current SPI API can't be used to handle the host commands communication. The main issues are unknown command size sent by the host (the SPI transaction sends/receives specific number of bytes) and need to constant sending status byte (the SPI module is enabled and disabled per transaction). It forces implementing the SPI driver within a backend, as it is done for SHI. That means a SPI backend has to be implemented per chip family. However, it can be changed in the future once the SPI API is extended to host command needs. Please check the discussion.

That approach requires configuring the SPI dts node in a special way. The main compatible string of a SPI node has changed to use the Host Command version of a SPI driver. The rest of the properties should be configured as usual. Example of the SPI node for STM32:

```
&spi1 {
        /* Change the compatible string to use the Host Command version of the
         * STM32 SPI driver
         */
        compatible = "st,stm32-spi-host-cmd";
        status = "okay";

        dmas = <&dma2 3 3 0x38440 0x03>,
             <&dma2 0 3 0x38480 0x03>;
        dma-names = "tx", "rx";
        /* This field is used to point at our CS pin */
```

(continues on next page)

(continued from previous page)

```
        cs-gpios = <&gpioa 4 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>;
};
```

The STM32 SPI host command backend driver supports the `st,stm32h7-spi` and `st,stm32-spi-fifo` variant implementations. To enable these variants, append the corresponding compatible string. For example, to enable FIFO support and support for the STM32H7 SoCs, modify the compatible string as shown.

```
&spi1 {
    compatible = "st,stm32h7-spi", "st,stm32-spi-fifo", "st,stm32-spi-host-cmd";
    ...
};
```

The chip that runs Zephyr is a SPI slave and the `cs-gpios` property is used to point our CS pin. For the SPI, it is required to set the backend chosen node `zephyr,host-cmd-spi-backend`.

The supported backend and peripheral drivers:

- Simulator
- SHI - ITE and NPCX
- eSPI - any eSPI slave driver that support `CONFIG_ESPI_PERIPHERAL_EC_HOST_CMD` and `CONFIG_ESPI_PERIPHERAL_CUSTOM_OPCODE`
- UART - any UART driver that supports the asynchronous API
- SPI - STM32

Initialization

If the application configures one of the following backend chosen nodes and `CONFIG_EC_HOST_CMD_INITIALIZE_AT_BOOT` is set, then the corresponding backend initializes the host command subsystem by calling `ec_host_cmd_init()`:

- `zephyr,host-cmd-espi-backend`
- `zephyr,host-cmd-shi-backend`
- `zephyr,host-cmd-uart-backend`
- `zephyr,host-cmd-spi-backend`

If no backend chosen node is configured, the application must call the `ec_host_cmd_init()` function directly. This way of initialization is useful if a backend is chosen in runtime based on e.g. GPIO state.

Buffers

The host command communication requires buffers for rx and tx. The buffers are be provided by the general handler if `CONFIG_EC_HOST_CMD_HANDLER_RX_BUFFER_SIZE` > 0 for rx buffer and `CONFIG_EC_HOST_CMD_HANDLER_TX_BUFFER_SIZE` > 0 for the tx buffer. The shared buffers are useful for applications that use multiple backends. Defining separate buffers by every backend would increase the memory usage. However, some buffers can be defined by a peripheral driver e.g. eSPI. These ones should be reused as much as possible.

Logging

The host command has an embedded logging system of the ongoing communication. The are a few logging levels:

- `LOG_INF` is used to log a command id of a new command and not success responses. Repeats of the same command are not logged

- `LOG_DBG` logs every command, even repeats
- `LOG_DBG` + `CONFIG_EC_HOST_CMD_LOG_DBG_BUFFERS` logs every command and responses with the data buffers

API Reference

EC Host Command Interface

4.9.10 Binary Descriptors

Binary Descriptors are constant data objects storing information about the binary executable. Unlike "regular" constants, binary descriptors are linked to a known offset in the binary, making them accessible to other programs, such as a different image running on the same device or a host tool. A few examples of constants that would make useful binary descriptors are: kernel version, app version, build time, compiler version, environment variables, compiling host name, etc.

Binary descriptors are created by using the `DEFINE_BINDESC_*` macros. For example:

```
#include <zephyr/bindesc.h>

BINDESC_STR_DEFINE(my_string, 2, "Hello world!"); // Unique ID is 2
```

`my_string` could then be accessed using:

```
printk("my_string: %s\n", BINDESC_GET_STR(my_string));
```

But it could also be retrieved by `west bindesc`:

```
$ west bindesc custom_search STR 2 build/zephyr/zephyr.bin
"Hello world!"
```

Internals

Binary descriptors are implemented with a TLV (tag, length, value) header linked to a known offset in the binary image. This offset may vary between architectures, but generally the descriptors are linked as close to the beginning of the image as possible. In architectures where the image must begin with a vector table (such as ARM), the descriptors are linked right after the vector table. The reset vector points to the beginning of the text section, which is after the descriptors. In architectures where the image must begin with executable code (e.g. x86), a jump instruction is injected at the beginning of the image, in order to skip over the binary descriptors, which are right after the jump instruction.

Each tag is a 16 bit unsigned integer, where the most significant nibble (4 bits) is the type (currently uint, string or bytes), and the rest is the ID. The ID is globally unique to each descriptor. For example, the ID of the app version string is `0x800`, and a string is denoted by 0x1, making the app version tag `0x1800`. The length is a 16 bit number equal to the length of the data in bytes. The data is the actual descriptor value. All binary descriptor numbers (magic, tags, uints) are laid out in memory in the endianness native to the SoC. `west bindesc` assumes little endian by default, so if the image belongs to a big endian SoC, the appropriate flag should be given to the tool.

The binary descriptor header starts with the magic number `0xb9863e5a7ea46046`. It's followed by the TLVs, and ends with the `DESCRIPTORS_END` (`0xffff`) tag. The tags are always aligned to 32 bits. If the value of the previous descriptor had a non-aligned length, zero padding will be added to ensure that the current tag is aligned.

Putting it all together, here is what the example above would look like in memory (of a little endian SoC):

```
 46 60 a4 7e 5a 3e 86 b9 02 10  0d 00  48 65 6c 6c 6f 20 77 6f 72 6c 64 21 00 00␣
↪00 00 ff ff 00 00
|         magic         | tag |length| H  e  l  l  o     w  o  r  l  d  !    |  ␣
↪pad  |    end    |
```

Usage

Binary descriptors are always created by the `BINDESC_*_DEFINE` macros. As shown in the example above, a descriptor can be generated from any string or integer, with any ID. However, it is recommended to comply with the standard tags defined in `include/zephyr/bindesc.h`, as that would have the following benefits:

1. The `west bindesc` tool would be able to recognize what the descriptor means and print a meaningful tag
2. It would enforce consistency between various apps from various sources
3. It allows upstream-ability of descriptor generation (see Standard Descriptors)

To define a descriptor with a standard tag, just use the tags included from `bindesc.h`:

```
#include <zephyr/bindesc.h>

BINDESC_STR_DEFINE(app_version, BINDESC_ID_APP_VERSION_STRING, "1.2.3");
```

Standard Descriptors Some descriptors might be trivial to implement, and could therefore be implemented in a standard way in upstream Zephyr. These could then be enabled via Kconfig, instead of requiring every user to reimplement them. These include build times, kernel version, and host info. For example, to add the build date and time as a string, the following configs should be enabled:

```
# Enable binary descriptors
CONFIG_BINDESC=y

# Enable definition of binary descriptors
CONFIG_BINDESC_DEFINE=y

# Enable default build time binary descriptors
CONFIG_BINDESC_DEFINE_BUILD_TIME=y
CONFIG_BINDESC_BUILD_DATE_TIME_STRING=y
```

To avoid collisions with user defined descriptors, the standard descriptors were allotted the range between `0x800-0xfff`. This leaves `0x000-0x7ff` to users. For more information read the `help` sections of these Kconfig symbols. By convention, each Kconfig symbol corresponds to a binary descriptor whose name is the Kconfig name (with `CONFIG_BINDESC_` removed) in lower case. For example, `CONFIG_BINDESC_KERNEL_VERSION_STRING` creates a descriptor that can be accessed using `BINDESC_GET_STR(kernel_version_string)`.

Reading Descriptors It's also possible to read and parse binary descriptors from an application. This can be useful both for an image trying to read its own descriptors, and for an image trying to read another image's descriptors. Reading can be performed through one of three backends:

1. RAM - assuming the descriptors have been copied to RAM (e.g. by a bootloader), they can be read from the buffer they reside in.
2. Memory mapped flash - If the flash where the image to be read resides in flash and is accessible through the program's address space, it can be read directly from flash. This option

uses the least amount of RAM, but will not work if the flash is not memory mapped, and is not recommended to read a bootloader's descriptors for security concerns.

3. Flash - Using an internal buffer, the descriptors are read one by one using the flash API, and given to the user while they're in the buffer.

To enable reading descriptors, enable `CONFIG_BINDESC_READ`. The three backends are enabled by these Kconfig symbols, respectively: `CONFIG_BINDESC_READ_RAM`, `CONFIG_BINDESC_READ_MEMORY_MAPPED_FLASH`, and `CONFIG_BINDESC_READ_FLASH`.

To read the descriptors, a handle to the descriptors should first be initialized:

```
struct bindesc_handle handle;

/* Assume buffer holds a copy of the descriptors */
bindesc_open_ram(&handle, buffer);
```

The `bindesc_open_*` functions are the only functions concerned with the backend used. The rest of the API is agnostic to where the data is. After the handle has been initialized, it can be used with the rest of the API:

```
char *version;
bindesc_find_str(&handle, BINDESC_ID_KERNEL_VERSION_STRING, &version);
printk("Kernel version: %s\n", version);
```

west bindesc tool `west` is able to parse and display binary descriptors from a given executable image.

For more information refer to `west bindesc --help` or the *documentation*.

API Reference

Binary Descriptor Definition

Related code samples

- Binary descriptors "Hello World"Set and access binary descriptors for a basic Zephyr application.

Binary Descriptor Reading

Related code samples

- Binary descriptors readDefine some binary descriptors and read them.

Algorithms & Data

4.10 Algorithms & Data

This section covers services for transforming, encoding, and validating data, including checksums, digital signal processing, and serialization.

4.10.1 CRC

Overview

The CRC subsystem provides software implementations of various Cyclic Redundancy Check algorithms for data integrity verification. CRCs are commonly used to detect accidental changes to data in storage and communication systems.

The subsystem offers a comprehensive set of CRC algorithms including CRC-4, CRC-7, CRC-8, CRC-16, CRC-24, and CRC-32 variants, and provides optional hardware acceleration support when available.

> **Note**
>
> This library is distinct from the *CRC hardware driver API*, which provides an interface to hardware CRC acceleration peripherals.
>
> When a hardware CRC unit is available and the `zephyr,crc` property has been set in the `/chosen` node in Devicetree, the library functions will default to using hardware acceleration for improved performance (can be disabled by setting `CONFIG_CRC_HW_HANDLER` to `n`).

Usage To compute a CRC, include the appropriate header and call the desired function:

```
#include <zephyr/sys/crc.h>

uint8_t data[] = {0x01, 0x02, 0x03, 0x04};
uint32_t checksum = crc32_ieee(data, sizeof(data));
```

For streaming data processed in chunks, use the "update" variants:

```
uint32_t crc = 0;
crc = crc32_ieee_update(crc, chunk1, len1);
crc = crc32_ieee_update(crc, chunk2, len2);
/* Final CRC value is in 'crc' */
```

The generic `crc_by_type()` function provides a unified interface to select the CRC algorithm at runtime.

Configuration

Related configuration options:

- `CONFIG_CRC` - Enable CRC support
- `CONFIG_CRC_HW_HANDLER` - Enable hardware CRC acceleration
- `CONFIG_CRC_SHELL` - Enable CRC shell commands
- `CONFIG_CRC[0-9].*` - Enable software implementations of specific CRC algorithms

API Reference

CRC

> **Related code samples**
>
> - Cyclic Redundancy Check Subsystem (CRC Subsys)Compute and verify a CRC computation using the CRC subsys API.

4.10.2 Digital Signal Processing (DSP)

- *Using zDSP*
- *Optimizing for your architecture*
- *API Reference*

The DSP API provides an architecture agnostic way for signal processing. Currently, the API will work on any architecture but will likely not be optimized. The status of the various architectures can be found below:

Architecture	Status
ARC	Optimized
ARM	Optimized
ARM64	Optimized
MIPS	Unoptimized
POSIX	Unoptimized
RISCV	Unoptimized
RISCV64	Unoptimized
SPARC	Unoptimized
X86	Unoptimized
XTENSA	Unoptimized

Using zDSP

zDSP provides various backend options which are selected automatically for the application. By default, including the CMSIS module will enable all architectures to use the zDSP APIs. This can be done by setting:

```
CONFIG_CMSIS_DSP=y
```

If your application requires some additional customization, it's possible to enable `CONFIG_DSP_BACKEND_CUSTOM` which means that the application is responsible for providing the implementation of the zDSP library.

Optimizing for your architecture

If your architecture is showing as `Unoptimized`, it's possible to add a new zDSP backend to better support it. To do that, a new Kconfig option should be added to `subsys/dsp/Kconfig` along with the required dependencies and the `default` set for `DSP_BACKEND` Kconfig choice.

Next, the implementation should be added at `subsys/dsp/<backend>/` and linked in at `subsys/dsp/CMakeLists.txt`. To add architecture-specific attributes, its corresponding Kconfig option should be added to `subsys/dsp/Kconfig` and use them to update `DSP_DATA` and `DSP_STATIC_DATA` in `include/zephyr/dsp/dsp.h`.

API Reference

DSP (Digital Signal Processing)

4.10.3 Serialization

Zephyr has support for several data serialization subsystems. These can be used to encode/decode structured data with a known format on-the-wire.

CBOR

CBOR (Concise Binary Object Representation) is a data format whose design goals include the possibility of extremely small code size, fairly small message size, and extensibility without the need for version negotiation.

Zephyr provides support for CBOR via the zcbor library, which is pulled in as a West module.

Configuration To enable CBOR support, enable the `CONFIG_ZCBOR` Kconfig option.

API Reference The zcbor library provides its own API documentation, please refer to it for more information.

JSON

Zephyr provides a JSON library that can be used to encode and decode JSON data.

Usage

Defining the Data Structure First, define the C structure that corresponds to the JSON object and the descriptor that maps the structure fields to JSON tokens.

```
#include <zephyr/data/json.h>

struct foo {
    int bar;
    const char *baz;
};

static const struct json_obj_descr foo_descr[] = {
    JSON_OBJ_DESCR_PRIM(struct foo, bar, JSON_TOK_NUMBER),
    JSON_OBJ_DESCR_PRIM(struct foo, baz, JSON_TOK_STRING),
};
```

Encoding To encode a C structure into a JSON string, use `json_obj_encode_buf()`.

```
void encode_example(void)
{
    struct foo data = { .bar = 42, .baz = "hello" };
    char buffer[128];
    int ret;

    ret = json_obj_encode_buf(foo_descr, ARRAY_SIZE(foo_descr),
                              &data,
                              buffer, sizeof(buffer));
    if (ret < 0) {
        /* handle error */
    }
}
```

Decoding To decode a JSON string into a C structure, use `json_obj_parse()`.

```
void decode_example(void)
{
    struct foo data;
```

(continues on next page)

(continued from previous page)

```
    const char *json_text = "{\"bar\": 42, \"baz\": \"hello\"}";
    int ret;

    ret = json_obj_parse(json_text, strlen(json_text),
                         foo_descr, ARRAY_SIZE(foo_descr),
                         &data);
    if (ret < 0) {
        /* handle error */
    }
}
```

Configuration To enable JSON support, enable the `CONFIG_JSON_LIBRARY` Kconfig option.

API Reference
JSON

Related code samples

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- Eclipse hawkBit Direct Device Integration APIUpdate a device using Eclipse hawkBit DDI API.
- HTTP ServerImplement an HTTP(s) Server demonstrating various resource types.
- Secure MQTT Sensor/ActuatorImplement an MQTT-based IoT sensor/actuator device

Nanopb

Nanopb is a C implementation of Google's Protocol Buffers.

Requirements Nanopb uses the protocol buffer compiler to generate source and header files, make sure the `protoc` executable is installed and available.

Ubuntu

Use `apt` to install dependency:

```
sudo apt install protobuf-compiler
```

macOS

Use `brew` to install dependency:

```
brew install protobuf
```

Windows

Use `choco` to install dependency:

```
choco install protoc
```

Configuration Make sure to include `nanopb` within your `CMakeLists.txt` file as follows:

```
list(APPEND CMAKE_MODULE_PATH ${ZEPHYR_BASE}/modules/nanopb)
include(nanopb)
```

Adding `proto` files can be done with the `zephyr_nanopb_sources()` CMake function which ensures the generated header and source files are created before building the specified target.

Nanopb has generator options that can be used to configure messages or fields. This allows to set fixed sizes or skip fields entirely.

The internal CMake generator has an extension to configure `*.options.in` files automatically with CMake variables.

See samples/modules/nanopb/src/simple.options.in and samples/modules/nanopb/CMakeLists.txt for usage example.

Frameworks

4.11 Frameworks

This section covers high-level frameworks and libraries that help structure application logic, load code at runtime, stream media, and model sensors.

4.11.1 Linkable Loadable Extensions (LLEXT)

The LLEXT subsystem provides a toolbox for extending the functionality of an application at runtime with linkable loadable code.

Extensions are precompiled executables in ELF format that can be verified, loaded, and linked with the main Zephyr binary. Extensions can be manipulated and introspected to some degree, as well as unloaded when no longer needed.

Configuration

The following Kconfig options are available for the LLEXT subsystem.

Harvard architecture `CONFIG_HARVARD`

Architecture uses separate instruction and data memory.

`CONFIG_HARVARD` is not a Kconfig defined by the LLEXT subsystem. Instead, it must be defined and selected by the board or SoC to signal that LLEXT should be built with Harvard architecture support. The board or SoC must also implement `arch_is_instr_mem()`.

Heap size The LLEXT subsystem needs a heap to be allocated for extension related data. The following option controls this allocation, when allocating a static heap.

`CONFIG_LLEXT_HEAP_SIZE`

Size of the LLEXT heap in kilobytes.

For boards using the Harvard architecture, the LLEXT heap is split into two: one heap in instruction memory and another in data memory. The following options control these allocations.

`CONFIG_LLEXT_INSTR_HEAP_SIZE`

Size of the LLEXT heap in instruction memory in kilobytes.

`CONFIG_LLEXT_DATA_HEAP_SIZE`

Size of the LLEXT heap in data memory in kilobytes.

Alternatively the application can configure a dynamic heap using the following option.

`CONFIG_LLEXT_HEAP_DYNAMIC`

Some applications require loading extensions into the memory which does not exist during the boot time and cannot be allocated statically. Make the application responsible for LLEXT heap allocation. Do not allocate LLEXT heap statically.

Application must call `llext_heap_init()` in order to assign a buffer to be used as the LLEXT heap, otherwise LLEXT modules will not load. When the application does not need LLEXT functionality any more, it should call `llext_heap_uninit()` which releases control of the buffer back to the application.

Note

When *user mode* is enabled, the heap size must be large enough to allow the extension sections to be allocated with the alignment required by the architecture.

Note

On Harvard architectures, applications must call `llext_heap_init_harvard()`.

The backing data structure for the LLEXT heap is by default a `k_heap`, but it can be changed to `sys_mem_blocks_t` for non-metadata (i.e., extension regions) by selecting `CONFIG_LLEXT_HEAP_MEMBLK` for `CONFIG_LLEXT_HEAP_MANAGEMENT`.

`CONFIG_LLEXT_HEAP_MANAGEMENT`

Select the memory management API used to store extension regions in LLEXT heap memory. This choice does not affect LLEXT metadata, which is always managed with `k_heap`.

`CONFIG_LLEXT_HEAP_MEMBLK`

Use `sys_mem_blocks_t` API to manage LLEXT heap memory for extension regions. A minimum of one block will be allocated per region. Block size must be selected with care to ensure proper alignment for extension regions.

Note

`CONFIG_LLEXT_HEAP_MEMBLK` does not support `CONFIG_LLEXT_HEAP_DYNAMIC`.

The heap will be split into two, with the size of each sub-heap controlled by the following options.

`CONFIG_LLEXT_EXT_HEAP_SIZE`

Heap size in kilobytes available for LLEXT extension sections. Must be a multiple of `CONFIG_LLEXT_HEAP_MEMBLK_BLOCK_SIZE`. Replaced by `CONFIG_LLEXT_INSTR_HEAP_SIZE` and `CONFIG_LLEXT_DATA_HEAP_SIZE` if `CONFIG_HARVARD` is selected.

`CONFIG_LLEXT_METADATA_HEAP_SIZE`

Heap size in kilobytes available for LLEXT metadata.

Another option controls block size.

`CONFIG_LLEXT_HEAP_MEMBLK_BLOCK_SIZE`

Block size in bytes for LLEXT `sys_mem_blocks_t` heap(s). Must be equal to or a multiple of `LLEXT_PAGE_SIZE` if MMU or MPU are enabled. The block size must also be equal to or a multiple of the largest alignment needed for any extension region. If `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT` is selected and regions are large, an unreasonably large block size may be needed to satisfy alignment requirements.

Heap placement The LLEXT heap(s) have custom sections. Non-Harvard heap sections (`.llext_heap`, or `.llext_metadata_heap` and `.llext_ext_heap` if `CONFIG_LLEXT_HEAP_MEMBLK` is selected) are placed alongside `.noinit` sections in the file `include/zephyr/linker/common-noinit.ld`. If none of your linker scripts include this file, you will need to place the non-Harvard LLEXT heap sections manually. One way to do this is by including `snippets-noinit.ld` in your linker script after your `.noinit` sections.

```
/* Located in generated directory. This file is populated by the
 * zephyr_linker_sources() CMake function.
 */
#include <snippets-noinit.ld>
```

Add the file as a linker source in your board, SoC, or architecture `CMakeFiles.txt`.

```
zephyr_linker_sources(NOINIT snippets-noinit.ld)
```

Then create a file in the same directory as your `CMakeFiles.txt` named `noinit.ld`.

```
#if defined(CONFIG_LLEXT) && !defined(CONFIG_LLEXT_CUSTOM_HEAP_PLACEMENT)
*(.llext_heap)
*(.llext_ext_heap)
*(.llext_metadata_heap)
#endif /* CONFIG_LLEXT && !CONFIG_LLEXT_CUSTOM_HEAP_PLACEMENT */
```

For ARC, the Harvard instruction and data heap sections (`.llext_instr_heap` and `.llext_data_heap`) are placed in instruction and data memory at the architecture level. If you are using a non-ARC board with a Harvard architecture, you will need to manually place `.llext_instr_heap` and `.llext_data_heap`.

Warning

LLEXT will be unable to load extensions if the instruction memory `.llext_instr_heap` is placed in is not writable at the time the extensions are loaded and linked.

Placements can also be specified by providing a custom linker script.

`CONFIG_CUSTOM_LINKER_SCRIPT`

Path to the linker script to be used instead of the one defined by the board.

The linker script must be based on a version provided by Zephyr since the kernel can expect a certain layout/certain regions.

This is useful when an application needs to add sections into the linker script and avoid having to change the script provided by Zephyr.

While using a custom linker script, you may need to override default placements. For example, you may wish to include `include/zephyr/linker/common-noinit.ld` in your linker script but place the heap section(s) elsewhere. To do this, select the following option.

`CONFIG_LLEXT_CUSTOM_HEAP_PLACEMENT`

Remove default placements of LLEXT heap sections in the linker script, allowing the user to place the heap(s) themselves.

Word granular access instruction memory heap Word granular access instruction memory is a type of instruction memory that is byte addressable, but can only be accessed with word-sized and aligned loads and stores. The LLEXT subsystem currently supports the placement of the instruction heap in word granular access instruction memory on the Xtensa architecture only. Support for non-Xtensa architectures will be added in the future on request.

To use LLEXT with the instruction heap in word granular access instruction memory, your Xtensa SoC or board must select the following option in addition to `CONFIG_HARVARD`.

`CONFIG_ARCH_HAS_WORD_GRANULAR_ACCESS_INSTR_MEM`

This option enables support for access to byte addressable and word granular access instruction memory.

If using the default backing data structure for the heap, `k_heap`, enable the following option.

`CONFIG_SYS_HEAP_BIG_ONLY`

Select this to optimize the code for big heaps only. This can accommodate any heap size but memory usage won't be as efficient with small sized heaps.

Unaligned and narrow accesses to instruction memory will be performed during the instruction heap initialization if this option is not selected.

Next, place the LLEXT instruction heap in that instruction memory following the instructions for heap placement above. Make sure your SoC or board implements `arch_memcpy_to_instr()` and `arch_memcpy_from_instr()` in addition to `arch_is_instr_mem()`. The word granular access library `memcpy_to_word_granular()` and `memcpy_from_word_granular()` functions may be helpful.

You may place your ELF buffer in RAM. At load time, the LLEXT subsystem will force the text region onto the instruction heap (even when the ELF buffer is writable) so that it is executable, and respect word granular access constraints when accessing the text region during loading and linking.

Warning

You may only use the buffer loader (`llext_buf_loader`) to load the ELF when the instruction heap is in word granular access instruction memory.

Note that the extension itself is responsible for ensuring any subsequent accesses to instruction memory also respect these constraints.

If you still encounter load / store exceptions, you can enable the unsigned load / store exception handler for Xtensa with the following option.

`CONFIG_XTENSA_EMULATE_UNSUPPORTED_UNSIGNED_LOAD_STORE`

When an unsigned load / store instruction triggers an unsupported load / store exception, the exception handler will perform the operation itself with supported word sized and aligned loads / stores. Does not currently support VLIW.

As mentioned in the option description, to use this option you must also disable VLIW, as enabling VLIW causes the compiler to generate signed load / store instructions.

`CONFIG_COMPILER_CODEGEN_VLIW_DISABLED`

Explicitly instructs the compiler to NEVER generate VLIW instructions.

ELF object type The LLEXT subsystem supports loading different types of extensions; the type can be set by choosing among the following Kconfig options:

`CONFIG_LLEXT_TYPE_ELF_OBJECT`

Build and expect relocatable files as binary object type for the LLEXT subsystem. A single compiler invocation is used to generate the object file.

`CONFIG_LLEXT_TYPE_ELF_RELOCATABLE`

Build and expect relocatable (partially linked) files as the binary object type for the LLEXT subsystem. These object files are generated by the linker by combining multiple object files into a single one.

`CONFIG_LLEXT_TYPE_ELF_SHAREDLIB`

Build and expect shared libraries as binary object type for the LLEXT subsystem. The standard linking process is used to generate the shared library from multiple object files.

> **Note**
>
> This is not currently supported on ARM architectures.

Minimize allocations The LLEXT subsystem loading mechanism, by default, uses a seek/read abstraction and copies all data into allocated memory; this is done to allow the extension to be loaded from any storage medium. Sometimes, however, data is already in a buffer in RAM and copying it is not necessary. The following option allows the LLEXT subsystem to optimize memory footprint in this case.

`CONFIG_LLEXT_STORAGE_WRITABLE`

Allow the extension to be loaded by directly referencing section data into the ELF buffer. To be effective, this requires the use of an ELF loader that supports the peek functionality, such as the `llext_buf_loader`.

> **Warning**
>
> The application must ensure that the buffer used to load the extension remains allocated until the extension is unloaded.

> **Note**
>
> This will directly modify the contents of the buffer during the link phase. Once the extension is unloaded, the buffer must be reloaded before it can be used again in a call to `llext_load()`.

Symbol Groups All LLEXT symbols belong to a group, with the inclusion of each group in the exported symbol table controlled by a corresponding Kconfig symbol. Exporting a symbol as part of a group is done with the `EXPORT_GROUP_SYMBOL` and `EXPORT_GROUP_SYMBOL_NAMED` macros. For example the following exports the symbol `memcpy` as part of the `LIBC` group:

```
EXPORT_GROUP_SYMBOL(LIBC, memcpy);
```

Group names are arbitrary, but they must be all uppercase. For each group used in C code, there **MUST** be a corresponding Kconfig symbol of the form:

```
config LLEXT_EXPORT_SYMBOL_GROUP_{GROUP_NAME}
   bool "Export all symbols from the {GROUP_NAME} group"
```

The default group for symbols (those declared with `EXPORT_SYMBOL` or `EXPORT_SYMBOL_NAMED`) is the `UNASSIGNED` group. As per the above rules, the inclusion of this group is controlled by `CONFIG_LLEXT_EXPORT_SYMBOL_GROUP_UNASSIGNED`.

The groups currently defined by Zephyr are:

Table 57: Zephyr LLEXT symbol groups

Group Name	Kconfig Symbol	Description
UNASSIGNED	CONFIG_LLEXT_EXPORT_SYMBOL_GROUP_UNAS	Symbols without an explicit group
SYSCALL	CONFIG_LLEXT_EXPORT_SYMBOL_GROUP_SYSC	Zephyr kernel system calls
LIBC	CONFIG_LLEXT_EXPORT_SYMBOL_GROUP_LIBC	C standard library functions (memcpy() etc)
DEVICE	CONFIG_LLEXT_EXPORT_SYMBOL_GROUP_DEVI	Devicetree devices

Using SLID for symbol lookups When an extension is loaded, the LLEXT subsystem must find the address of all the symbols residing in the main application that the extension references. To this end, the main binary contains a LLEXT-dedicated symbol table, filled with one symbol-name-to-address mapping entry for each symbol exported by the main application to extensions. This table can then be searched into by the LLEXT linker at extension load time. This process is pretty slow due to the nature of string comparisons, and the size consumed by the table can become significant as the number of exported symbols increases.

CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID

Perform an extra processing step on the Zephyr binary and on all extensions being built, converting every string in the symbol tables to a pointer-sized hash called Symbol Link Identifier (SLID), which is stored in the binary.

This speeds up the symbol lookup process by allowing usage of integer-based comparisons rather than string-based ones. Another benefit of SLID-based linking is that storing symbol names in the binary is no longer necessary, which provides a significant decrease in symbol table size.

> **Note**
>
> This option is not currently compatible with the *LLEXT EDK*.

> **Note**
>
> Using a different value for this option in the main binary and in extensions is not supported. For example, if the main application is built with CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID=y, it is forbidden to load an extension that was compiled with CONFIG_LLEXT_EXPORT_BUILTINS_BY_SLID=n.

EDK configuration Options influencing the generation and behavior of the LLEXT EDK are described in *LLEXT EDK Kconfig options*.

Building extensions

The LLEXT subsystem allows for the creation of extensions that can be loaded into a running Zephyr application. When building these extensions, it's very often useful to have access to the headers and compiler flags used by the main Zephyr application.

The easiest path to achieve this is to build the extension as part of the Zephyr application, using the *native Zephyr CMake features*. This will result in a single build providing both the main Zephyr application and the extension(s), which will all automatically be built with the same parameters.

In some cases, involving the full Zephyr build system may not be feasible or convenient; maybe the extension is built using a different compiler suite or as part of a different project altogether. In this case, the extension developer needs to export the headers and compiler flags used by the main Zephyr application. This can be done using the *LLEXT Extension Development Kit*.

Using the Zephyr CMake features The Zephyr build system provides a set of features that can be used to build extensions as part of the Zephyr application. This is the simplest way to build extensions, as it requires minimal additions to an application build system.

Building the extension An extension can be defined in the app's `CMakeLists.txt` by invoking the *`add_llext_target()`* function, providing the target name, the output and the source files. Usage is similar to the standard `add_custom_target` CMake function:

```
add_llext_target(
    <target_name>
    OUTPUT <ext_file.llext>
    SOURCES <src1> [<src2>...]
)
```

where:

- `<target_name>` is the name of the final CMake target that will result in the LLEXT binary being created;
- `<ext_file.llext>` is the name of the output file that will contain the packaged extension;
- `<src1> [<src2>...]` is the list of source files that will be compiled to create the extension.

The exact steps of the extension building process depend on the currently selected *ELF object format*.

The following custom properties of `<target_name>` are defined and can be retrieved using the `get_target_property()` CMake function:

`lib_target`

Target name for the source compilation and/or link step.

`lib_output`

The binary file resulting from compilation and/or linking steps.

`pkg_input`

The file to be used as input for the packaging step.

`pkg_output`

The final extension file name.

Tweaking the build process The following CMake functions can be used to modify the build system behavior during the extension build process to a fine degree. Each of the below functions takes the LLEXT target name as its first argument; it is otherwise functionally equivalent to the common Zephyr `target_*` version.

- *`llext_compile_definitions()`*
- *`llext_compile_features()`*
- *`llext_compile_options()`*
- *`llext_include_directories()`*
- *`llext_link_options()`*

Custom build steps The `add_llext_command` CMake function can be used to add custom build steps that will be executed during the extension build process. The command will be run at the specified build step and can refer to the properties of the target for build-specific details.

The function signature is:

```
add_llext_command(
    TARGET <target_name>
    [PRE_BUILD | POST_BUILD | POST_PKG]
    COMMAND <command> [args...]
)
```

The different build steps are:

`PRE_BUILD`

Before the extension code is linked, if the architecture uses dynamic libraries. This step can access `lib_target` and its own properties.

`POST_BUILD`

After the extension code is built, but before packaging it in an `.llext` file. This step is expected to create a `pkg_input` file by reading the contents of `lib_output`.

`POST_PKG`

After the extension output file has been created. The command can operate on the final llext file `pkg_output`.

Anything else after `COMMAND` will be passed to `add_custom_command()` as-is (including multiple commands and other options).

LLEXT Extension Development Kit (EDK) When building extensions as a standalone project, outside of the main Zephyr build system, it's important to have access to the same set of generated headers and compiler flags used by the main Zephyr application, since they have a direct impact on how Zephyr headers are interpreted and the extension is compiled in general.

This can be achieved by asking Zephyr to generate an Extension Development Kit (EDK) from the build artifacts of the main Zephyr application, by running the following command which uses the `llext-edk` target:

```
west build -t llext-edk
```

The generated EDK can be found in the build directory under the `zephyr` directory. It's a tarball that contains the headers and compile flags needed to build extensions. The extension developer can then include the headers and use the compile flags in their build system to build the extension.

EDK definition files The EDK includes several convenience files which define a set of variables that contain the compile flags needed by the project, as well as other build-related information, when enabled. The information is currently exported in the following formats:

- `Makefile.cflags`, for Makefile-based projects;
- `cmake.cflags`, for CMake-based projects.

Paths to the headers and flags are prefixed by the EDK root directory. This is automatically obtained for CMake projects from `CMAKE_CURRENT_LIST_DIR`; other formats refer to an `LLEXT_EDK_INSTALL_DIR` variable, which must be set by the user with the path where the EDK is installed before including the generated file.

> **Note**
>
> The `LLEXT_EDK` prefix in the variable name may be changed with the `CONFIG_LLEXT_EDK_NAME` option.

Compile flags The full list of flags needed to build an extension is provided by `LLEXT_CFLAGS`. Also provided is a more granular set of flags that can be used in support of different use cases, such as when building mocks for unit tests:

`LLEXT_INCLUDE_CFLAGS`

Compiler flags to add directories containing non-autogenerated headers to the compiler's include search paths.

`LLEXT_GENERATED_INCLUDE_CFLAGS`

Compiler flags to add directories containing autogenerated headers to the compiler's include search paths.

`LLEXT_ALL_INCLUDE_CFLAGS`

Compiler flags to add all directories containing headers used in the build to the compiler's include search paths. This is a combination of `LLEXT_INCLUDE_CFLAGS` and `LLEXT_GENERATED_INCLUDE_CFLAGS`.

`LLEXT_GENERATED_IMACROS_CFLAGS`

Compiler flags for autogenerated headers that must be included in the build via `-imacros`.

`LLEXT_BASE_CFLAGS`

Other compiler flags that control code generation for the target CPU. None of these flags are included in the above lists.

`LLEXT_CFLAGS`

All flags required to build an extension. This is a combination of `LLEXT_ALL_INCLUDE_CFLAGS`, `LLEXT_GENERATED_IMACROS_CFLAGS` and `LLEXT_BASE_CFLAGS`.

Target information The EDK includes information that identifies the target of the current Zephyr build. The following variables are currently defined and mirror the information available in the Zephyr build system:

`LLEXT_EDK_BOARD_NAME`
The board name used in the Zephyr build.

`LLEXT_EDK_BOARD_QUALIFIERS`
The board qualifiers, if provided, used in the Zephyr build.

`LLEXT_EDK_BOARD_REVISION`
The board revision, if provided, used in the Zephyr build.

`LLEXT_EDK_BOARD_TARGET`
The fully qualified board target used in the Zephyr build.

> **Note**
>
> The `LLEXT_EDK` prefix in the variable names may be changed with the `CONFIG_LLEXT_EDK_NAME` option.

LLEXT EDK Kconfig options The LLEXT EDK can be configured using the following Kconfig options:

`CONFIG_LLEXT_EDK_NAME`
: The name of the generated EDK tarball. This is also used as the prefix for several variables defined in the EDK files.

`CONFIG_LLEXT_EDK_USERSPACE_ONLY`
: If set, the EDK will include headers that do not contain code to route syscalls to the kernel. This is useful when building extensions that will run exclusively in user mode.

EDK Sample Refer to llext-edk for an example of how to use the LLEXT EDK.

Loading extensions

Once an extension is built and the ELF file is available, it can be loaded into the Zephyr application using the LLEXT API, which provides a way to load the extension into memory, access its symbols and call its functions.

Loading an extension An extension may be loaded using any implementation of a `llext_loader` which has a set of function pointers that provide the necessary functionality to read the ELF data. A loader also provides some minimal context (memory) needed by the `llext_load()` function. Several loaders are already provided:

- An implementation over a buffer containing an ELF in addressable memory in memory is available as `llext_buf_loader`. To use this kind of loader, it is helpful to use one of the `LLEXT_TEMPORARY_BUF_LOADER`, `LLEXT_PERSISTENT_BUF_LOADER`, or `LLEXT_WRITABLE_BUF_LOADER` macros to tell LLEXT the appropriate type of memory buffer.
- An implementation that reads data from a file in the filesystem is available as the `llext_fs_loader`. The path to the file must be provided when creating the loader with the `LLEXT_FS_LOADER` macro.
- An implementation that uses semihosting to read from a file in the host filesystem is available as the `llext_semihost_loader`. The path to the file must be provided when creating the loader with the `LLEXT_SEMIHOST_LOADER` macro.

The extensions are loaded with a call to the `llext_load()` function, passing in the extension name and the configured loader. Once that completes successfully, the extension is loaded into memory and is ready to be used.

> **Note**
>
> When *User Mode* is enabled, the extension will not be included in any user memory domain. To allow access from user mode, the `llext_add_domain()` function must be called.

Initializing and cleaning up the extension The extension may define a number of initialization functions that must be called after loading but before any function in it can be used; this is typical in languages such as C++ that provide the concept of object constructors. The same is true for cleanup functions that must be called before unloading the extension.

LLEXT supports calling the functions listed in the `.preinit_array` and `.init_array` sections of the ELF file with the `llext_bringup()` function, and the functions listed in the `.fini_array` section with the `llext_teardown()` function. These APIs are compatible with *User Mode*, and thus can be called from either kernel or user context.

Important

The code run by these functions is fully determined by the contents of the ELF file. This may have security implications if its origin is untrusted.

If the extension requires a dedicated thread, the `llext_bootstrap()` function can be used to minimize boilerplate code. This function has a signature that is compatible with the `k_thread_create()` API, and will call `llext_bringup()`, then a user-specified function in the same context, and finally `llext_teardown()` before returning.

Accessing code and data To interact with the newly loaded extension, the host application must use the `llext_find_sym()` function to get the address of the exported symbol. The returned `void *` can then be cast to the appropriate type and used.

A wrapper for calling a function with no arguments is provided in `llext_call_fn()`.

Advanced users that need direct access to areas of the newly loaded extension may want to refer to `llext_get_section_info()` and other LLEXT inspection APIs.

Cleaning up after use The `llext_unload()` function must be called to free the memory used by the extension once it is no longer required. After this call completes, all pointers to symbols in the extension that were obtained will be invalid.

Troubleshooting

This feature is being actively developed and as such it is possible that some issues may arise. Since linking does modify the binary code, in case of errors the results are difficult to predict. Some common issues may be:

- Results from `llext_find_sym()` point to an invalid address;
- Constants and variables defined in the extension do not have the expected values;
- Calling a function defined in an extension results in a hard fault, or memory in the main application is corrupted after returning from it.

If any of this happens, the following tips may help understand the issue:

- Make sure `CONFIG_LLEXT_LOG_LEVEL` is set to `DEBUG`, then obtain a log of the `llext_load()` invocation.
- If possible, disable memory protection (MMU/MPU) and see if this results in different behavior.
- Try to simplify the extension to the minimum possible code that reproduces the issue.
- Use a debugger to inspect the memory and registers to try to understand what is happening. See *Debugging extensions* for more details.

If the issue persists, please open an issue in the GitHub repository, including all the above information.

Debugging extensions

Debugging extensions is a complex task. Since the extension code is by definition not built with the Zephyr application, the final Zephyr ELF file does not contain the symbols for extension code. Furthermore, the extension is dynamically relocated by `llext_load()` at runtime, so even if the symbols were available, it would be impossible for the debugger to know the final locations of the symbols in the extension code.

Setting up the debugger session properly in this case requires a few manual steps. The following sections will provide some tips on how to do it with the Zephyr SDK and the debug features provided by `west`, but the instructions can be adapted to any GDB-based debugging environment.

Extension debugging process

1. Make sure the project is set up to display the verbose LLEXT debug output (`CONFIG_LOG` and `CONFIG_LLEXT_LOG_LEVEL_DBG` are set).
2. Build the Zephyr application and the extensions.

   For each target `name` included in the current build, two files will be generated into the `llext` subdirectory of the build root:

   `name_ext_debug.elf`

   An intermediate ELF file with full debugging information.

   `name.llext`

   The final extension binary, stripped to the essential data required for loading into the Zephyr application.

   Other files may be present, depending on the target architecture and the build configuration.
3. Start a debugging session of the main Zephyr application. This is described in the *Debugging* section of the documentation; on supported boards it is as easy as running `west debug`, perhaps with some additional arguments.
4. Set a breakpoint just after the `llext_load()` function in your code and let it run. This will load the extension into memory and relocate it. The output logs will contain a line with gdb `add-symbol-file flags:`, followed by lines all starting with `-s`.
5. Type this command in the GDB console to load this extension's symbols:

   ```
   add-symbol-file <path-to-debug.elf> <load-addresses>
   ```

   where `<path-to-debug.elf>` is the full path of the ELF file with debug information identified in step 2, and `<load-addresses>` is a space separated list of all the `-s` lines collected from the log in the previous step.
6. The extension symbols are now available to the debugger. You can set breakpoints, inspect variables, and step through the code as usual.

Steps 4-6 can be repeated for every extension that is loaded by the application, if there are several.

Warning

It is almost certain that the loaded symbols will be shadowed by others in the main application; for example, they may be located inside the memory area of the ELF buffer or the LLEXT heap.

In this case GDB chooses the first known symbol and therefore associates the addresses to some `elf_buffer+0x123` instead of an expected `ext_fn`. This further confuses its high-level operations like source stepping or inspecting locals, since they are meaningless in that context.

Symbol lookup issues Two possible solutions to this problem are discussed in the following paragraphs.

Discard all Zephyr symbols The simplest option is to drop all the Zephyr application symbols from GDB by invoking `add-symbol-file` with no arguments, before step 5. This will however focus the debugging session to the llext only, as all information about the Zephyr application will be lost. For example, the debugger may not be able to properly follow stack traces outside the extension code.

It is possible to use the same technique multiple times in the same session to switch between the main and extension symbol tables as required, but it rapidly becomes cumbersome.

Edit the ELF file This alternative is more complex but allows for a more seamless debugging experience. The idea is to edit the main Zephyr ELF file to remove information about the symbols that overlap with the extension that is to be debugged, so that when the extension symbols are loaded, GDB will not have any ambiguity. This can be done by using `objcopy` with the `-N <symbol>` option.

Identifying the offending symbols is however an iterative trial-and-error procedure, as there can be many different layers; for example, the ELF buffer may be itself contained in a symbol for the data segment. Fortunately, this knowledge can then be used several times as the list is unlikely to change for a given project.

Example debugging session This example demonstrates how to debug the `detached_fn` extension in the `tests/subsys/llext` project (specifically, the `writable` case), on an emulated `mps2/an385` board which is based on an ARM Cortex-M3.

> **Note**
>
> The logs below have been obtained using Zephyr version 4.1 and the Zephyr SDK version 0.17.0. However, the exact addresses may still vary between runs even when using the same versions. Adjust the commands below to match the results of your own session.

The following command will build the project and start the emulator in debugging mode:

Listing 13: Terminal 1 (build, QEMU emulator, GDB server)

```
zephyr$ west build -p -b mps2/an385 tests/subsys/llext/ -T llext.writable -t␣
↪debugserver_qemu
-- west build: generating a build system
[...]
-- west build: running target debugserver_qemu
[...]
[186/187] To exit from QEMU enter: 'CTRL+a, x'[QEMU] CPU: cortex-m3
```

On a separate terminal, set `ZEPHYR_SDK_INSTALL_DIR` to the directory for the Zephyr SDK on your installation, then start the GDB client for the target:

Listing 14: Terminal 2 (GDB client)

```
zephyr$ export LLEXT_SDK_INSTALL_DIR=/opt/zephyr-sdk-0.17.0
zephyr$ ${LLEXT_SDK_INSTALL_DIR}/arm-zephyr-eabi/bin/arm-zephyr-eabi-gdb␣
↪build/zephyr/zephyr.elf
GNU gdb (Zephyr SDK 0.17.0) 12.1
[...]
Reading symbols from build/zephyr/zephyr.elf...
(gdb)
```

Connect, set a breakpoint on the `llext_load` function and run until it finishes:

Listing 15: Terminal 2 (GDB client)

```
(gdb) target extended-remote :1234
Remote debugging using :1234
z_arm_reset () at zephyr/arch/arm/core/cortex_m/reset.S:124
124         movs.n r0, #_EXC_IRQ_DEFAULT_PRIO
(gdb) break llext_load
Breakpoint 1 at 0x236c: file zephyr/subsys/llext/llext.c, line 168.
(gdb) continue
Continuing.

Breakpoint 1, llext_load (ldr=ldr@entry=0x2000bef0 <ztest_thread_stack+3488>,
                          name=name@entry=0x9d98 "test_detached",
                          ext=ext@entry=0x2000abb8 <detached_llext>,
                          ldr_parm=ldr_parm@entry=0x2000bee8
↪<ztest_thread_stack+3480>)
              at zephyr/subsys/llext/llext.c:168
168             *ext = llext_by_name(name);
(gdb) finish
Run till exit from #0  llext_load ([...])
    at zephyr/subsys/llext/llext.c:168
llext_test_detached () at zephyr/tests/subsys/llext/src/test_llext.c:481
481             zassert_ok(res, "load should succeed");
```

The first terminal will have printed lots of debugging information related to the extension loading. Find the section with the addresses:

Listing 16: Terminal 1 (build, QEMU emulator, GDB server)

```
[...]
D: Allocate and copy regions...
[...]
D: gdb add-symbol-file flags:
D: -s .text 0x20000034
D: -s .data 0x200000b4
D: -s .bss 0x2000c2e0
D: -s .rodata 0x200000b8
D: -s .detach 0x200001d0
D: Counting exported symbols...
[...]
```

Use these addresses to load the symbols into GDB:

Listing 17: Terminal 2 (GDB client)

```
(gdb) add-symbol-file build/llext/detached_fn_ext_debug.elf -s .text 0x20000034 -s␣
↪.data 0x200000b4 -s .bss 0x2000c2e0 -s .rodata 0x200000b8 -s .detach 0x200001d0
add symbol table from file "build/llext/detached_fn_ext_debug.elf" at
        .text_addr = 0x20000034
        .data_addr = 0x200000b4
        .bss_addr = 0x2000c2e0
        .rodata_addr = 0x200000b8
        .detach_addr = 0x200001d0
(y or n) y
Reading symbols from build/llext/detached_fn_ext_debug.elf...
(gdb) break detached_entry
Breakpoint 2 at 0x200001d0 (2 locations)
(gdb) continue
```

(continues on next page)

(continued from previous page)

```
Continuing.

Breakpoint 2, 0x200001d0 in test_detached_ext ()
(gdb) backtrace
#0  0x200001d0 in test_detached_ext ()
#1  0x200000ac in test_detached_ext ()
#2  0x00000706 in llext_test_detached () at␣
↪zephyr/tests/subsys/llext/src/test_llext.c:496
#3  0x00001a36 in run_test_functions (suite=0x92bc <z_ztest_test_node_llext>,␣
↪data=0x0 <cbvprintf_package>, test=0x92d8
↪<z_ztest_unit_test.llext.test_detached>) at␣
↪zephyr/subsys/testsuite/ztest/src/ztest.c:328
#4  test_cb (a=0x92bc <z_ztest_test_node_llext>, b=0x92d8
↪<z_ztest_unit_test.llext.test_detached>, c=0x0 <cbvprintf_package>) at␣
↪zephyr/subsys/testsuite/ztest/src/ztest.c:662
#5  0x00000e96 in z_thread_entry (entry=0x1a05 <test_cb>, p1=0x92bc
↪<z_ztest_test_node_llext>, p2=0x92d8 <z_ztest_unit_test.llext.test_detached>,␣
↪p3=0x0 <cbvprintf_package>) at zephyr/lib/os/thread_entry.c:48
#6  0x00000000 in ?? ()
```

The symbol associated with the breakpoint location and the last stack frames mistakenly reference the ELF buffer in the Zephyr application instead of the extension symbols. Note that GDB however knows both:

Listing 18: Terminal 2 (GDB client)

```
(gdb) info sym 0x200001d0
test_detached_ext + 464 in section datas of zephyr/build/zephyr/zephyr.elf
detached_entry in section .detach of zephyr/build/llext/detached_fn_ext_debug.elf
(gdb) info sym 0x200000ac
test_detached_ext + 172 in section datas of zephyr/build/zephyr/zephyr.elf
test_entry + 8 in section .text of zephyr/build/llext/detached_fn_ext_debug.elf
```

It is also impossible to inspect the variables in the extension or step through code properly:

Listing 19: Terminal 2 (GDB client)

```
(gdb) print bss_cnt
No symbol "bss_cnt" in current context.
(gdb) print data_cnt
No symbol "data_cnt" in current context.
(gdb) next
Single stepping until exit from function test_detached_ext,
which has no line number information.

Breakpoint 2, 0x200001ea in test_detached_ext ()
(gdb)
```

Discarding symbols Discarding the Zephyr symbols and only focusing on the extension restores full debugging functionality at the cost of losing the global context (note the backtrace stops outside the extension):

Listing 20: Terminal 2 (GDB client)

```
(gdb) symbol-file
Discard symbol table from `zephyr/build/zephyr/zephyr.elf'? (y or n) y
```

(continues on next page)

(continued from previous page)

```
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file"␣
↪command.
No symbol file now.
(gdb) add-symbol-file build/llext/detached_fn_ext_debug.elf -s .text 0x20000034 -s␣
↪.data 0x200000b4 -s .bss 0x2000c2e0 -s .rodata 0x200000b8 -s .detach 0x200001d0
add symbol table from file "build/llext/detached_fn_ext_debug.elf" at
        .text_addr = 0x20000034
        .data_addr = 0x200000b4
        .bss_addr = 0x2000c2e0
        .rodata_addr = 0x200000b8
        .detach_addr = 0x200001d0
(y or n) y
Reading symbols from build/llext/detached_fn_ext_debug.elf...
(gdb) backtrace
#0  detached_entry () at zephyr/tests/subsys/llext/src/detached_fn_ext.c:18
#1  0x200000ac in test_entry () at␣
↪zephyr/tests/subsys/llext/src/detached_fn_ext.c:26
#2  0x00000706 in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb) next
19              zassert_true(data_cnt < 0);
(gdb) print bss_cnt
$1 = 1
(gdb) print data_cnt
$2 = -2
(gdb)
```

Editing the ELF file In this alternative approach, the patches to the Zephyr ELF file must be performed after building the Zephyr binary and starting the emulator on Terminal 1, but before starting the GDB client on Terminal 2.

The above debugging session already identified `test_detached_ext`, the char array that holds the ELF file, as an offending symbol, so that will be removed in a first pass. Performing the same steps multiple times, `__data_start` and `__data_region_start` can also be found to overlap the memory area of interest.

The following commands will remove all of these from the Zephyr ELF file, then start a debugging session on the modified file:

Listing 21: Terminal 2 (GDB client)

```
zephyr$ export LLEXT_SDK_INSTALL_DIR=/opt/zephyr-sdk-0.17.0
zephyr$ ${LLEXT_SDK_INSTALL_DIR}/arm-zephyr-eabi/bin/arm-zephyr-eabi-objcopy -N␣
↪test_detached_ext -N __data_start -N __data_region_start build/zephyr/zephyr.elf␣
↪build/zephyr/zephyr-edit.elf
zephyr$ ${LLEXT_SDK_INSTALL_DIR}/arm-zephyr-eabi/bin/arm-zephyr-eabi-gdb␣
↪build/zephyr/zephyr-edit.elf
GNU gdb (Zephyr SDK 0.17.0) 12.1
[...]
Reading symbols from build/zephyr/zephyr-edit.elf...
(gdb)
```

The same steps used in the previous run can be performed again to attach to the GDB server and load both the extension and its debug symbols. This time, however, the result is rather different:

- the `break` command includes line number information;
- the output from `backtrace` contains functions from both the extension and the Zephyr ap-

plication;

- the local variables can be properly inspected.

Listing 22: Terminal 2 (GDB client)

```
(gdb) add-symbol-file build/llext/detached_fn_ext_debug.elf [...]
[...]
Reading symbols from build/llext/detached_fn_ext_debug.elf...
(gdb) break detached_entry
Breakpoint 2 at 0x200001d6: file zephyr/tests/subsys/llext/src/detached_fn_ext.c,␣
↪line 17.
(gdb) continue
Continuing.

Breakpoint 2, detached_entry () at␣
↪zephyr/tests/subsys/llext/src/detached_fn_ext.c:17
17              printk("bss %u @ %p\n", bss_cnt++, &bss_cnt);
(gdb) backtrace
#0  detached_entry () at zephyr/tests/subsys/llext/src/detached_fn_ext.c:17
#1  0x200000ac in test_entry () at␣
↪zephyr/tests/subsys/llext/src/detached_fn_ext.c:26
#2  0x00000706 in llext_test_detached () at␣
↪zephyr/tests/subsys/llext/src/test_llext.c:496
#3  0x00001a36 in run_test_functions (suite=0x92bc <z_ztest_test_node_llext>,␣
↪data=0x0 <cbvprintf_package>, test=0x92d8
↪<z_ztest_unit_test.llext.test_detached>) at␣
↪zephyr/subsys/testsuite/ztest/src/ztest.c:328
#4  test_cb (a=0x92bc <z_ztest_test_node_llext>, b=0x92d8
↪<z_ztest_unit_test.llext.test_detached>, c=0x0 <cbvprintf_package>) at␣
↪zephyr/subsys/testsuite/ztest/src/ztest.c:662
#5  0x00000e96 in z_thread_entry (entry=0x1a05 <test_cb>, p1=0x92bc
↪<z_ztest_test_node_llext>, p2=0x92d8 <z_ztest_unit_test.llext.test_detached>,␣
↪p3=0x0 <cbvprintf_package>) at zephyr/lib/os/thread_entry.c:48
#6  0x00000000 in ?? ()
(gdb) print bss_cnt
$1 = 0
(gdb) print data_cnt
$2 = -3
(gdb)
```

API Reference

Linkable loadable extensions

Related code samples

- Linkable loadable extensions "module" sampleCall a function in a loadable extension module, either built-in or loaded at runtime.
- Linkable loadable extensions EDKEnable linkable loadable extension development outside the Zephyr tree using LLEXT EDK (Extension Development Kit).
- Linkable loadable extensions shell moduleManage loadable extensions using shell commands.

Exported symbol definitions

ELF loader context

ELF inspection APIs

> **Note**
>
> The LLEXT subsystem requires architecture-specific support. It is currently available only on RISC-V, ARM, ARM64, ARC, x86, and Xtensa cores.

4.11.2 Multimedia Pipeline (mpipe)

- *Overview*
- *Building a pipeline*
- *Elements, pads and links*
- *The state machine*
- *Capability negotiation*
- *Buffer pool negotiation*
- *Buffer flow*
- *Pipeline runtime*
- *Writing an element*
- *Configuration Options*
- *API Reference*

Overview

The Multimedia Pipeline subsystem (mpipe) builds a media stream out of self-contained processing components called **elements**. An application declares the elements it needs, links them into a graph, and drives that graph through a state machine; mpipe negotiates the data format between neighboring elements, settles which buffer pool provides the buffers, and moves those buffers from one element to the next.

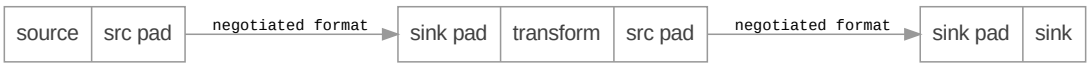


Fig. 45: A pipeline is a graph of elements joined at their pads.

mpipe provides the pieces and the rules by which they fit together rather than finished solutions, so a pipeline is assembled much like building with LEGO bricks: the same graph runs on a different board by binding its elements to different devices, and a new requirement is usually one more element rather than a rewrite.

The elements themselves live in **plugins**, grouped by media domain. A plugin brings its own directory, its own Kconfig and its own headers, and the build picks it up without any edit to the framework, so a silicon vendor or a middleware provider can ship elements without altering the core.

mpipe performs no dynamic allocation: buffers come from pools sized while the pipeline starts, and everything on the negotiation path is fixed-size and held by value. The cost lands on the

stack instead. Negotiation runs inside `mpipe_element_set_state()`, on whichever thread calls it, and holds several capabilities live at once - budget for it on that thread, which for the example below is the one running `main`.

mpipe is optional, and it is not always the right tool. An application driving a single device is better served by that device's API directly. mpipe earns its place once several devices have to agree on a format and hand buffers to each other.

Building a pipeline

An application includes `<zephyr/mpipe/mpipe.h>`, and only that, for the whole core API; each element it instantiates adds that element's own header.

Elements are plain objects the application owns; mpipe allocates none of them. Each element type has its own init function taking that type and an id:

```
/* Concrete element types come from plugins; each has its own header. */
static struct mpipe pipe;
static struct my_src source;
static struct my_sink sink;

ret = mpipe_pipeline_init(&pipe, PIPE_ID);
ret = my_src_init(&source, SRC_ID);
ret = my_sink_init(&sink, SINK_ID);
```

Ids only have to be unique within the pipeline. Elements are configured through properties, which is how an element is set up without the caller knowing its concrete type:

```
ret = mpipe_object_set_properties((struct mpipe_object *)&source,
                                  MY_SRC_PROP_PATH, "/SD:/in.bin",
                                  MPIPE_PROP_LIST_END);
```

They are then added to the pipeline, in any order, and linked in stream order:

```
ret = mpipe_bin_add((struct mpipe_bin *)&pipe,
                    (struct mpipe_element *)&source,
                    (struct mpipe_element *)&sink, NULL);

ret = mpipe_element_link((struct mpipe_element *)&source,
                         (struct mpipe_element *)&sink, NULL);
```

The casts are well defined because every element embeds its base as its first member. Linking refuses a pair of elements whose capabilities cannot possibly intersect, so an impossible graph fails while it is being built rather than when it is started.

Setting the pipeline to `MPIPE_STATE_PLAYING` is not a jump to that state. The pipeline and its children step through the states in order, `READY` to `PAUSED` to `PLAYING`, which is why the format is negotiated and the pools are started along the way.

The application then observes the pipeline's message channel, both to learn how the run ends and to act on failures. An error message names the element that raised it and the phase it was in, so an application can report that capability negotiation failed at element 3 rather than only that the run stopped. Returning the graph to `MPIPE_STATE_READY` tears it down:

```
/* Needs CONFIG_ZBUS_MSG_SUBSCRIBER=y */
ZBUS_MSG_SUBSCRIBER_DEFINE(main_sub);

struct zbus_channel *bus = mpipe_element_get_bus_chan((struct mpipe_element *)&
↪pipe);
```

(continues on next page)

(continued from previous page)

```
ret = zbus_chan_add_obs(bus, &main_sub, K_FOREVER);

if (mpipe_element_set_state((struct mpipe_element *)&pipe, MPIPE_STATE_PLAYING) !=␣
↪0) {
        /* the element that refused is still in its previous state */
}

do {
        ret = zbus_sub_wait_msg(&main_sub, &chan, &msg, K_FOREVER);
} while ((msg.type & (MPIPE_MESSAGE_ERROR | MPIPE_MESSAGE_EOS)) == 0);

(void)mpipe_element_set_state((struct mpipe_element *)&pipe, MPIPE_STATE_READY);
```

Elements, pads and links

Every mpipe *element* derives from `mpipe_object` by embedding it as its first member. The object layer carries what the framework needs of anything it holds: an id, the container that holds it, list linkage, and the property callbacks. On top of it, `mpipe_element` adds the state machine and the pads, and the element bases specialize it. The types that only carry data - a capability, a message, a dispatch - are plain structs outside that hierarchy:

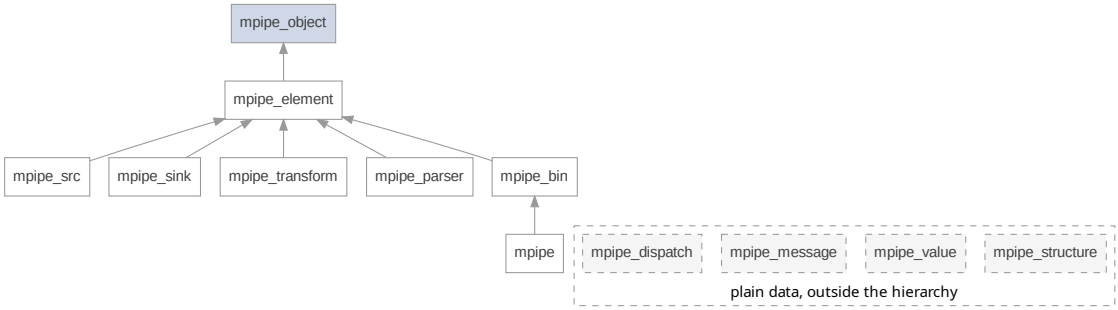


Fig. 46: Inheritance is struct embedding, so upcasting is a plain C cast.

- `mpipe_src` produces buffers and has source pads only. It also drives the negotiation, because it sits at the head of the graph.
- `mpipe_sink` consumes them and has sink pads only.
- `mpipe_transform` has one of each and turns input into output.
- `mpipe_parser` cuts a formless byte stream into whole frames, which is a different job from transforming one format into another.
- `mpipe_bin` holds other elements and forwards state changes to them in the order the transition requires.
- `mpipe` is the top-level bin: it owns the streaming thread and the message channel.

A `mpipe_pad` is where two elements meet. It carries a direction (source or sink), the peer it is linked to, the capability negotiated on it, and the callbacks the framework dispatches to: `chain_fn` receives a buffer, `query_fn` answers a query, and `event_fn` handles an event. Linking two elements links a source pad to a sink pad, and every hop of the stream is one such pair.

The state machine

An element is in one of three states and moves between neighbors one step at a time:

Fig. 47: What each transition does.

READY means constructed and linked, holding no format and no buffers. The READY to PAUSED transition is where the work happens: the source drives capability negotiation across the whole graph, the buffer pool query settles who provides buffers and how many, and the pools start.

Which way a transition goes determines the order a bin works through its children:

- Going **up**, children transition from the sink towards the source, so a downstream element is ready before anything is pushed into it.
- Going **down**, children transition from the source towards the sink, so nothing keeps producing into an element that has already been torn down.

Nothing unwinds a failed transition. An element that refuses one stays where it was while the elements that already moved keep their new state, which is deliberate: the resulting graph shows exactly which element refused and in which transition.

Capability negotiation

A **capability** describes the data crossing a link: a media type plus a set of fields, each an `mpipe_caps_field` identifier paired with an `mpipe_value`. It is a `mpipe_structure`, a fixed-size type held by value:

```
struct mpipe_structure s;

mpipe_structure_init_fields(&s, MPIPE_MEDIA_VIDEO,
    MPIPE_CAPS_PIXEL_FORMAT, MPIPE_TYPE_UINT, VIDEO_PIX_FMT_RGB565,
    MPIPE_CAPS_IMAGE_WIDTH, MPIPE_TYPE_UINT_RANGE, 16, 1280, 2,
    MPIPE_CAPS_IMAGE_HEIGHT, MPIPE_TYPE_UINT_RANGE, 16, 720, 2,
    MPIPE_CAPS_END);
```

A field holds either a single value or a range written `[min, max, step]`, so the capability above covers every width from 16 to 1280 in steps of 2. Two capabilities intersect when they share a media type, have at least one field identifier in common, and every shared field has intersecting values. The result is the union of both: a shared field holds the intersected value, and a field only one side carries passes through unchanged, which is what lets a constraint travel down a chain of elements that do not themselves care about it. An *ANY* capability constrains nothing and intersects with anything - it is what a pad carries before it has negotiated - while an *empty* one carries no field and intersects with nothing.

The source drives the negotiation on READY to PAUSED, in two passes:

Fig. 48: Capability negotiation, driven by the source on READY to PAUSED

A caps query travels downstream, each element narrowing it against what it supports before asking its own downstream, and the answer comes back narrowed to what the whole chain has in common. If an element refuses, the source simply offers the next format it supports rather than failing the negotiation.

A transform is the interesting case, because its two sides may speak different formats: a decoder takes one format in and produces another out. They need not - an element working in place rewrites the buffer where it lies, so the same format crosses it - but where they do differ, the `transform_caps` hook is what maps a capability from one side of the element to what the other side could then be. The query crosses the element through it in both directions: out to ask the downstream peer, and back to express the answer in terms of the input side again.

The answer may still hold ranges, so the source **fixates** it - each range reduced to a single value - and announces the result downstream as a caps event. Every element applies its side through its `set_caps` hook, which is where hardware is actually configured.

Because a pad holds one capability rather than a set, an element that supports several formats is walked by index: the framework asks it for capability 0, then 1, and so on until it reports there are no more. An element whose formats come from a device answers each index by asking its driver, so nothing has to be materialized in advance.

Buffer pool negotiation

Formats alone do not say how many buffers a graph needs, how large they must be, how they must be aligned, or whose pool provides them. A second query settles that, immediately after the format is fixed and in the same transition:

Fig. 49: Buffer pool negotiation, immediately after the format is fixed

The query travels down to the sink and proposals are written on the way back up, so an element always has its downstream's proposal in hand before it decides anything.

A transform's two sides are settled independently: `decide_buffer_pool` uses what its downstream proposed to settle the **output** pool, while `propose_buffer_pool` answers the upstream query with what the element needs on its **input**. The exception is passthrough, where one buffer crosses the element and there is only one pool to settle.

A proposal is either an entire pool or a bare config. Offering the pool lets the upstream element adopt it and avoid a copy; offering a config states requirements without handing anything over. Either way, a demand reaches a pool its proposer still owns only through `mpipe_buffer_pool_set_config()`, and the pool's owner validates, clamps or refuses it.

Whoever starts a pool is who stops it, on `PAUSED` to `READY`. That symmetry is what makes a stop and replay behave like a first run.

Buffer flow

Buffer flow is zero-copy. Each element that produces data owns a `mpipe_buffer_pool`, and a pool is a vtable - `configure`, `set_config`, `start`, `stop`, `acquire_buffer`, `release_buffer` - over whatever backs it. That indirection is what lets a plugin hand out the buffer its driver already owns rather than a copy of it, so a frame captured by a camera reaches the display without ever being moved. What travels between elements is a reference, not the pixels.

Buffers are Zephyr `net_buf` allocations, with an `mpipe_buffer_meta` alongside carrying what the framework needs to know about one: the pool that owns it, how much of it is valid, a timestamp.

`mpipe_push_buffer()` walks a buffer downstream: for each hop it takes the source pad's peer, checks that pad's flushing gate, calls its `chain_fn`, and follows the buffer the element produced to the next element. A NULL output means the buffer was consumed and the walk stops. The chain function owns the buffer it is given and releases it even when it fails, so ownership never depends on the error path taken.

Pipeline runtime

The pipeline's own thread drives the source: it acquires a buffer and pushes it downstream until the source reports the end of its data, at which point it sends an end-of-stream event downstream

and pauses itself. An element that needs to decouple two halves of a graph onto separate threads does so by placing a queuing element between them.

Tearing a running graph down is ordered carefully, and the order is what keeps it from losing data or deadlocking:

- `PLAYING` to `PAUSED` only pauses the source thread. Whatever is queued is preserved, so resuming continues without loss. This is a pause, not a teardown.
- `PAUSED` to `READY` raises a flushing gate on every pad *before* the children dismantle their pools, so a buffer still in flight is dropped rather than pushed into an element that has already been torn down. The streaming thread is joined only *after* the children have drained, because a child still holding that thread in a full queue would otherwise deadlock the join.

Messages travel to the application on the pipeline's message channel, a zbus channel reachable with `mpipe_element_get_bus_chan()`. A message carries where something happened - the element that emitted it - and what happened: a type, plus, on a failure, a domain naming the phase that failed and an errno saying why. It carries no human-facing string; the sentence describing a failure belongs in the log at the site that detected it, while an application branches on the domain and the errno.

A pipeline with several sinks produces one end-of-stream message per sink. The pipeline counts them and passes on only the last, so the application is notified exactly once and never tears the graph down while a branch is still running.

Writing an element

An element is written outside the framework and requires no change to it. It embeds one of the bases as its first member, calls that base's init with its own id, and overrides only what it cares about:

```
struct my_sink {
        struct mpipe_sink sink;   /* must be first */
        const struct device *dev;
};

int my_sink_init(struct my_sink *self, uint8_t id)
{
        int ret = mpipe_sink_init(&self->sink, id);

        if (ret != 0) {
                return ret;
        }

        self->sink.sink_pad.enum_caps_fn = my_sink_enum_caps;
        self->sink.set_caps = my_sink_set_caps;

        return 0;
}
```

An element that overrides `change_state` must chain to its base, which is what performs the capability reset and the pool teardown every element is expected to do.

The other half is saying what the element supports. A capability known at build time is a `static const struct mpipe_structure` the element copies out of, and `MPIPE_STRUCTURE_DEFINE` places one in `.rodata`; several are an array of them. An element backed by a device instead builds each capability inside its `enum_caps_fn` by asking the driver, and one configured by the application takes it from a property.

Configuration Options

- CONFIG_MPIPE enables the framework. Each plugin has its own option and contributes its own Kconfig file.
- CONFIG_MPIPE_STRUCTURE_MAX_FIELDS sizes one capability. Size it for the largest union of fields that can meet in one intersection, not for the number of fields an element sets, since an intersection carries through the fields only one side constrains.
- CONFIG_MPIPE_NET_BUF_POOL_COUNT sizes the shared net_buf pool: the sum of the in-flight buffers of every pipeline.
- CONFIG_MPIPE_BIN_MAX_CHILDREN bounds the elements in one bin, which sizes the arrays used to order them during a state change.
- CONFIG_MPIPE_THREADS_NUM, CONFIG_MPIPE_THREAD_STACK_SIZE and CONFIG_MPIPE_THREAD_DEFAULT_PRIORITY configure the pooled threads a pipeline draws from.
- CONFIG_MPIPE_WORKQUEUE lets elements offload finite work items onto a shared priority work queue.
- CONFIG_MPIPE_RPC builds the client-side elements that run their processing on another core.
- CONFIG_MPIPE_FAKE_SRC provides a source of synthetic data, used to exercise a graph where a real source would need hardware.

API Reference

Framework

4.11.3 Sensing Subsystem

- *Overview*
- *Configurability*
- *Main Features*
- *Major Flows*
- *Sensor Types And Instance*
- *Sensor Instance Handler*
- *Sensor Sample Value*
- *Device Tree Configuration*
- *API Reference*

Overview

Sensing Subsystem is a high level sensor framework inside the OS user space service layer. It is a framework focused on sensor fusion, client arbitration, sampling, timing, scheduling and sensor based power management.

Key concepts in Sensing Subsystem include physical sensor and virtual sensor objects, and a scheduling framework over sensor object relationships. Physical sensors do not depend on any other sensor objects for input, and will directly interact with existing zephyr sensor device drivers. Virtual sensors rely on other sensor objects (physical or virtual) as report inputs.

The sensing subsystem relies on Zephyr sensor device APIs (existing version or update in future) to leverage Zephyr's large library of sensor device drivers (100+).

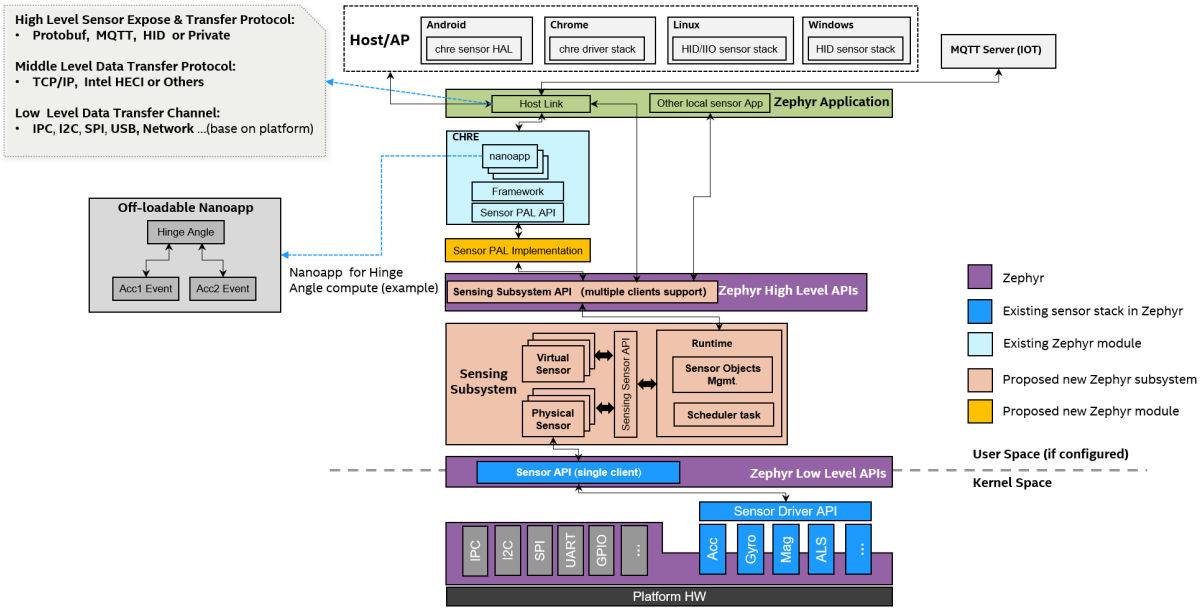
Use of the sensing subsystem is optional. Applications that only need to access simple sensors devices can use the Zephyr *Sensors* API directly.

Since the sensing subsystem is separated from device driver layer or kernel space and could support various customizations and sensor algorithms in user space with virtual sensor concepts. The existing sensor device driver can focus on low layer device side works, can keep simple as much as possible, just provide device HW abstraction and operations etc. This is very good for system stability.

The sensing subsystem is decoupled with any sensor expose/transfer protocols, the target is to support various up-layer frameworks and Applications with different sensor expose/transfer protocols, such as CHRE, HID sensors Applications, MQTT sensor Applications according different products requirements. Or even support multiple Applications with different up-layer sensor protocols at the same time with it's multiple clients support design.

Sensing subsystem can help build a unified Zephyr sensing architecture for cross host OSes support and as well as IoT sensor solutions.

The diagram below illustrates how the Sensing Subsystem integrates with up-layer frameworks.



Configurability

- Reusable and configurable standalone subsystem.
- Based on Zephyr existing low-level Sensor API (reuse 100+ existing sensor device drivers)
- Provide Zephyr high-level Sensing Subsystem API for Applications.
- Separate option CHRE Sensor PAL Implementation module to support CHRE.
- Decoupled with any host link protocols, it's Zephyr Application's role to handle different protocols (MQTT, HID or Private, all configurable)

Main Features

Scope

Focus on framework for sensor fusion, multiple clients, arbitration, data sampling, timing management and scheduling.

Sensor Abstraction

- **Physical sensor**: interacts with Zephyr sensor device drivers, focus on data collecting.
- **Virtual sensor**: relies on other sensor(s), **physical** or **virtual**, focus on data fusion.

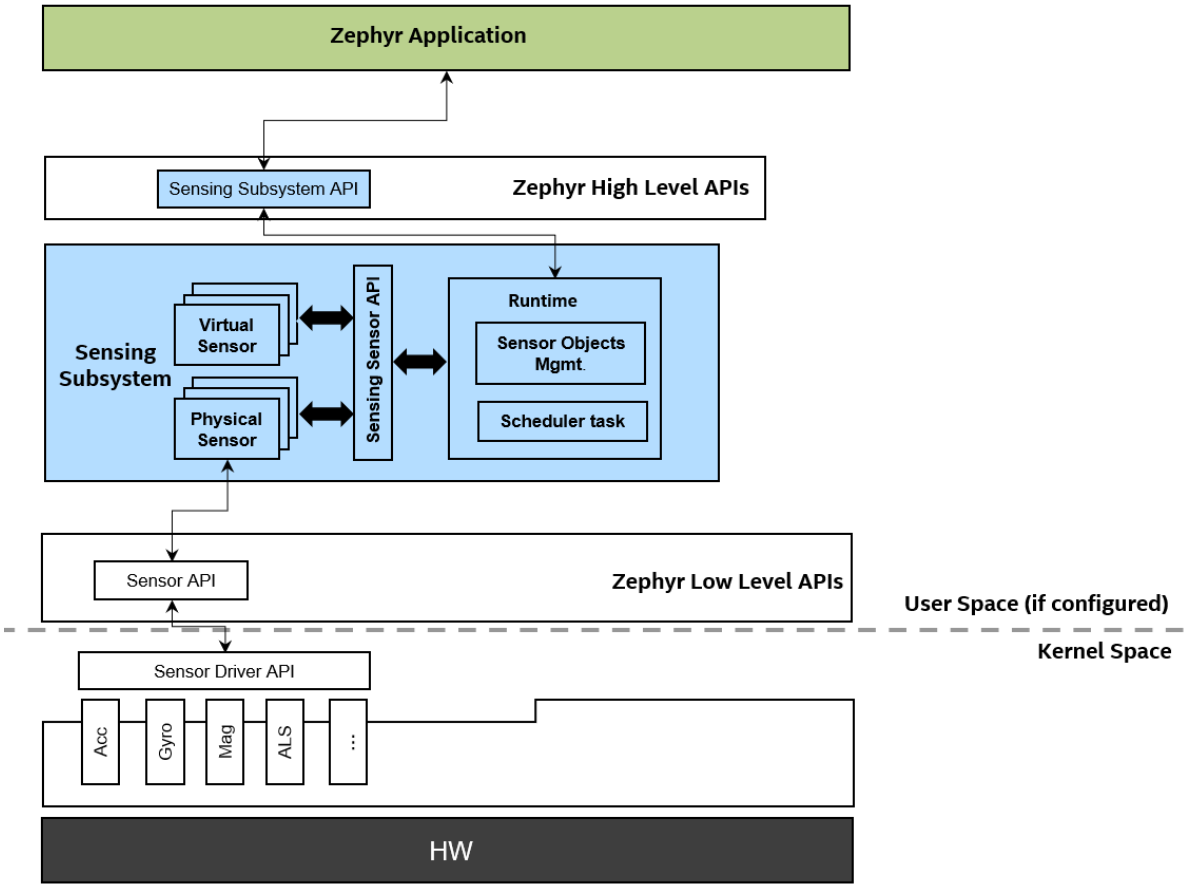
Data Driven Model

- **Polling mode**: periodical sampling rate
- **Interrupt mode**: data ready, threshold interrupt etc.

Scheduling
Single thread main loop for all sensor objects sampling and process.

Buffer Mode for Batching
Configurable Via Devicetree

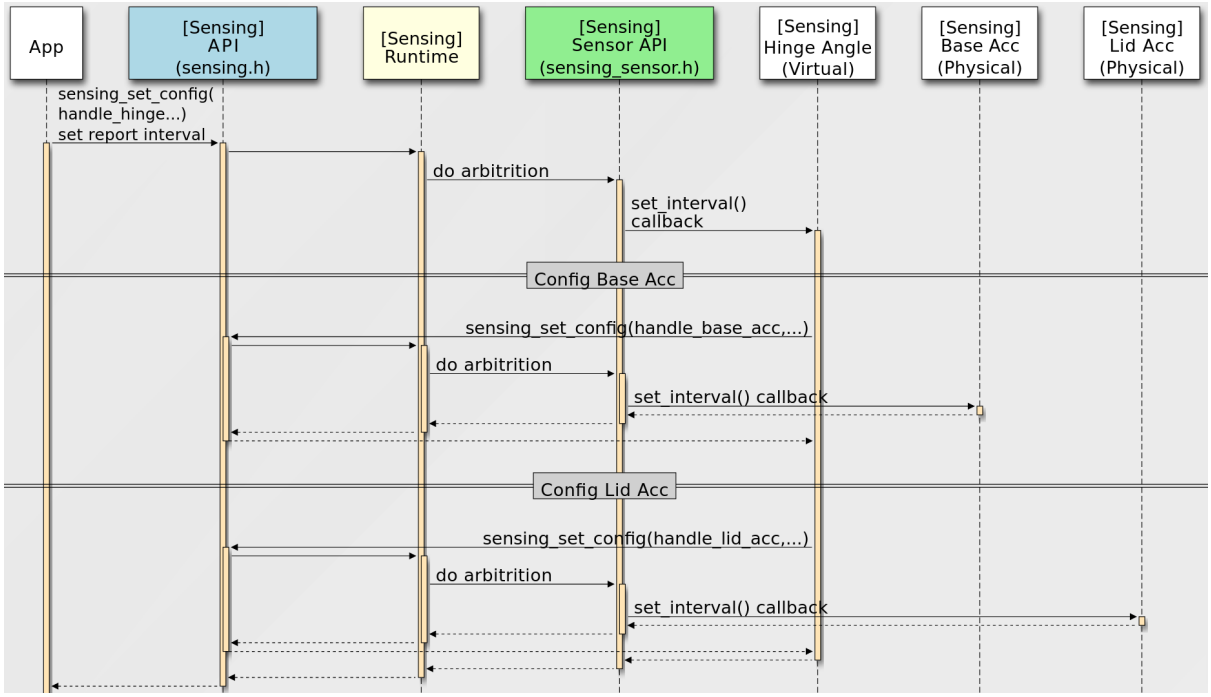
Below diagram shows the API position and scope:



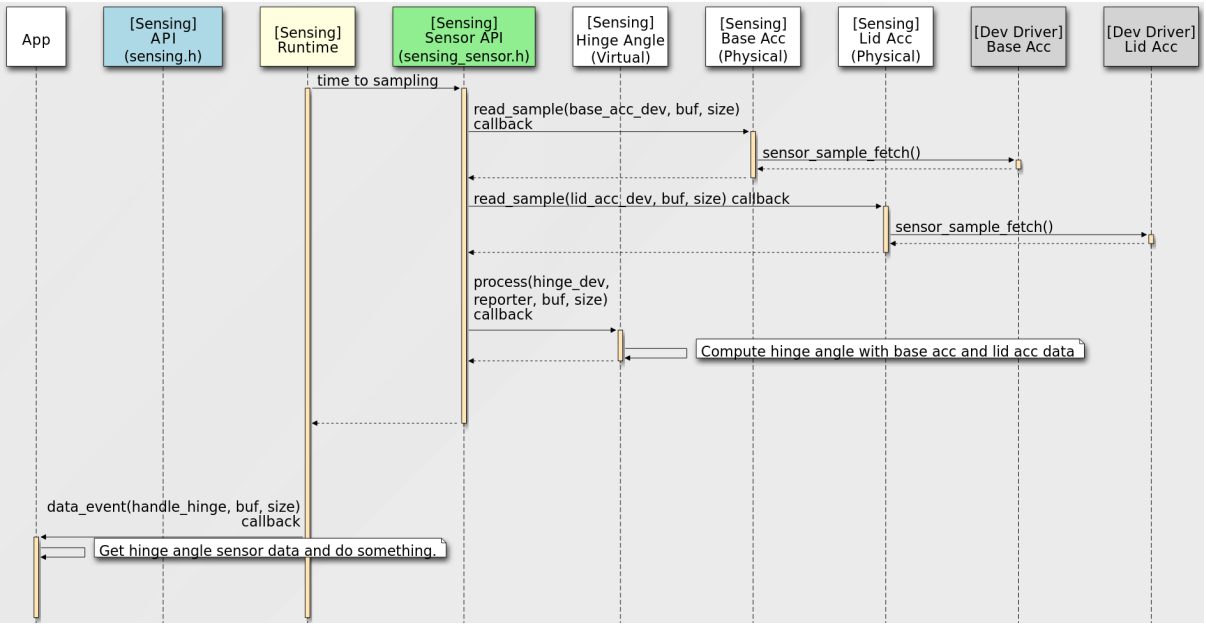
`Sensing Subsystem API` is for Applications. `Sensing Sensor API` is for development sensors.

Major Flows

- Sensor Configuration Flow



- Sensor Data Flow



Sensor Types And Instance

The `Sensing Subsystem` supports multiple instances of the same sensor type, there're two methods for Applications to identify and open an unique sensor instance:

- Enumerate all sensor instances

  `sensing_get_sensors()` returns all current board configuration supported sensor instances' information in a `sensing_sensor_info` pointer array .

  Then Applications can use `sensing_open_sensor()` to open specific sensor instance for future accessing, configuration and receive sensor data etc.

  This method is suitable for supporting some up-layer frameworks like `CHRE`, `HID` which need to dynamically enumerate the underlying platform's sensor instances.

- Open the sensor instance by devicetree node directly

  Applications can use `sensing_open_sensor_by_dt()` to open a sensor instance directly with sensor devicetree node identifier.

  For example:

```
sensing_open_sensor_by_dt(DEVICE_DT_GET(DT_NODELABEL(base_accel)), cb_list,␣
↪handle);
sensing_open_sensor_by_dt(DEVICE_DT_GET(DT_CHOSEN(zephyr_sensing_base_accel)),␣
↪cb_list, handle);
```

This method is useful and easy use for some simple Application which just want to access specific sensor(s).

`Sensor type` follows the HID standard sensor types definition.

See include/zephyr/sensing/sensing_sensor_types.h

Sensor Instance Handler

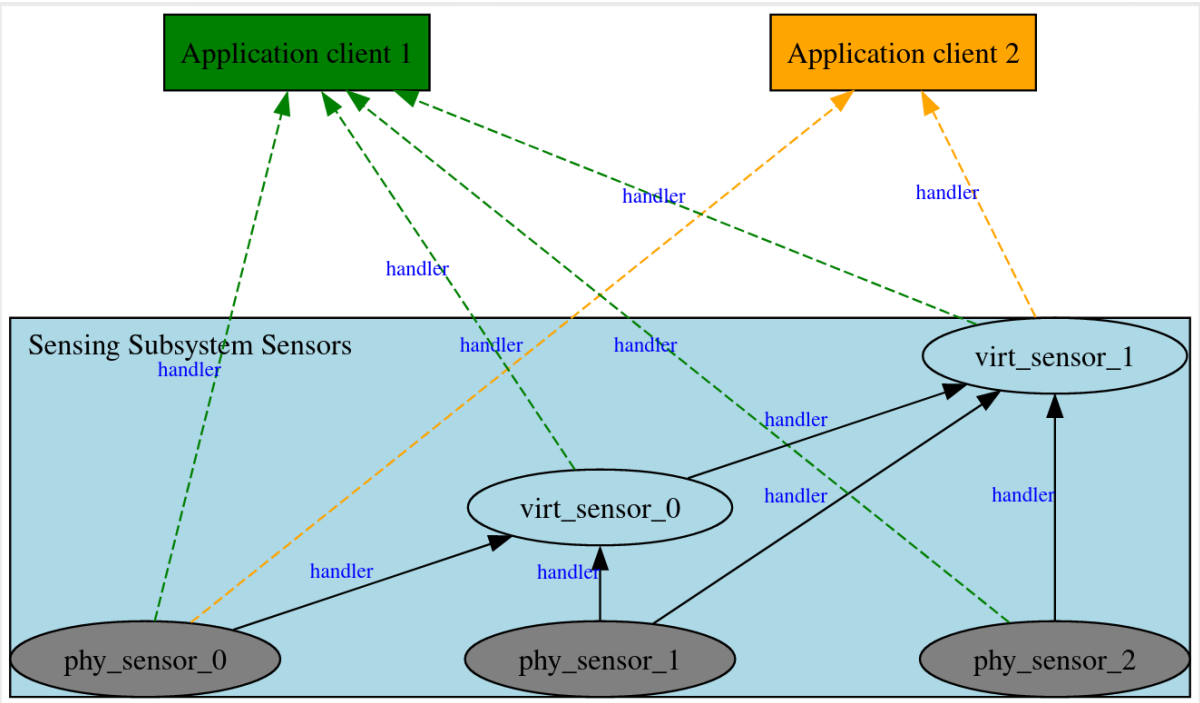
Clients using a `sensing_sensor_handle_t` type handler to handle a opened sensor instance, and all subsequent operations on this sensor instance need use this handler, such as set configurations, read sensor sample data, etc.

For a sensor instance, could have two kinds of clients: `Application clients` and `Sensor clients`.

`Application clients` can use `sensing_open_sensor()` to open a sensor instance and get it's handler.

For `Sensor clients`, there is no open API for opening a reporter, because the client-report relationship is built at the sensor's registration stage with devicetree.

The `Sensing Subsystem` will auto open and create `handlers` for client sensor to it's reporter sensors. `Sensor clients` can get it's reporters' handlers via `sensing_sensor_get_reporters()`.



Note

Sensors inside the Sensing Subsystem, the reporting relationship between them are all auto generated by Sensing Subsystem according devicetree definitions, handlers between client sensor and reporter sensors are auto created. Application(s) need to call `sensing_open_sensor()` to explicitly open the sensor instance.

Sensor Sample Value

- Data Structure

  Each sensor sample value defines as a common `header + readings[]` data structure, like `sensing_sensor_value_3d_q31`, `sensing_sensor_value_q31`, and `sensing_sensor_value_uint32`.

  The `header` definition `sensing_sensor_value_header()`.

- Time Stamp

  Time stamp unit in sensing subsystem is `micro seconds`.

  The `header` defines a **base_timestamp**, and each element in the **readings[]** array defines **timestamp_delta**.

  The **timestamp_delta** is in relation to the previous **readings** (or the **base_timestamp**)

  For example:

  – timestamp of `readings[0]` is `header.base_timestamp + readings[0].timestamp_delta`.

  – timestamp of `readings[1]` is `timestamp of readings[0] + readings[1].timestamp_delta`.

  Since timestamp unit is micro seconds, the max **timestamp_delta** (`uint32_t`) is `4295` seconds.

  If a sensor has batched data where two consecutive readings differ by more than `4295` seconds, the sensing subsystem runtime will split them across multiple instances of the readings structure, and send multiple events.

  This concept is referred from CHRE Sensor API.

- Data Format

  `Sensing Subsystem` uses per sensor type defined data format structure, and support `Q Format` defined in include/zephyr/dsp/types.h for `zdsp` lib support.

  For example `sensing_sensor_value_3d_q31` can be used by 3D IMU sensors like `SENSING_SENSOR_TYPE_MOTION_ACCELEROMETER_3D`, `SENSING_SENSOR_TYPE_MOTION_UNCALIB_ACCELEROMETER_3D`, and `SENSING_SENSOR_TYPE_MOTION_GYROMETER_3D`.

  `sensing_sensor_value_uint32` can be used by `SENSING_SENSOR_TYPE_LIGHT_AMBIENTLIGHT` sensor,

  and `sensing_sensor_value_q31` can be used by `SENSING_SENSOR_TYPE_MOTION_HINGE_ANGLE` sensor

  See include/zephyr/sensing/sensing_datatypes.h

Device Tree Configuration

Sensing subsystem using device tree to configuration all sensor instances and their properties, reporting relationships.

See the example samples/subsys/sensing/simple/boards/native_sim.overlay

API Reference

Sensing

Related code samples

- Sensing subsystemGet high-level sensor data in defined intervals.

4.11.4 State Machine Framework

Overview

The State Machine Framework (SMF) is an application agnostic framework that provides an easy way for developers to integrate state machines into their application. The framework can be added to any project by enabling the `CONFIG_SMF` option.

State Creation

A state is represented by three functions, where one function implements the Entry actions, another function implements the Run actions, and the last function implements the Exit actions. The prototype for the entry and exit functions are as follows: `void funct(void *obj)`, and the prototype for the run action is `enum smf_state_result funct(void *obj)` where the `obj` parameter is a user defined structure that has the state machine context, `smf_ctx`, as its first member. For example:

```
struct user_object {
   struct smf_ctx ctx;
   /* All User Defined Data Follows */
};
```

The `smf_ctx` member must be first because the state machine framework's functions casts the user defined object to the `smf_ctx` type with the `SMF_CTX` macro.

For example instead of doing this `(struct smf_ctx *)&user_obj`, you could use `SMF_CTX(&user_obj)`.

By default, a state can have no ancestor states, resulting in a flat state machine. But to enable the creation of a hierarchical state machine, the `CONFIG_SMF_ANCESTOR_SUPPORT` option must be enabled.

The return value of the run action, `smf_state_result` determines if the state machine propagates the event to parent run actions (`SMF_EVENT_PROPAGATE`) or if the event was handled by the run action (`SMF_EVENT_HANDLED`). Flat state machines do not have parent actions, so the return code is ignored; returning `SMF_EVENT_HANDLED` is recommended.

Calling `smf_set_state()` prevents calling parent run actions, even if `SMF_EVENT_PROPAGATE` is returned.

By default, the hierarchical state machines do not support initial transitions to child states on entering a superstate. To enable them the `CONFIG_SMF_INITIAL_TRANSITION` option must be enabled.

The following macro can be used for easy state creation:

- `SMF_CREATE_STATE` Create a state

State Machine Creation

A state machine is created by defining a table of states that's indexed by an enum. For example, the following creates three flat states:

```
enum demo_state { S0, S1, S2 };

const struct smf_state demo_states[] = {
   [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, NULL, NULL),
   [S1] = SMF_CREATE_STATE(s1_entry, s1_run, s1_exit, NULL, NULL),
   [S2] = SMF_CREATE_STATE(s2_entry, s2_run, s2_exit, NULL, NULL)
};
```

And this example creates three hierarchical states:

```
enum demo_state { S0, S1, S2 };

const struct smf_state demo_states[] = {
   [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, parent_s0, NULL),
   [S1] = SMF_CREATE_STATE(s1_entry, s1_run, s1_exit, parent_s12, NULL),
   [S2] = SMF_CREATE_STATE(s2_entry, s2_run, s2_exit, parent_s12, NULL)
};
```

This example creates three hierarchical states with an initial transition from parent state S0 to child state S2:

```
enum demo_state { S0, S1, S2 };

/* Forward declaration of state table */
const struct smf_state demo_states[];

const struct smf_state demo_states[] = {
   [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, NULL, demo_states[S2]),
   [S1] = SMF_CREATE_STATE(s1_entry, s1_run, s1_exit, demo_states[S0], NULL),
   [S2] = SMF_CREATE_STATE(s2_entry, s2_run, s2_exit, demo_states[S0], NULL)
};
```

To set the initial state, the `smf_set_initial()` function should be called.

To transition from one state to another, the `smf_set_state()` function is used.

Note

If `CONFIG_SMF_INITIAL_TRANSITION` is not set, `smf_set_initial()` and `smf_set_state()` function should not be passed a parent state as the parent state does not know which child state to transition to. Transitioning to a parent state is OK if an initial transition to a child state is defined. A well-formed HSM should have initial transitions defined for all parent states.

Note

While the state machine is running, `smf_set_state()` should only be called from the Entry or Run function. Calling `smf_set_state()` from Exit functions will generate a warning in the log and no transition will occur.

State Machine Execution

To run the state machine, the `smf_run_state()` function should be called in some application dependent way. An application should cease calling smf_run_state if it returns a non-zero value.

State Machine Termination

To terminate the state machine, the `smf_set_terminate()` function should be called. It can be called from the entry, run, or exit actions. The function takes a non-zero user defined value that will be returned by the `smf_run_state()` function.

Retrieving the Current State

Leaf State: In the context of a hierarchical state machine, a *leaf state* is a state that does not contain any child states. It represents the most granular level of state in the hierarchy, where no further decomposition is possible.

Executing State: The *executing state* refers to the state whose entry, run, or exit action is currently being executed by the state machine. This may be a parent or leaf state, depending on the current operation.

To retrieve the current leaf state, the `smf_get_current_leaf_state()` function should be called. For example:

```
const struct smf_state *leaf_state = smf_get_current_leaf_state(SMF_CTX(&s_obj));
```

> **Note**
>
> If `CONFIG_SMF_INITIAL_TRANSITION` is not enabled, or if the initial state of a parent state is not defined, always set the state to a leaf state. Otherwise, the state machine may enter a parent state directly, and `smf_get_current_leaf_state()` may return a parent state instead of a leaf state. Ensure initial transitions are properly configured for all parent states to avoid malformed hierarchical state machines.

To retrieve the state whose entry, run, or exit action is currently being executed, use the `smf_get_current_executing_state()` function.

UML State Machines

SMF follows UML hierarchical state machine rules for transitions i.e., the entry and exit actions of the least common ancestor are not executed on transition, unless said transition is a transition to self.

The UML Specification for StateMachines may be found in chapter 14 of the UML specification available here: https://www.omg.org/spec/UML/

SMF breaks from UML rules in:

1. Executing the actions associated with the transition within the context of the source state, rather than after the exit actions are performed.
2. Only allowing external transitions to self, not to sub-states. A transition from a superstate to a child state is treated as a local transition.
3. Prohibiting transitions using `smf_set_state()` in exit actions.

SMF also does not provide any pseudostates except the Initial Pseudostate. Terminate pseudostates can be modelled by calling `smf_set_terminate()` from the entry action of a 'terminate' state. Orthogonal regions are modelled by calling `smf_run_state()` for each region.

State Machine Examples

Flat State Machine Example This example turns the following state diagram into code using the SMF, where the initial state is S0.

Code:

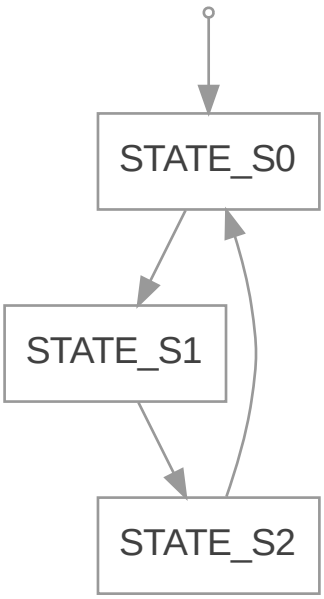


Fig. 50: Flat state machine diagram

```
#include <zephyr/smf.h>

/* Forward declaration of state table */
static const struct smf_state demo_states[];

/* List of demo states */
enum demo_state { S0, S1, S2 };

/* User defined object */
struct s_object {
        /* This must be first */
        struct smf_ctx ctx;

        /* Other state specific data add here */
} s_obj;

/* State S0 */
static void s0_entry(void *o)
{
        /* Do something */
}
static enum smf_state_result s0_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S1]);
        return SMF_EVENT_HANDLED;
}
static void s0_exit(void *o)
{
```

(continues on next page)

(continued from previous page)

```
        /* Do something */
}

/* State S1 */
static enum smf_state_result s1_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S2]);
        return SMF_EVENT_HANDLED;
}
static void s1_exit(void *o)
{
        /* Do something */
}

/* State S2 */
static void s2_entry(void *o)
{
        /* Do something */
}
static enum smf_state_result s2_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S0]);
        return SMF_EVENT_HANDLED;
}

/* Populate state table */
static const struct smf_state demo_states[] = {
        [S0] = SMF_CREATE_STATE(s0_entry, s0_run, s0_exit, NULL, NULL),
        /* State S1 does not have an entry action */
        [S1] = SMF_CREATE_STATE(NULL, s1_run, s1_exit, NULL, NULL),
        /* State S2 does not have an exit action */
        [S2] = SMF_CREATE_STATE(s2_entry, s2_run, NULL, NULL, NULL),
};

int main(void)
{
        int32_t ret;

        /* Set initial state */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Run the state machine */
        while(1) {
                /* State machine terminates if a non-zero value is returned */
                ret = smf_run_state(SMF_CTX(&s_obj));
                if (ret) {
                        /* handle return code and terminate state machine */
                        break;
                }
                k_msleep(1000);
        }
}
```

Hierarchical State Machine Example This example turns the following state diagram into code using the SMF, where S0 and S1 share a parent state and S0 is the initial state.

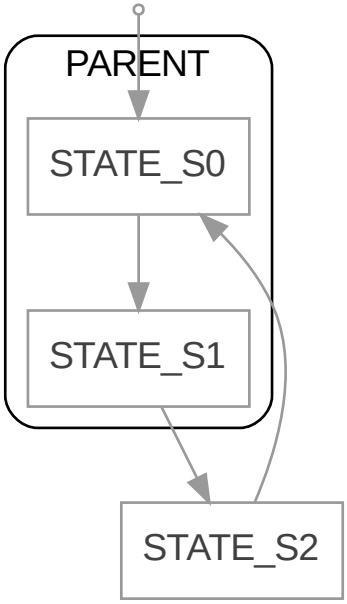


Fig. 51: Hierarchical state machine diagram

Code:

```
#include <zephyr/smf.h>

/* Forward declaration of state table */
static const struct smf_state demo_states[];

/* List of demo states */
enum demo_state { PARENT, S0, S1, S2 };

/* User defined object */
struct s_object {
        /* This must be first */
        struct smf_ctx ctx;

        /* Other state specific data add here */
} s_obj;

/* Parent State */
static void parent_entry(void *o)
{
        /* Do something */
}
static void parent_exit(void *o)
{
        /* Do something */
}
```

(continues on next page)

(continued from previous page)

```
/* State S0 */
static enum smf_state_result s0_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S1]);
        return SMF_EVENT_HANDLED;
}

/* State S1 */
static enum smf_state_result s1_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S2]);
        return SMF_EVENT_HANDLED;
}

/* State S2 */
static enum smf_state_result s2_run(void *o)
{
        smf_set_state(SMF_CTX(&s_obj), &demo_states[S0]);
        return SMF_EVENT_HANDLED;
}

/* Populate state table */
static const struct smf_state demo_states[] = {
        /* Parent state does not have a run action */
        [PARENT] = SMF_CREATE_STATE(parent_entry, NULL, parent_exit, NULL, NULL),
        /* Child states do not have entry or exit actions */
        [S0] = SMF_CREATE_STATE(NULL, s0_run, NULL, &demo_states[PARENT], NULL),
        [S1] = SMF_CREATE_STATE(NULL, s1_run, NULL, &demo_states[PARENT], NULL),
        /* State S2 do not have entry or exit actions and no parent */
        [S2] = SMF_CREATE_STATE(NULL, s2_run, NULL, NULL, NULL),
};

int main(void)
{
        int32_t ret;

        /* Set initial state */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Run the state machine */
        while(1) {
                /* State machine terminates if a non-zero value is returned */
                ret = smf_run_state(SMF_CTX(&s_obj));
                if (ret) {
                        /* handle return code and terminate state machine */
                        break;
                }
                k_msleep(1000);
        }
}
```

When designing hierarchical state machines, the following should be considered:

- Ancestor entry actions are executed before the sibling entry actions. For example, the parent_entry function is called before the s0_entry function.
- Transitioning from one sibling to another with a shared ancestry does not re-execute the ancestor's entry action or execute the exit action. For example, the parent_entry function

is not called when transitioning from S0 to S1, nor is the parent_exit function called.

- Ancestor exit actions are executed after the exit action of the current state. For example, the s1_exit function is called before the parent_exit function is called.
- The parent_run function only executes if the child_run function does not call either `smf_set_state()` or return `SMF_EVENT_HANDLED`.
- Avoid malformed hierarchical state machines by ensuring the state always transitions to a leaf state when `CONFIG_SMF_INITIAL_TRANSITION` is not enabled, or when a parent state's initial state is undefined.

Event Driven State Machine Example Events are not explicitly part of the State Machine Framework but an event driven state machine can be implemented using Zephyr *Events*.

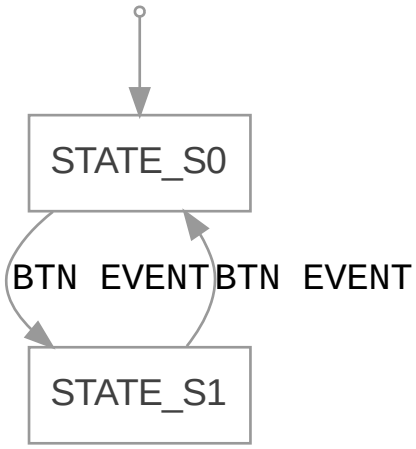


Fig. 52: Event driven state machine diagram

Code:

```
#include <zephyr/kernel.h>
#include <zephyr/drivers/gpio.h>
#include <zephyr/smf.h>

#define SW0_NODE        DT_ALIAS(sw0)

/* List of events */
#define EVENT_BTN_PRESS BIT(0)

static const struct gpio_dt_spec button =
        GPIO_DT_SPEC_GET_OR(SW0_NODE, gpios, {0});

static struct gpio_callback button_cb_data;

/* Forward declaration of state table */
static const struct smf_state demo_states[];

/* List of demo states */
```

(continues on next page)

(continued from previous page)

```
enum demo_state { S0, S1 };

/* User defined object */
struct s_object {
        /* This must be first */
        struct smf_ctx ctx;

        /* Events */
        struct k_event smf_event;
        int32_t events;

        /* Other state specific data add here */
} s_obj;

/* State S0 */
static void s0_entry(void *o)
{
        printk("STATE0\n");
}

static enum smf_state_result s0_run(void *o)
{
        struct s_object *s = (struct s_object *)o;

        /* Change states on Button Press Event */
        if (s->events & EVENT_BTN_PRESS) {
                smf_set_state(SMF_CTX(&s_obj), &demo_states[S1]);
        }
        return SMF_EVENT_HANDLED;
}

/* State S1 */
static void s1_entry(void *o)
{
        printk("STATE1\n");
}

static enum smf_state_result s1_run(void *o)
{
        struct s_object *s = (struct s_object *)o;

        /* Change states on Button Press Event */
        if (s->events & EVENT_BTN_PRESS) {
                smf_set_state(SMF_CTX(&s_obj), &demo_states[S0]);
        }
        return SMF_EVENT_HANDLED;
}

/* Populate state table */
static const struct smf_state demo_states[] = {
        [S0] = SMF_CREATE_STATE(s0_entry, s0_run, NULL, NULL, NULL),
        [S1] = SMF_CREATE_STATE(s1_entry, s1_run, NULL, NULL, NULL),
};

void button_pressed(const struct device *dev,
                struct gpio_callback *cb, uint32_t pins)
```

(continues on next page)

(continued from previous page)

```
{
        /* Generate Button Press Event */
        k_event_post(&s_obj.smf_event, EVENT_BTN_PRESS);
}

int main(void)
{
        int ret;

        if (!gpio_is_ready_dt(&button)) {
                printk("Error: button device %s is not ready\n",
                        button.port->name);
                return;
        }

        ret = gpio_pin_configure_dt(&button, GPIO_INPUT);
        if (ret != 0) {
                printk("Error %d: failed to configure %s pin %d\n",
                        ret, button.port->name, button.pin);
                return;
        }

        ret = gpio_pin_interrupt_configure_dt(&button,
                GPIO_INT_EDGE_TO_ACTIVE);
        if (ret != 0) {
                printk("Error %d: failed to configure interrupt on %s pin %d\n",
                        ret, button.port->name, button.pin);
                return;
        }

        gpio_init_callback(&button_cb_data, button_pressed, BIT(button.pin));
        gpio_add_callback(button.port, &button_cb_data);

        /* Initialize the event */
        k_event_init(&s_obj.smf_event);

        /* Set initial state */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Run the state machine */
        while(1) {
                /* Block until an event is detected */
                s_obj.events = k_event_wait(&s_obj.smf_event,
                                EVENT_BTN_PRESS, true, K_FOREVER);

                /* State machine terminates if a non-zero value is returned */
                ret = smf_run_state(SMF_CTX(&s_obj));
                if (ret) {
                        /* handle return code and terminate state machine */
                        break;
                }
        }
}
```

State Machine Example With Initial Transitions And Transition To Self tests/lib/smf/src/test_lib_self_transition_smf.c defines a state machine for testing the initial

transitions and transitions to self in a parent state. The statechart for this test is below.

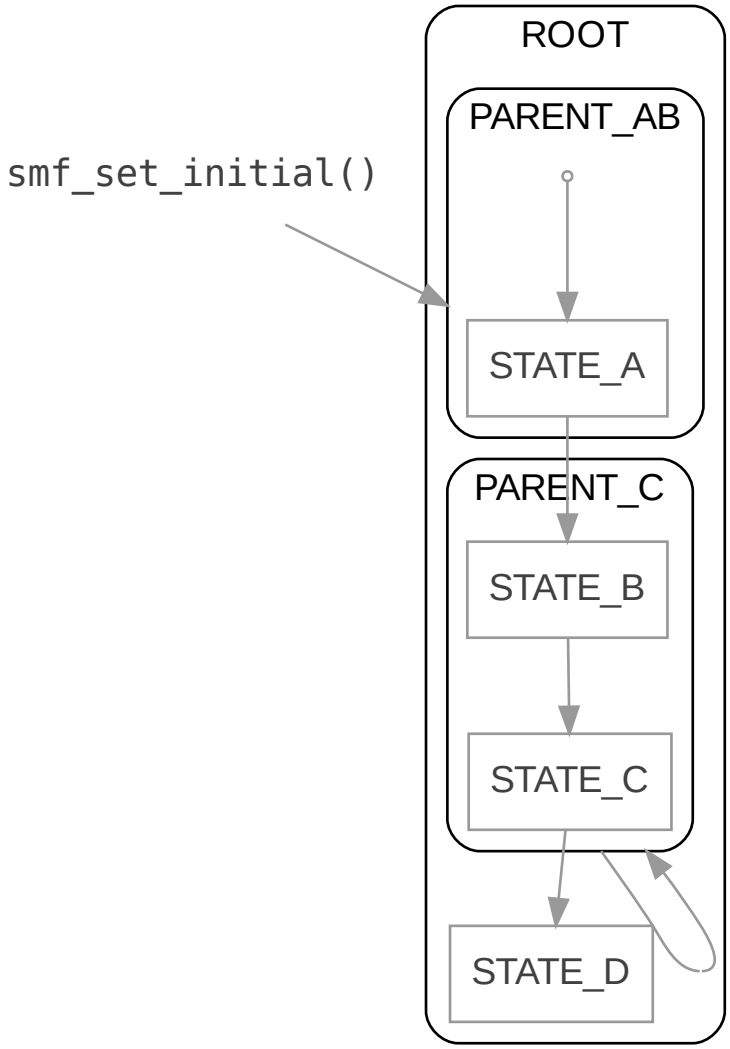


Fig. 53: Test state machine for UML State Transitions

Test Instrumentation

The SMF provides optional instrumentation hooks for observing state machine behavior during testing. To enable them, set `CONFIG_SMF_INSTRUMENTATION`.

When enabled, three hook callbacks can be registered on a state machine context via `smf_set_hooks()`:

- **on_action** — called before each entry, run, or exit action executes.
- **on_transition** — called after the current state pointer is updated but before the new state's entry actions execute.

- **on_error** — called when an invalid operation is detected (e.g. a NULL transition target or a transition attempted from an exit action).

> **Important**
>
> `smf_set_hooks()` must be called **after** `smf_set_initial()`, because `smf_set_initial()` resets the hooks pointer to `NULL`. As a consequence, entry actions executed during `smf_set_initial()` (i.e. the initial state's entry actions and those of its ancestors) will **not** be captured by the hooks.

Example:

```
#include <zephyr/smf.h>

static void on_action(struct smf_ctx *ctx,
                      const struct smf_state *state,
                      smf_action_type action_type)
{
        /* Log or record the action */
}

static void on_transition(struct smf_ctx *ctx,
                          const struct smf_state *source,
                          const struct smf_state *dest)
{
        /* Log or record the transition */
}

static const struct smf_hooks hooks = {
        .on_action = on_action,
        .on_transition = on_transition,
        /* .on_error = NULL - any member may be NULL */
};

void test_example(void)
{
        struct s_object s_obj;

        /* Set the initial state first */
        smf_set_initial(SMF_CTX(&s_obj), &demo_states[S0]);

        /* Install hooks after init - initial entry actions are not captured */
        smf_set_hooks(SMF_CTX(&s_obj), &hooks);

        /* Run the state machine - hooks fire on every action and transition */
        while (!smf_run_state(SMF_CTX(&s_obj))) {
                /* ... */
        }
}
```

When `CONFIG_SMF_INSTRUMENTATION` is not set, all instrumentation code is compiled out and there is zero runtime overhead.

API Reference

State Machine Framework API

Related code samples

- Hierarchical State Machine Demo based on example from PSiCC2Implement an event-driven hierarchical state machine using State Machine Framework (SMF).
- SMF CalculatorCreate a simple desk calculator using the State Machine Framework.

OS Abstraction

4.12 OS Abstraction

OS abstraction layers (OSAL) provide wrapper function APIs that encapsulate common system functions offered by any operating system. These APIs make it easier and quicker to develop for, and port code to multiple software and hardware platforms.

These sections describe the software and hardware abstraction layers supported by the Zephyr RTOS.

4.12.1 POSIX

Overview

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. Zephyr implements a subset of the standard POSIX API specified by IEEE 1003.1-2017 (also known as POSIX-1.2017).

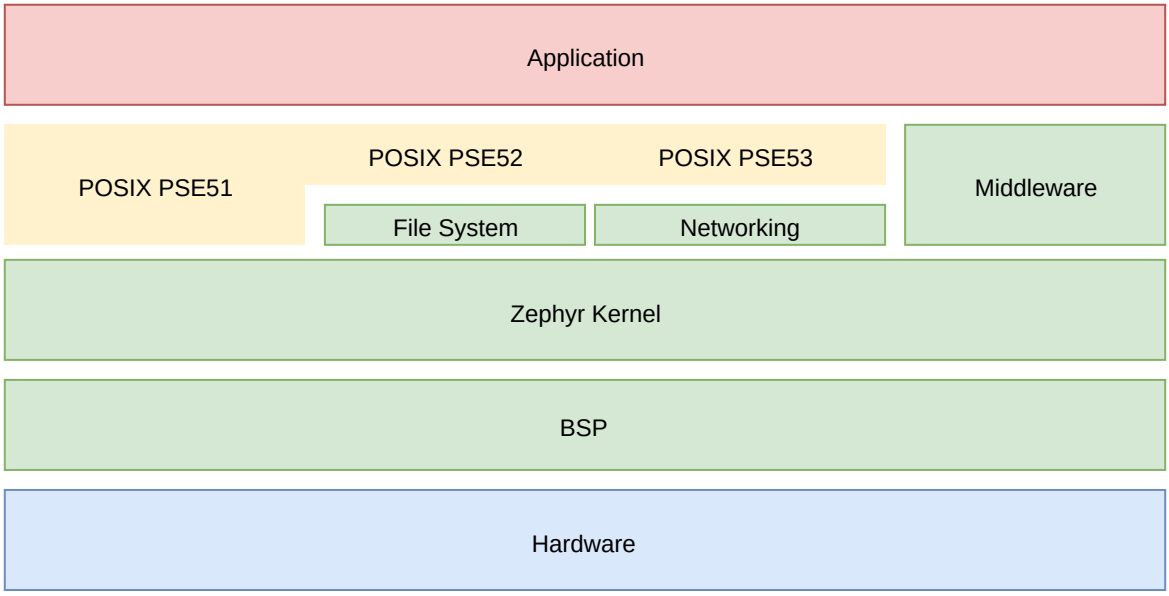


Fig. 54: POSIX support in Zephyr

Note

This page does not document Zephyr's POSIX architecture, which is used to run Zephyr as a native application under the host operating system for prototyping, test, and diagnostic purposes.

With the POSIX support available in Zephyr, an existing POSIX conformant application can be ported to run on the Zephyr kernel, and therefore leverage Zephyr features and functionality.

Additionally, a library designed to be POSIX conformant can be ported to Zephyr kernel based applications with no changes.

The POSIX API is an increasingly popular OSAL (operating system abstraction layer) for IoT and embedded applications, as can be seen in Zephyr, AWS:FreeRTOS, TI-RTOS, and NuttX.

Benefits of POSIX support in Zephyr include:

- Offering a familiar API to non-embedded programmers, especially from Linux
- Enabling reuse (portability) of existing libraries based on POSIX APIs
- Providing an efficient API subset appropriate for small (MCU) embedded systems

POSIX Subprofiles While Zephyr supports running multiple *threads* (possibly in an *SMP* configuration), as well as *Virtual Memory and MMUs*, Zephyr code and data normally share a common address space that is partitioned into separate *Memory Domains*. The Zephyr kernel executable code and the application executable code are typically compiled into the same binary artifact. From that perspective, Zephyr apps can be seen as running in the context of a single process.

While multi-purpose operating systems (OS) offer full POSIX conformance, Real-Time Operating Systems (RTOS) such as Zephyr typically serve a fixed-purpose, have limited hardware resources, and experience limited user interaction. In such systems, full POSIX conformance can be impractical and unnecessary.

For that reason, POSIX defined the following *Application Environment Profiles (AEP)* as part of IEEE 1003.13-2003 (also known as POSIX.13-2003). Each AEP adds incrementally more features over the required *POSIX System Interfaces*.

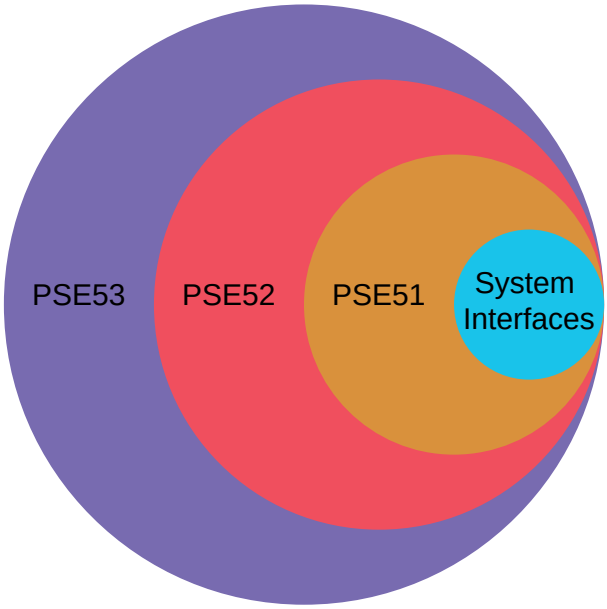


Fig. 55: POSIX Application Environment Profiles (AEP)

- Minimal Realtime System Profile (*PSE51*)
- Realtime Controller System Profile (*PSE52*)
- Dedicated Realtime System Profile (*PSE53*)
- Multi-Purpose Realtime System (PSE54)

POSIX.13-2003 AEP were formalized in 2003 via "Units of Functionality" but the specification is now inactive (for reference only). Nevertheless, the intent is still captured as part of POSIX-1.2017 via *Options* and *Option Groups*.

For more information, please see IEEE 1003.1-2017, Section E, Subprofiling Considerations.

POSIX Applications in Zephyr A POSIX app in Zephyr is *built like any other app* and therefore requires the usual `prj.conf`, `CMakeLists.txt`, and source code. For example, the app below leverages the `nanosleep()` and `perror()` POSIX functions.

Listing 23: *prj.conf* for a simple POSIX app in Zephyr

```
CONFIG_POSIX_API=y
```

Listing 24: A simple app that uses Zephyr's POSIX API

```
#include <stddef.h>
#include <stdio.h>
#include <time.h>

void megasleep(size_t megaseconds)
{
    struct timespec ts = {
        .tv_sec = megaseconds * 1000000,
        .tv_nsec = 0,
    };

    printf("See you in a while!\n");
    if (nanosleep(&ts, NULL) == -1) {
        perror("nanosleep");
    }
}

int main()
{
    megasleep(42);
    return 0;
}
```

For more examples of POSIX applications, please see the POSIX sample applications.

Configuration Like most features in Zephyr, POSIX features are *highly configurable* but disabled by default. Users must explicitly choose to enable POSIX options via *Kconfig* selection.

Subprofiles Enable one of the Kconfig options below to quickly configure a pre-defined *POSIX subprofile*.

- `CONFIG_POSIX_AEP_CHOICE_BASE` (*Base*)
- `CONFIG_POSIX_AEP_CHOICE_PSE51` (*PSE51*)
- `CONFIG_POSIX_AEP_CHOICE_PSE52` (*PSE52*)
- `CONFIG_POSIX_AEP_CHOICE_PSE53` (*PSE53*)

Additional POSIX *Options and Option Groups* may be enabled as needed via Kconfig (e.g. `CONFIG_POSIX_C_LIB_EXT=y`). Further fine-tuning may be accomplished via *additional POSIX-related Kconfig options*.

Subprofiles, Options, and Option Groups should be considered the preferred way to configure POSIX in Zephyr going forward.

Legacy Historically, Zephyr used `CONFIG_POSIX_API` to configure a set of POSIX features that was overloaded and always increasing in size.

- `CONFIG_POSIX_API`

The option is now frozen, and can be considered equivalent to the following:

- `CONFIG_POSIX_AEP_CHOICE_PSE51`
- `CONFIG_POSIX_FD_MGMT`
- `CONFIG_POSIX_MESSAGE_PASSING`
- `CONFIG_POSIX_NETWORKING`

However, `CONFIG_POSIX_API` should be considered legacy and should not be used for new Zephyr applications.

POSIX Conformance

As per IEEE 1003.1-2017, this section details Zephyr's POSIX conformance.

POSIX System Interfaces

Table 58: POSIX System Interfaces

Symbol	Support	Remarks
_POSIX_CHOWN_RESTRICTED	0	
_POSIX_NO_TRUNC	0	
_POSIX_VDISABLE	`'\0'`	

Table 59: POSIX System Interfaces

Symbol	Support	Remarks
_POSIX_VERSION	200809	
_POSIX_ASYNCHRONOUS_IO	200809	`CONFIG_POSIX_ASYNCHRONOUS_IO`†
_POSIX_BARRIERS	200809	`CONFIG_POSIX_BARRIERS`
_POSIX_CLOCK_SELECTION	200809	`CONFIG_POSIX_CLOCK_SELECTION`
_POSIX_MAPPED_FILES	200809	`CONFIG_POSIX_MAPPED_FILES`
_POSIX_MEMORY_PROTECTION	200809	`CONFIG_POSIX_MEMORY_PROTECTION` †
_POSIX_READER_WRITER_LOCKS	200809	`CONFIG_POSIX_RW_LOCKS`
_POSIX_REALTIME_SIGNALS	-1	`CONFIG_POSIX_REALTIME_SIGNALS`
_POSIX_SEMAPHORES	200809	`CONFIG_POSIX_SEMAPHORES`
_POSIX_SPIN_LOCKS	200809	`CONFIG_POSIX_SPIN_LOCKS`
_POSIX_THREAD_SAFE_FUNCTIONS	-1	`CONFIG_POSIX_THREAD_SAFE_FUNCTIONS`
_POSIX_THREADS	-1	`CONFIG_POSIX_THREADS`
_POSIX_TIMEOUTS	200809	`CONFIG_POSIX_TIMEOUTS`
_POSIX_TIMERS	200809	`CONFIG_POSIX_TIMERS`
_POSIX2_C_BIND	200809	

Table 60: POSIX System Interfaces (Unsupported)

Symbol	Support	Remarks
_POSIX_JOB_CONTROL	-1	†
_POSIX_REGEXP	-1	†
_POSIX_SAVED_IDS	-1	†
_POSIX_SHELL	-1	†

Table 61: POSIX System Interfaces (Optional)

Symbol	Support	Remarks
_POSIX_ADVISORY_INFO	-1	
_POSIX_CPUTIME	200809	CONFIG_POSIX_CPUTIME
_POSIX_FSYNC	200809	CONFIG_POSIX_FSYNC
_POSIX_IPV6	200809	CONFIG_POSIX_IPV6
_POSIX_MEMLOCK	200809	CONFIG_POSIX_MEMLOCK *†*
_POSIX_MEMLOCK_RANGE	200809	CONFIG_POSIX_MEMLOCK_RANGE
_POSIX_MESSAGE_PASSING	200809	CONFIG_POSIX_MESSAGE_PASSING
_POSIX_MONOTONIC_CLOCK	200809	CONFIG_POSIX_MONOTONIC_CLOCK
_POSIX_PRIORITIZED_IO	-1	
_POSIX_PRIORITY_SCHEDULING	200809	CONFIG_POSIX_PRIORITY_SCHEDULING
_POSIX_RAW_SOCKETS	200809	CONFIG_POSIX_RAW_SOCKETS
_POSIX_SHARED_MEMORY_OBJECTS	200809	CONFIG_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SPAWN	-1	*†*
_POSIX_SPORADIC_SERVER	-1	*†*
_POSIX_SYNCHRONIZED_IO	200809	CONFIG_POSIX_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	200809	CONFIG_POSIX_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	200809	CONFIG_POSIX_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	200809	CONFIG_POSIX_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	200809	CONFIG_POSIX_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	200809	CONFIG_POSIX_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	200809	CONFIG_POSIX_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PROCESS_SHARED	-1	
_POSIX_THREAD_SPORADIC_SERVER	-1	
_POSIX_TRACE	-1	
_POSIX_TRACE_EVENT_FILTER	-1	
_POSIX_TRACE_INHERIT	-1	
_POSIX_TRACE_LOG	-1	
_POSIX_TYPED_MEMORY_OBJECTS	-1	
_XOPEN_CRYPT	-1	
_XOPEN_REALTIME	700	CONFIG_XSI_REALTIME
_XOPEN_REALTIME_THREADS	-1	
_XOPEN_STREAMS	200809	CONFIG_XSI_STREAMS
_XOPEN_UNIX	-1	

POSIX Shell and Utilities Zephyr does not support a POSIX shell or utilities at this time.

Table 62: POSIX Shell and Utilities

Symbol	Support	Remarks
_POSIX2_C_DEV	-1	†
_POSIX2_CHAR_TERM	-1	†
_POSIX2_FORT_DEV	-1	†
_POSIX2_FORT_RUN	-1	†
_POSIX2_LOCALEDEF	-1	†
_POSIX2_PBS	-1	†
_POSIX2_PBS_ACCOUNTING	-1	†
_POSIX2_PBS_LOCATE	-1	†
_POSIX2_PBS_MESSAGE	-1	†
_POSIX2_PBS_TRACK	-1	†
_POSIX2_SW_DEV	-1	†
_POSIX2_UPE	-1	†
_POSIX2_UNIX	-1	†
_POSIX2_UUCP	-1	†

XSI Conformance

X/Open System Interfaces

Table 63: X/Open System Interfaces

Symbol	Support	Remarks
_POSIX_FSYNC	200809L	`CONFIG_POSIX_FSYNC`
_POSIX_THREAD_ATTR_STACKADDR	200809L	`CONFIG_POSIX_THREAD_ATTR_STACKADDR`
_POSIX_THREAD_ATTR_STACKSIZE	200809L	`CONFIG_POSIX_THREAD_ATTR_STACKSIZE`
_POSIX_THREAD_PROCESS_SHARED	-1	

Note

Some features may exhibit undefined behaviour as they fall beyond the scope of Zephyr's current design and capabilities. For example, multi-processing, ad-hoc memory-mapping, multiple users, or regular expressions are features that are uncommon in low-footprint embedded systems. Such undefined behaviour is denoted with the † (obelus) symbol. Additional details *here*.

Note

Features listed in various POSIX Options or Option Groups may be provided in whole or in part by a conformant C library implementation. This includes (but is not limited to) POSIX Extensions to the ISO C Standard (CX).

POSIX Application Environment Profiles (AEP)

Although inactive, IEEE 1003.13-2003 defined a number of AEP that inspired the modern subprofiling options of IEEE 1003.1-2017. The single-purpose realtime system profiles are listed below, for reference, in terms that agree with the current POSIX-1 standard. PSE54 is not considered at this time.

System Interfaces The required POSIX *System Interfaces* are supported for each Application Environment Profile.

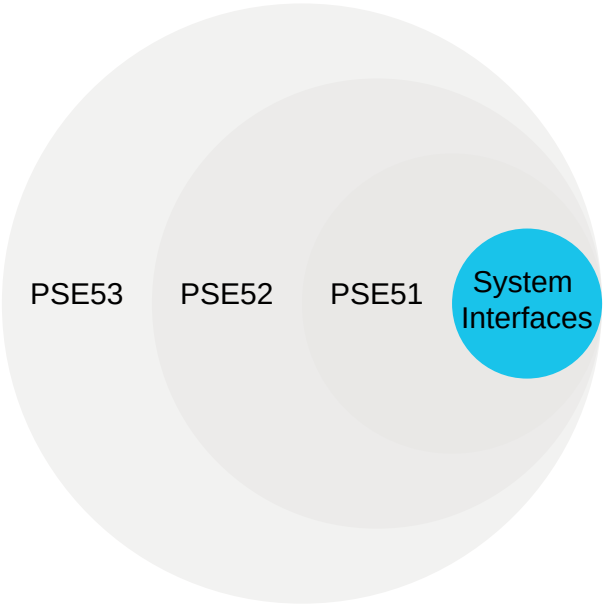


Fig. 56: System Interfaces

Minimal Realtime System Profile (PSE51) The *Minimal Realtime System Profile* (PSE51) includes all of the *System Interfaces* along with several additional features.

Table 64: PSE51 System Interfaces

Symbol	Support	Remarks
_POSIX_AEP_REALTIME_MINIMAL	-1	`CONFIG_POSIX_AEP_REALTIME_MINIMAL`

Table 65: PSE51 Option Groups

Symbol	Support	Remarks
POSIX_C_LANG_JUMP	yes	
POSIX_C_LANG_SUPPORT	yes	
POSIX_DEVICE_IO	yes	`CONFIG_POSIX_DEVICE_IO`
POSIX_SIGNALS	yes	`CONFIG_POSIX_SIGNALS` †
POSIX_SINGLE_PROCESS	yes	`CONFIG_POSIX_SINGLE_PROCESS`
XSI_THREADS_EXT	yes	`CONFIG_XSI_THREADS_EXT`

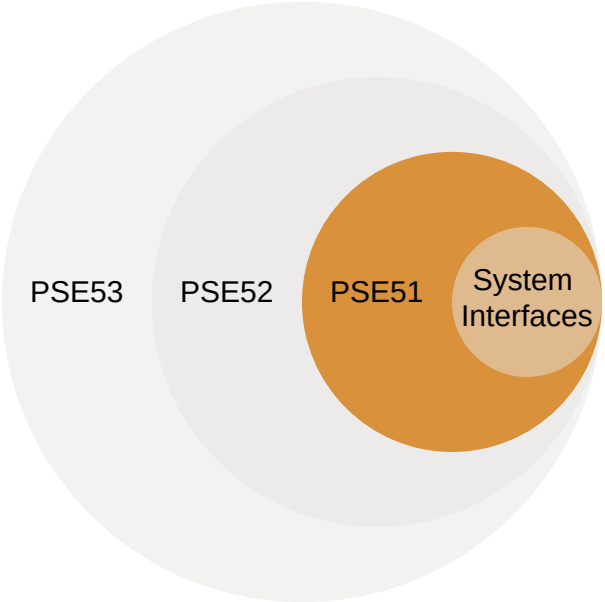


Fig. 57: Minimal Realtime System Profile (PSE51)

Table 66: PSE51 Option Requirements

Symbol	Support	Remarks
_POSIX_FSYNC	200809	`CONFIG_POSIX_FSYNC`
_POSIX_MEMLOCK	200809	`CONFIG_POSIX_MEMLOCK` †
_POSIX_MEMLOCK_RANGE	200809	`CONFIG_POSIX_MEMLOCK_RANGE`
_POSIX_MONOTONIC_CLOCK	200809	`CONFIG_POSIX_MONOTONIC_CLOCK`
_POSIX_SHARED_MEMORY_OBJECTS	200809	`CONFIG_POSIX_SHARED_MEMORY_OBJECTS`
_POSIX_SYNCHRONIZED_IO	200809	`CONFIG_POSIX_SYNCHRONIZED_IO`
_POSIX_THREAD_ATTR_STACKADDR	200809	`CONFIG_POSIX_THREAD_ATTR_STACKADDR`
_POSIX_THREAD_ATTR_STACKSIZE	200809	`CONFIG_POSIX_THREAD_ATTR_STACKSIZE`
_POSIX_THREAD_CPUTIME	200809	`CONFIG_POSIX_CPUTIME`
_POSIX_THREAD_PRIO_INHERIT	200809	`CONFIG_POSIX_THREAD_PRIO_INHERIT`
_POSIX_THREAD_PRIO_PROTECT	200809	`CONFIG_POSIX_THREAD_PRIO_PROTECT`
_POSIX_THREAD_PRIORITY_SCHEDULING	200809	`CONFIG_POSIX_THREAD_PRIORITY_SCHEDULING`
_POSIX_THREAD_SPORADIC_SERVER	-1	

Realtime Controller System Profile (PSE52) The *Realtime Controller System Profile* (PSE52) includes all features from PSE51 and the *System Interfaces*.

Table 67: PSE52 System Interfaces

Symbol	Support	Remarks
_POSIX_AEP_REALTIME_CONTROLLER	-1	`CONFIG_POSIX_AEP_REALTIME_CONTROLLER`

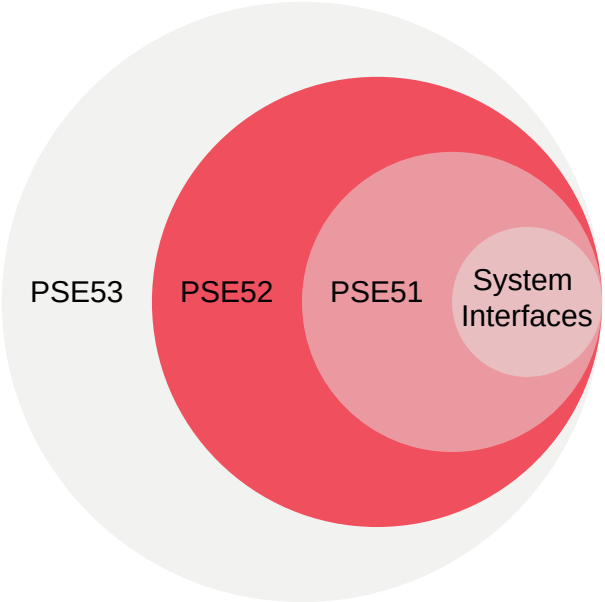


Fig. 58: Realtime Controller System Profile (PSE52)

Table 68: PSE52 Option Groups

Symbol	Support	Remarks
POSIX_C_LANG_MATH	yes	
POSIX_FD_MGMT		`CONFIG_POSIX_FD_MGMT`
POSIX_FILE_SYSTEM		`CONFIG_POSIX_FILE_SYSTEM`

Table 69: PSE52 Option Requirements

Symbol	Support	Remarks
_POSIX_MESSAGE_PASSING	200809L	`CONFIG_POSIX_MESSAGE_PASSING`
_POSIX_TRACE	-1	
_POSIX_TRACE_EVENT_FILTER	-1	
_POSIX_TRACE_LOG	-1	

Dedicated Realtime System Profile (PSE53) The *Dedicated Realtime System Profile* (PSE53) includes all features from PSE52, PSE51, and the *System Interfaces*.

Table 70: PSE53 System Interfaces

Symbol	Support	Remarks
_POSIX_AEP_REALTIME_DEDICATED	-1	`CONFIG_POSIX_AEP_REALTIME_DEDICATED`

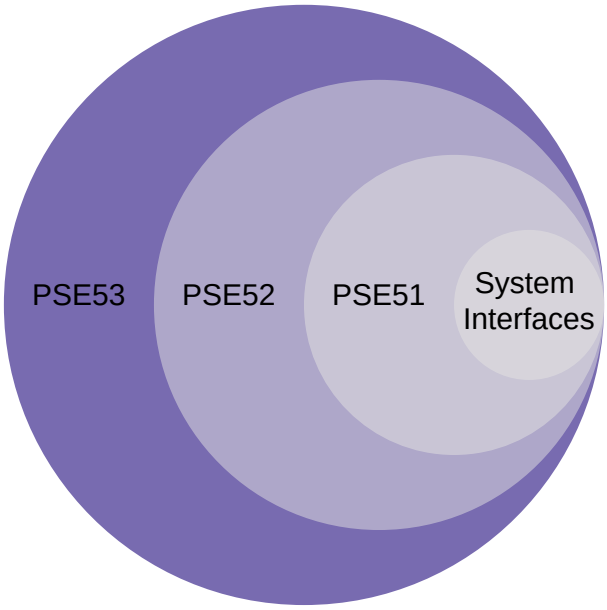


Fig. 59: Dedicated Realtime System Profile (PSE53)

Table 71: PSE53 Option Groups

Symbol	Support	Remarks
POSIX_MULTI_PROCESS		CONFIG_POSIX_MULTI_PROCESS†
POSIX_NETWORKING	yes	CONFIG_POSIX_NETWORKING
POSIX_PIPE		
POSIX_SIGNAL_JUMP		

Table 72: PSE53 Option Requirements

Symbol	Support	Remarks
_POSIX_CPUTIME	200809	CONFIG_POSIX_CPUTIME
_POSIX_PRIORITIZED_IO	-1	
_POSIX_PRIORITY_SCHEDULING	-1	
_POSIX_RAW_SOCKETS	200809	CONFIG_POSIX_RAW_SOCKETS
_POSIX_SPAWN	-1	†
_POSIX_SPORADIC_SERVER	-1	†

Implementation Details

In many ways, Zephyr provides support like any POSIX OS; API bindings are provided in the C programming language, POSIX headers are available in the standard include path, when configured.

Unlike other multi-purpose POSIX operating systems

- Zephyr is not "a POSIX OS". The Zephyr kernel was not designed around the POSIX standard, and POSIX support is an opt-in feature
- Zephyr apps are not linked separately, nor do they execute as subprocesses

- Zephyr, libraries, and application code are compiled and linked together, running similarly to a single-process application, in a single (possibly virtual) address space
- Zephyr does not provide a POSIX shell, compiler, utilities, and is not self-hosting.

> **Note**
>
> Unlike the Linux kernel or FreeBSD, Zephyr does not maintain a static table of system call numbers for each supported architecture, but instead generates system calls dynamically at build time. See *System Calls* for more information.

Design As a library, Zephyr's POSIX API implementation makes an effort to be a thin abstraction layer between the application, middleware, and the Zephyr kernel.

Some general design considerations:

- The POSIX interface and implementations should be part of Zephyr's POSIX library, and not elsewhere, unless required both by the POSIX API implementation and some other feature. An example where the implementation should remain part of the POSIX implementation is `getopt()`. Examples where the implementation should be part of separate libraries are multithreading and networking.
- When the POSIX API and another Zephyr subsystem both rely on a feature, the implementation of that feature should be as a separate Zephyr library that can be used by both the POSIX API and the other library or subsystem. This reduces the likelihood of dependency cycles in code. When practical, that rule should expand to include macros. In the example below, `libposix` depends on `libzfoo` for the implementation of some functionality "foo" in Zephyr. If `libzfoo` also depends on `libposix`, then there is a dependency cycle. The cycle can be removed via mutual dependency, `libcommon`.

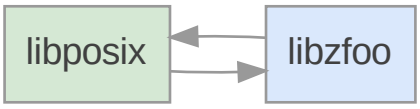


Fig. 60: Dependency cycle between POSIX and another Zephyr library

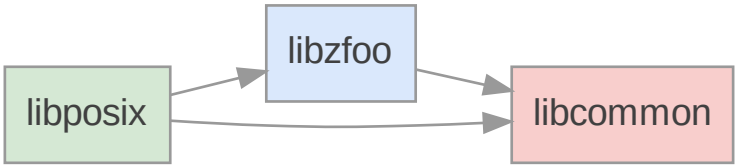


Fig. 61: Mutual dependencies between POSIX and other Zephyr libraries

- POSIX API calls should be provided as regular callable C functions; if a Zephyr *System Call* is needed as part of the implementation, the declaration and the implementation of that system call should be hidden behind the POSIX API.

POSIX Option and Option Group Details

POSIX Option Groups

POSIX_BARRIERS Enable this option group with `CONFIG_POSIX_BARRIERS`.

Table 73: POSIX_BARRIERS

API	Supported
pthread_barrier_destroy()	yes
pthread_barrier_init()	yes
pthread_barrier_wait()	yes
pthread_barrierattr_destroy()	yes
pthread_barrierattr_init()	yes

POSIX_C_LANG_JUMP The `POSIX_C_LANG_JUMP` Option Group is included in the ISO C standard.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the `POSIX_C_LANG_JUMP` Option Group is considered supported.

Table 74: POSIX_C_LANG_JUMP

API	Supported
setjmp()	yes
longjmp()	yes

POSIX_C_LANG_MATH The `POSIX_C_LANG_MATH` Option Group is included in the ISO C standard.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the `POSIX_C_LANG_MATH` Option Group is considered supported.

Please refer to Subprofiling Considerations for details on the `POSIX_C_LANG_MATH` Option Group.

POSIX_C_LANG_SUPPORT The POSIX_C_LANG_SUPPORT option group contains the general ISO C Library.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the entire `POSIX_C_LANG_SUPPORT` Option Group is considered supported.

Please refer to Subprofiling Considerations for details on the `POSIX_C_LANG_SUPPORT` Option Group.

For more information on developing Zephyr applications in the C programming language, please refer to *details*.

POSIX_C_LANG_SUPPORT_R Enable this option group with `CONFIG_POSIX_C_LANG_SUPPORT_R`.

Table 75: POSIX_C_LANG_SUPPORT_R

API	Supported
asctime_r()	yes
ctime_r()	yes
gmtime_r()	yes
localtime_r()	yes
rand_r()	yes
strerror_r()	yes
strtok_r()	yes

POSIX_C_LIB_EXT Enable this option group with `CONFIG_POSIX_C_LIB_EXT`.

Table 76: POSIX_C_LIB_EXT

API	Supported
fnmatch()	yes
getopt()	yes
getsubopt()	
optarg	yes
opterr	yes
optind	yes
optopt	yes
stpcpy()	
stpncpy()	
strcasecmp()	
strdup()	
strfmon()	
strncasecmp()	yes
strndup()	
strnlen()	yes

POSIX_CLOCK_SELECTION Enable this option group with `CONFIG_POSIX_CLOCK_SELECTION`.

Table 77: POSIX_CLOCK_SELECTION

API	Supported
pthread_condattr_getclock()	yes
pthread_condattr_setclock()	yes
clock_nanosleep()	yes

POSIX_DEVICE_IO Enable this option group with `CONFIG_POSIX_DEVICE_IO`.

> **Note**
>
> When using Newlib, Picolibc, or other C libraries conforming to the ISO C Standard, the C89 components of the `POSIX_DEVICE_IO` Option Group are considered supported.

Table 78: POSIX_DEVICE_IO

API	Supported
FD_CLR()	yes
FD_ISSET()	yes
FD_SET()	yes
FD_ZERO()	yes
clearerr()	yes
close()	yes
fclose()	yes
fdopen()	yes
feof()	yes
ferror()	yes
fflush()	yes
fgetc()	yes
fgets()	yes
fileno()	yes
fopen()	yes
fprintf()	yes
fputc()	yes
fputs()	yes
fread()	yes
freopen()	yes
fscanf()	yes
fwrite()	yes
getc()	yes
getchar()	yes
gets()	yes
open()	yes
perror()	yes
poll()	yes
printf()	yes
pread()	yes
pselect()	yes
putc()	yes
putchar()	yes
puts()	yes
pwrite()	yes
read()	yes
scanf()	yes
select()	yes
setbuf()	yes
setvbuf()	yes
stderr	yes
stdin	yes
stdout	yes
ungetc()	yes
vfprintf()	yes

continues on next page

Table 78 – continued from previous page

API	Supported
vfscanf()	yes
vprintf()	yes
vscanf()	yes
write()	yes

POSIX_FD_MGMT Enable this option group with `CONFIG_POSIX_FD_MGMT`.

Table 79: POSIX_FD_MGMT

API	Supported
dup()	
dup2()	
fcntl()	
fgetpos()	
fseek()	
fseeko()	
fsetpos()	
ftell()	
ftello()	
ftruncate()	yes
lseek()	
rewind()	

POSIX_FILE_LOCKING

Table 80: POSIX_FILE_LOCKING

API	Supported
flockfile()	
ftrylockfile()	
funlockfile()	
getc_unlocked()	
getchar_unlocked()	
putc_unlocked()	
putchar_unlocked()	

POSIX_FILE_SYSTEM Enable this option group with `CONFIG_POSIX_FILE_SYSTEM`.

Table 81: POSIX_FILE_SYSTEM

API	Supported
access()	
chdir()	
closedir()	yes
creat()	
fchdir()	
fpathconf()	
fstat()	yes
fstatvfs()	
getcwd()	
link()	
mkdir()	yes
mkstemp()	
opendir()	yes
pathconf()	
readdir()	yes
remove()	yes
rename()	yes
rewinddir()	
rmdir()	yes
stat()	yes
statvfs()	
tmpfile()	
tmpnam()	
truncate()	
unlink()	yes
utime()	

POSIX_FILE_SYSTEM_R Enable this option with `CONFIG_POSIX_FILE_SYSTEM_R`.

Table 82: POSIX_FILE_SYSTEM_R

API	Supported
readdir_r()	yes

POSIX_MAPPED_FILES Enable this option group with `CONFIG_POSIX_MAPPED_FILES`.

Table 83: POSIX_MAPPED_FILES

API	Supported
mmap()	yes
msync()	yes
munmap()	yes

POSIX_MEMORY_PROTECTION Enable this option group with `CONFIG_POSIX_MEMORY_PROTECTION`.

Table 84: POSIX_MEMORY_PROTECTION

API	Supported
mprotect()	yes †

POSIX_MULTI_PROCESS Enable this option group with `CONFIG_POSIX_MULTI_PROCESS`.

Table 85: POSIX_MULTI_PROCESS

API	Supported
_Exit()	yes
_exit()	yes
assert()	yes
atexit()	†
clock()	
execl()	†
execle()	†
execlp()	†
execv()	†
execve()	†
execvp()	†
exit()	yes
fork()	†
getpgrp()	†
getpgid()	†
getpid()	yes †
getppid()	†
getsid()	†
setsid()	†
sleep()	yes
times()	
wait()	†
waitid()	†
waitpid()	†

POSIX_NETWORKING The function `sockatmark()` is not yet supported and is expected to fail setting `errno` to `ENOSYS` †.

Enable this option group with `CONFIG_POSIX_NETWORKING`.

Table 86: POSIX_NETWORKING

API	Supported
accept()	yes
bind()	yes
connect()	yes
endhostent()	yes
endnetent()	yes
endprotoent()	yes
endservent()	yes
freeaddrinfo()	yes
gai_strerror()	yes
getaddrinfo()	yes
gethostent()	yes
gethostname()	yes
getnameinfo()	yes
getnetbyaddr()	yes
getnetbyname()	yes
getnetent()	yes
getpeername()	yes
getprotobyname()	yes

continues on next page

Table 86 – continued from previous page

API	Supported
getprotobynumber()	yes
getprotoent()	yes
getservbyname()	yes
getservbyport()	yes
getservent()	yes
getsockname()	yes
getsockopt()	yes
htonl()	yes
htons()	yes
if_freenameindex()	yes
if_indextoname()	yes
if_nameindex()	yes
if_nametoindex()	yes
inet_addr()	yes
inet_ntoa()	yes
inet_ntop()	yes
inet_pton()	yes
listen()	yes
ntohl()	yes
ntohs()	yes
recv()	yes
recvfrom()	yes
recvmsg()	yes
send()	yes
sendmsg()	yes
sendto()	yes
sethostent()	yes
setnetent()	yes
setprotoent()	yes
setservent()	yes
setsockopt()	yes
shutdown()	yes
socket()	yes
sockatmark()	yes †
socketpair()	yes

POSIX_PIPE

Table 87: POSIX_PIPE

API	Supported
pipe()	

POSIX_REALTIME_SIGNALS Enable this option group with `CONFIG_POSIX_REALTIME_SIGNALS`.

Table 88: POSIX_REALTIME_SIGNALS

API	Supported
sigqueue()	
sigtimedwait()	
sigwaitinfo()	

POSIX_RW_LOCKS Enable this option with `CONFIG_POSIX_RW_LOCKS`.

Table 89: POSIX_RW_LOCKS

API	Supported
pthread_rwlock_destroy()	yes
pthread_rwlock_init()	yes
pthread_rwlock_rdlock()	yes
pthread_rwlock_tryrdlock()	yes
pthread_rwlock_trywrlock()	yes
pthread_rwlock_unlock()	yes
pthread_rwlock_wrlock()	yes
pthread_rwlockattr_destroy()	yes
pthread_rwlockattr_getpshared()	yes
pthread_rwlockattr_init()	yes
pthread_rwlockattr_setpshared()	yes

POSIX_SEMAPHORES Enable this option group with `CONFIG_POSIX_SEMAPHORES`.

Table 90: POSIX_SEMAPHORES

API	Supported
sem_close()	yes
sem_destroy()	yes
sem_getvalue()	yes
sem_init()	yes
sem_open()	yes
sem_post()	yes
sem_trywait()	yes
sem_unlink()	yes
sem_wait()	yes

POSIX_SIGNAL_JUMP

Table 91: POSIX_SIGNAL_JUMP

API	Supported
siglongjmp()	
sigsetjmp()	

POSIX_SIGNALS Enable this option group with `CONFIG_POSIX_SIGNALS`.

Note

As processes are not yet supported in Zephyr, the ISO C functions `abort()`, `signal()`, and `raise()`, as well as the other POSIX functions listed below, may exhibit undefined behaviour. The POSIX functions `kill()`, `pause()`, `sigaction()`, `sigpending()`, `sigsuspend()`, and `sigwait()` are implemented to ensure that conformant applications can link, but they are expected to fail, setting errno to `ENOSYS` †.

Table 92: POSIX_SIGNALS

API	Supported
abort()	yes †
alarm()	yes †
kill()	yes †
pause()	yes †
raise()	yes †
sigaction()	yes †
sigaddset()	yes
sigdelset()	yes
sigemptyset()	yes
sigfillset()	yes
sigismember()	yes
signal()	yes †
sigpending()	yes †
sigprocmask()	yes
sigsuspend()	yes †
sigwait()	yes †
strsignal()	yes

POSIX_SINGLE_PROCESS The POSIX_SINGLE_PROCESS option group contains services for single process applications.

Enable this option group with `CONFIG_POSIX_SINGLE_PROCESS`.

Table 93: POSIX_SINGLE_PROCESS

API	Supported
confstr()	yes
environ	yes
errno	yes
getenv()	yes
setenv()	yes
sysconf()	yes
uname()	yes
unsetenv()	yes

POSIX_SPIN_LOCKS Enable this option group with `CONFIG_POSIX_SPIN_LOCKS`.

Table 94: POSIX_SPIN_LOCKS

API	Supported
pthread_spin_destroy()	yes
pthread_spin_init()	yes
pthread_spin_lock()	yes
pthread_spin_trylock()	yes
pthread_spin_unlock()	yes

POSIX_THREADS_BASE The basic assumption in this profile is that the system consists of a single (implicit) process with multiple threads. Therefore, the standard requires all basic thread services, except those related to multiple processes.

Enable this option group with `CONFIG_POSIX_THREADS`.

Table 95: POSIX_THREADS_BASE

API	Supported
pthread_atfork()	yes
pthread_attr_destroy()	yes
pthread_attr_getdetachstate()	yes
pthread_attr_getschedparam()	yes
pthread_attr_init()	yes
pthread_attr_setdetachstate()	yes
pthread_attr_setschedparam()	yes
pthread_barrier_destroy()	yes
pthread_barrier_init()	yes
pthread_barrier_wait()	yes
pthread_barrierattr_destroy()	yes
pthread_barrierattr_getpshared()	yes
pthread_barrierattr_init()	yes
pthread_barrierattr_setpshared()	yes
pthread_cancel()	yes
pthread_cleanup_pop()	yes
pthread_cleanup_push()	yes
pthread_cond_broadcast()	yes
pthread_cond_destroy()	yes
pthread_cond_init()	yes
pthread_cond_signal()	yes
pthread_cond_timedwait()	yes
pthread_cond_wait()	yes
pthread_condattr_destroy()	yes
pthread_condattr_init()	yes
pthread_create()	yes
pthread_detach()	yes
pthread_equal()	yes
pthread_exit()	yes
pthread_getspecific()	yes
pthread_join()	yes
pthread_key_create()	yes
pthread_key_delete()	yes
pthread_kill()	
pthread_mutex_destroy()	yes
pthread_mutex_init()	yes
pthread_mutex_lock()	yes
pthread_mutex_trylock()	yes

continues on next page

Table 95 – continued from previous page

API	Supported
pthread_mutex_unlock()	yes
pthread_mutexattr_destroy()	yes
pthread_mutexattr_init()	yes
pthread_once()	yes
pthread_self()	yes
pthread_setcancelstate()	yes
pthread_setcanceltype()	yes
pthread_setspecific()	yes
pthread_sigmask()	yes
pthread_testcancel()	yes
sched_yield()	yes

POSIX_THREADS_EXT Enable this option group with `CONFIG_POSIX_THREADS_EXT`.

Table 96: POSIX_THREADS_EXT

API	Supported
pthread_attr_getguardsize()	yes
pthread_attr_setguardsize()	yes
pthread_mutexattr_gettype()	yes
pthread_mutexattr_settype()	yes

POSIX_TIMERS Enable this option group with `CONFIG_POSIX_TIMERS`.

Table 97: POSIX_TIMERS

API	Supported
clock_getres()	yes
clock_gettime()	yes
clock_settime()	yes
nanosleep()	yes
timer_create()	yes
timer_delete()	yes
timer_gettime()	yes
timer_getoverrun()	yes
timer_settime()	yes

XSI_REALTIME The `XSI_REALTIME` option group indicates that the *_POSIX_FSYNC*, *_POSIX_MEMLOCK*, *_POSIX_MEMLOCK_RANGE*, *_POSIX_MESSAGE_PASSING*, *_POSIX_PRIORITY_SCHEDULING*, *_POSIX_SHARED_MEMORY_OBJECTS*, and *_POSIX_SYNCHRONIZED_IO* options are enabled.

Enable this option group with `CONFIG_XSI_REALTIME`.

When this option group is enabled, the `_XOPEN_REALTIME` feature test macro will be defined to a value other than -1.

XSI_SINGLE_PROCESS Enable this option group with `CONFIG_XSI_SINGLE_PROCESS`.

Table 98: XSI_SINGLE_PROCESS

API	Supported
gethostid()	yes
gettimeofday()	yes
putenv()	yes

XSI_SYSTEM_LOGGING Enable this option group with `CONFIG_XSI_SYSTEM_LOGGING`.

Table 99: XSI_SYSTEM_LOGGING

API	Supported
closelog()	yes
openlog()	yes
setlogmask()	yes
syslog()	yes

XSI_THREADS_EXT The XSI_THREADS_EXT option group is required because it provides functions to control a thread's stack. This is considered useful for any real-time application.

Enable this option group with `CONFIG_XSI_THREADS_EXT`.

Table 100: XSI_THREADS_EXT

API	Supported
pthread_attr_getstack()	yes
pthread_attr_setstack()	yes
pthread_getconcurrency()	yes
pthread_setconcurrency()	yes

POSIX Options

_POSIX_ASYNCHRONOUS_IO Functions part of the `_POSIX_ASYNCHRONOUS_IO` Option are not implemented in Zephyr but are provided so that conformant applications can still link. These functions will fail, setting `errno` to `ENOSYS`†.

Enable this option with `CONFIG_POSIX_ASYNCHRONOUS_IO`.

Table 101: _POSIX_ASYNCHRONOUS_IO

API	Supported
aio_cancel()	yes †
aio_error()	yes †
aio_fsync()	yes †
aio_read()	yes †
aio_return()	yes †
aio_suspend()	yes †
aio_write()	yes †
lio_listio()	yes †

_POSIX_CPUTIME Enable this option with `CONFIG_POSIX_CPUTIME`.

Table 102: _POSIX_CPUTIME

API	Supported
CLOCK_PROCESS_CPUTIME_ID	yes

_POSIX_FSYNC Enable this option with `CONFIG_POSIX_FSYNC`.

Table 103: _POSIX_FSYNC

API	Supported
fsync()	yes

_POSIX_IPV6 Internet Protocol Version 6 is supported.

For more information, please refer to *Networking*.

Enable this option with `CONFIG_POSIX_IPV6`.

_POSIX_MEMLOCK Zephyr's *Demand Paging API* does not yet support pinning or unpinning all virtual memory regions. The functions below are expected to fail and set `errno` to `ENOSYS` †.

Enable this option with `CONFIG_POSIX_MEMLOCK`.

Table 104: _POSIX_MEMLOCK

API	Supported
mlockall()	yes
munlockall()	yes

_POSIX_MEMLOCK_RANGE Enable this option with `CONFIG_POSIX_MEMLOCK_RANGE`.

Table 105: _POSIX_MEMLOCK_RANGE

API	Supported
mlock()	yes
munlock()	yes

_POSIX_MESSAGE_PASSING Enable this option with `CONFIG_POSIX_MESSAGE_PASSING`.

Table 106: _POSIX_MESSAGE_PASSING

API	Supported
mq_close()	yes
mq_getattr()	yes
mq_notify()	yes
mq_open()	yes
mq_receive()	yes
mq_send()	yes
mq_setattr()	yes
mq_unlink()	yes

_POSIX_MONOTONIC_CLOCK Enable this option with `CONFIG_POSIX_MONOTONIC_CLOCK`.

Table 107: _POSIX_MONOTONIC_CLOCK

API	Supported
CLOCK_MONOTONIC	yes

_POSIX_PRIORITY_SCHEDULING As processes are not yet supported in Zephyr, the functions `sched_rr_get_interval()`, `sched_setparam()`, and `sched_setscheduler()` are expected to fail setting `errno` to `ENOSYS`†.

Enable this option with `CONFIG_POSIX_PRIORITY_SCHEDULING`.

Table 108: _POSIX_PRIORITY_SCHEDULING

API	Supported
sched_get_priority_max()	yes
sched_get_priority_min()	yes
sched_getparam()	yes
sched_getscheduler()	yes
sched_rr_get_interval()	yes †
sched_setparam()	yes †
sched_setscheduler()	yes †

_POSIX_RAW_SOCKETS Raw sockets are supported.

For more information, please refer to `CONFIG_NET_SOCKETS_PACKET`.

Enable this option with `CONFIG_POSIX_RAW_SOCKETS`.

_POSIX_SHARED_MEMORY_OBJECTS Enable this option with `CONFIG_POSIX_SHARED_MEMORY_OBJECTS`.

Table 109: _POSIX_SHARED_MEMORY_OBJECTS

API	Supported
mmap()	yes
munmap()	yes
shm_open()	yes
shm_unlink()	yes

_POSIX_SYNCHRONIZED_IO Enable this option with `CONFIG_POSIX_SYNCHRONIZED_IO`.

Table 110: _POSIX_SYNCHRONIZED_IO

API	Supported
fdatasync()	yes
fsync()	yes
msync()	yes

_POSIX_THREAD_ATTR_STACKADDR Enable this option with `CONFIG_POSIX_THREAD_ATTR_STACKADDR`.

Table 111: _POSIX_THREAD_ATTR_STACKADDR

API	Supported
pthread_attr_getstackaddr()	yes
pthread_attr_setstackaddr()	yes

_POSIX_THREAD_ATTR_STACKSIZE Enable this option with `CONFIG_POSIX_THREAD_ATTR_STACKSIZE`.

Table 112: _POSIX_THREAD_ATTR_STACKSIZE

API	Supported
pthread_attr_getstacksize()	yes
pthread_attr_setstacksize()	yes

_POSIX_THREAD_CPUTIME Enable this option with `CONFIG_POSIX_THREAD_CPUTIME`.

Table 113: _POSIX_THREAD_CPUTIME

API	Supported
CLOCK_THREAD_CPUTIME_ID	yes
pthread_getcpuclockid()	yes

_POSIX_THREAD_PRIO_INHERIT Enable this option with `CONFIG_POSIX_THREAD_PRIO_INHERIT`.

Table 114: _POSIX_THREAD_PRIO_INHERIT

API	Supported
pthread_mutexattr_getprotocol()	yes
pthread_mutexattr_setprotocol()	yes

_POSIX_THREAD_PRIO_PROTECT Enable this option with `CONFIG_POSIX_THREAD_PRIO_PROTECT`.

Table 115: _POSIX_THREAD_PRIO_PROTECT

API	Supported
pthread_mutex_getprioceiling()	yes
pthread_mutex_setprioceiling()	yes
pthread_mutexattr_getprioceiling()	yes
pthread_mutexattr_getprotocol()	yes
pthread_mutexattr_setprioceiling()	yes
pthread_mutexattr_setprotocol()	yes

_POSIX_THREAD_PRIORITY_SCHEDULING Enable this option with `CONFIG_POSIX_THREAD_PRIORITY_SCHEDULING`.

Table 116: _POSIX_THREAD_PRIORITY_SCHEDULING

API	Supported
pthread_attr_getinheritsched()	yes
pthread_attr_getschedpolicy()	yes
pthread_attr_getscope()	yes
pthread_attr_setinheritsched()	yes
pthread_attr_setschedpolicy()	yes
pthread_attr_setscope()	yes
pthread_getschedparam()	yes
pthread_setschedparam()	yes
pthread_setschedprio()	yes

_POSIX_THREAD_SAFE_FUNCTIONS Enable this option with `CONFIG_POSIX_THREAD_SAFE_FUNCTIONS`.

Table 117: _POSIX_THREAD_SAFE_FUNCTIONS

API	Supported
asctime_r()	yes
ctime_r()	yes (UTC timezone only)
flockfile()	
ftrylockfile()	
funlockfile()	
getc_unlocked()	
getchar_unlocked()	
getgrgid_r()	yes †
getgrnam_r()	yes †
getpwnam_r()	yes †
getpwuid_r()	yes †
gmtime_r()	yes
localtime_r()	yes (UTC timezone only)
putc_unlocked()	
putchar_unlocked()	
rand_r()	yes
readdir_r()	yes
strerror_r()	yes
strtok_r()	yes

_POSIX_TIMEOUTS Enable this option with `CONFIG_POSIX_TIMEOUTS`.

Table 118: _POSIX_TIMEOUTS

API	Supported
mq_timedreceive()	yes
mq_timedsend()	yes
pthread_mutex_timedlock()	yes
pthread_rwlock_timedrdlock()	yes
pthread_rwlock_timedwrlock()	yes
sem_timedwait()	yes
posix_trace_timedgetnext_event()	

_XOPEN_STREAMS With the exception of `ioctl()`, functions in the `_XOPEN_STREAMS` option group are not implemented in Zephyr but are provided so that conformant applications can still link. Unimplemented functions in this option group will fail, setting `errno` to `ENOSYS` †.

Enable this option with `CONFIG_XSI_STREAMS`.

Table 119: _XOPEN_STREAMS

API	Supported
fattach()	yes †
fdetach()	yes †
getmsg()	yes †
getpmsg()	yes †
ioctl()	yes
isastream()	yes †
putmsg()	yes †
putpmsg()	yes †

Additional Configuration Options

Below is a non-exhaustive list of additional *Configuration System (Kconfig)* options relating to Zephyr's implementation of the POSIX API.

- `CONFIG_DYNAMIC_THREAD`
- `CONFIG_DYNAMIC_THREAD_POOL_SIZE`
- `CONFIG_EVENTFD`
- `CONFIG_GETOPT_LONG`
- `CONFIG_MAX_PTHREAD_SPINLOCK_COUNT`
- `CONFIG_MQUEUE_NAMELEN_MAX`
- `CONFIG_POSIX_MQ_OPEN_MAX`
- `CONFIG_MSG_SIZE_MAX`
- `CONFIG_NET_SOCKETPAIR`
- `CONFIG_NET_SOCKETS`
- `CONFIG_ZVFS_POLL_MAX`
- `CONFIG_ZVFS_OPEN_MAX`
- `CONFIG_POSIX_API`
- `CONFIG_POSIX_OPEN_MAX`
- `CONFIG_POSIX_PTHREAD_ATTR_GUARDSIZE_BITS`
- `CONFIG_POSIX_PTHREAD_ATTR_GUARDSIZE_DEFAULT`
- `CONFIG_POSIX_PTHREAD_ATTR_STACKSIZE_BITS`
- `CONFIG_POSIX_RTSIG_MAX`
- `CONFIG_POSIX_SIGNAL_STRING_DESC`
- `CONFIG_POSIX_THREAD_KEYS_MAX`
- `CONFIG_POSIX_THREAD_THREADS_MAX`
- `CONFIG_POSIX_UNAME_NODENAME_LEN`
- `CONFIG_POSIX_UNAME_VERSION_LEN`
- `CONFIG_PTHREAD_CREATE_BARRIER`

- CONFIG_PTHREAD_RECYCLER_DELAY_MS
- CONFIG_POSIX_SEM_NAMELEN_MAX
- CONFIG_POSIX_SEM_NSEMS_MAX
- CONFIG_POSIX_SEM_VALUE_MAX
- CONFIG_TIMER_CREATE_WAIT
- CONFIG_THREAD_STACK_INFO
- CONFIG_ZVFS_EVENTFD_MAX
- CONFIG_ZVFS_FDTABLE

4.12.2 CMSIS RTOS v1

Cortex-M Software Interface Standard (CMSIS) RTOS is a vendor-independent hardware abstraction layer for the ARM Cortex-M processor series and defines generic tool interfaces. Though it was originally defined for ARM Cortex-M microcontrollers alone, it could be easily extended to other microcontrollers making it generic. For more information on CMSIS RTOS v1, please refer to the CMSIS-RTOS1 Documentation.

4.12.3 CMSIS RTOS v2

Cortex-M Software Interface Standard (CMSIS) RTOS is a vendor-independent hardware abstraction layer for the ARM Cortex-M processor series and defines generic tool interfaces. Though it was originally defined for ARM Cortex-M microcontrollers alone, it could be easily extended to other microcontrollers making it generic. For more information on CMSIS RTOS v2, please refer to the CMSIS-RTOS2 Documentation.

Features not supported in Zephyr implementation

Kernel
: `osKernelGetState`, `osKernelSuspend`, `osKernelResume`, `osKernelInitialize` and `osKernelStart` are not supported.

Mutex
: `osMutexPrioInherit` is supported by default and is not configurable, you cannot select/unselect this attribute.

  `osMutexRecursive` is also supported by default. If this attribute is not set, an error is thrown when the same thread tries to acquire it the second time.

  `osMutexRobust` is not supported in Zephyr.

Return values not supported in the Zephyr implementation

`osKernelUnlock`, `osKernelLock`, `osKernelRestoreLock`
: `osError` (Unspecified error) is not supported.

`osSemaphoreDelete`
: `osErrorResource` (the semaphore specified by parameter semaphore_id is in an invalid semaphore state) is not supported.

`osMutexDelete`
: `osErrorResource` (mutex specified by parameter mutex_id is in an invalid mutex state) is not supported.

`osTimerDelete`
: `osErrorResource` (the timer specified by parameter timer_id is in an invalid timer state) is not supported.

`osMessageQueueReset`
: `osErrorResource` (the message queue specified by parameter msgq_id is in an invalid message queue state) is not supported.

`osMessageQueueDelete`
: `osErrorResource` (the message queue specified by parameter msgq_id is in an invalid message queue state) is not supported.

`osMemoryPoolFree`
: `osErrorResource` (the memory pool specified by parameter mp_id is in an invalid memory pool state) is not supported.

`osMemoryPoolDelete`
: `osErrorResource` (the memory pool specified by parameter mp_id is in an invalid memory pool state) is not supported.

`osEventFlagsSet`, `osEventFlagsClear`
: `osFlagsErrorUnknown` (Unspecified error) and osFlagsErrorResource (Event flags object specified by parameter ef_id is not ready to be used) are not supported.

`osEventFlagsDelete`
: `osErrorParameter` (the value of the parameter ef_id is incorrect) is not supported.

`osThreadFlagsSet`
: `osFlagsErrorUnknown` (Unspecified error) and `osFlagsErrorResource` (Thread specified by parameter thread_id is not active to receive flags) are not supported.

`osThreadFlagsClear`
: `osFlagsErrorResource` (Running thread is not active to receive flags) is not supported.

`osDelayUntil`
: `osParameter` (the time cannot be handled) is not supported.

Chapter 5

Build and Configuration Systems

5.1 Build System (CMake)

CMake is used to build your application together with the Zephyr kernel. A CMake build is done in two stages. The first stage is called **configuration**. During configuration, the CMakeLists.txt build scripts are executed. After configuration is finished, CMake has an internal model of the Zephyr build, and can generate build scripts that are native to the host platform.

CMake supports generating scripts for several build systems, but only Ninja and Make are tested and supported by Zephyr. After configuration, you begin the **build** stage by executing the generated build scripts. These build scripts can recompile the application without involving CMake following most code changes. However, after certain changes, the configuration step must be executed again before building. The build scripts can detect some of these situations and reconfigure automatically, but there are cases when this must be done manually.

Zephyr uses CMake's concept of a 'target' to organize the build. A target can be an executable, a library, or a generated file. For application developers, the library target is the most important to understand. All source code that goes into a Zephyr build does so by being included in a library target, even application code.

Library targets have source code, that is added through CMakeLists.txt build scripts like this:

```
target_sources(app PRIVATE src/main.c)
```

In the above `CMakeLists.txt`, an existing library target named `app` is configured to include the source file `src/main.c`. The `PRIVATE` keyword indicates that we are modifying the internals of how the library is being built. Using the keyword `PUBLIC` would modify how other libraries that link with app are built. In this case, using `PUBLIC` would cause libraries that link with `app` to also include the source file `src/main.c`, behavior that we surely do not want. The `PUBLIC` keyword could however be useful when modifying the include paths of a target library.

When introducing build system code using CMake or adding new CMake files, please follow the style guidelines outlined *here*.

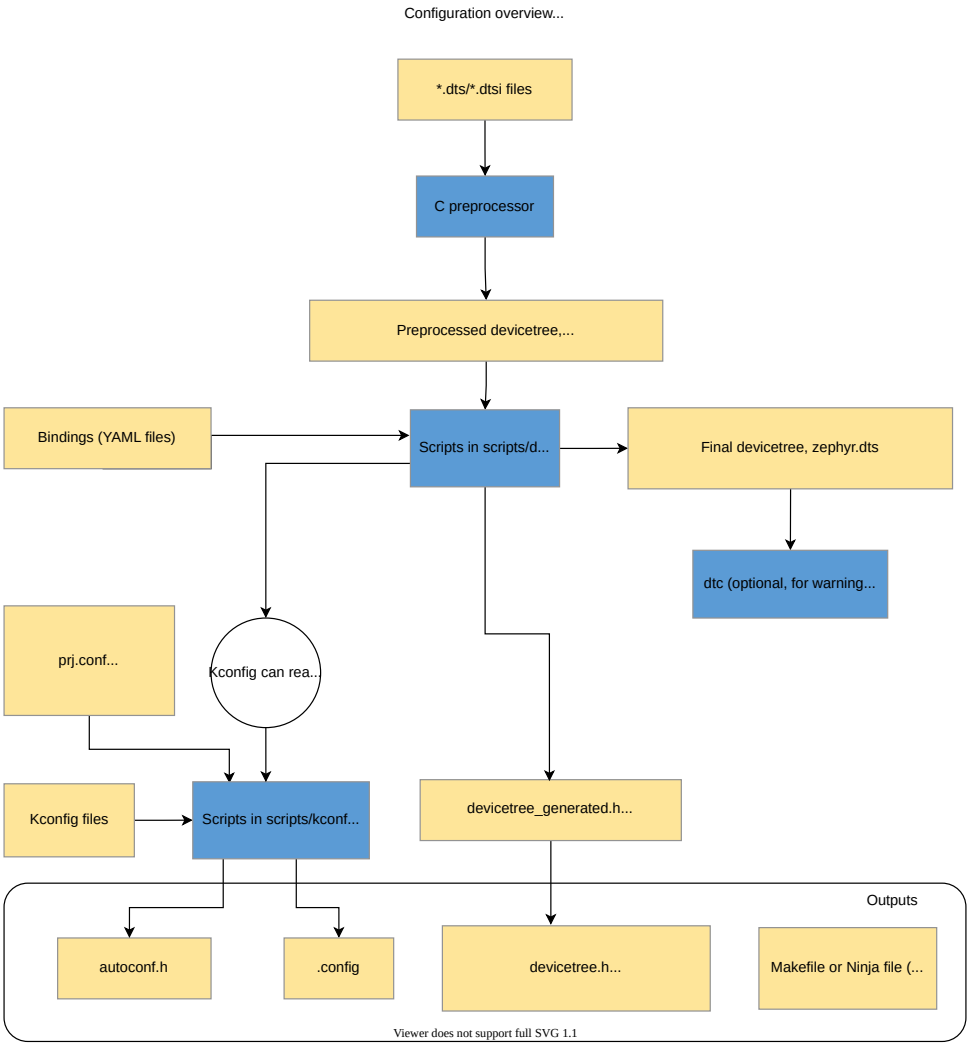
The *CMake Reference* section provides a reference for all CMake commands, variables, and modules used by Zephyr and available to application developers.

5.1.1 Build and Configuration Phases

The Zephyr build process can be divided into two main phases: a configuration phase (driven by CMake) and a build phase (driven by Make or Ninja).

Configuration Phase

The configuration phase begins when the user invokes *CMake* to generate a build system, specifying a source application directory and a board target.



CMake begins by processing the `CMakeLists.txt` file in the application directory, which refers to the `CMakeLists.txt` file in the Zephyr top-level directory, which in turn refers to `CMakeLists.txt` files throughout the build tree (directly and indirectly). Its primary output is a set of Makefiles or Ninja files to drive the build process, but the CMake scripts also do some processing of their own, which is explained here.

Note that paths beginning with `build/` below refer to the build directory you create when running CMake.

Devicetree

`*.dts` (*devicetree source*) and `*.dtsi` (*devicetree source include*) files are collected from the target's architecture, SoC, board, and application directories.

`*.dtsi` files are included by `*.dts` files via the C preprocessor (often abbreviated *cpp*, which should not be confused with C++). The C preprocessor is also used to merge in any devicetree `*.overlay` files, and to expand macros in `*.dts`, `*.dtsi`, and `*.overlay` files. The preprocessor output is placed in `build/zephyr/zephyr.dts.pre`.

The preprocessed devicetree sources are parsed by gen_defines.py to generate a `build/zephyr/include/generated/zephyr/devicetree_generated.h` header with preprocessor macros.

Source code should access preprocessor macros generated from devicetree by including the devicetree.h header, which includes `devicetree_generated.h`.

`gen_defines.py` also writes the final devicetree to `build/zephyr/zephyr.dts` in the build directory. This file's contents may be useful for debugging.

If the devicetree compiler `dtc` is installed, it is run on `build/zephyr/zephyr.dts` to catch any extra warnings and errors generated by this tool. The output from `dtc` is unused otherwise, and this step is skipped if `dtc` is not installed.

The above is just a brief overview. For more information on devicetree, see *Devicetree Guide*.

Kconfig

`Kconfig` files define available configuration options for the target architecture, SoC, board, and application, as well as dependencies between options.

Kconfig configurations are stored in *configuration files*. The initial configuration is generated by merging configuration fragments from the board and application (e.g. `prj.conf`).

The output from Kconfig is an `autoconf.h` header with preprocessor assignments, and a `.config` file that acts both as a saved configuration and as configuration output (used by CMake). The definitions in `autoconf.h` are automatically exposed at compile time, so there is no need to include this header.

Information from devicetree is available to Kconfig, through the functions defined in kconfigfunctions.py.

See *the Kconfig section of the manual* for more information.

Build Phase

The build phase begins when the user invokes `make` or `ninja`. Its ultimate output is a complete Zephyr application in a format suitable for loading/flashing on the desired target board (`zephyr.elf`, `zephyr.hex`, etc.) The build phase can be broken down, conceptually, into four stages: the pre-build, first-pass binary, final binary, and post-processing.

Pre-build Pre-build occurs before any source files are compiled, because during this phase header files used by the source files are generated.

Build-time constants generation

Some constants must be derived from C struct layouts at build time rather than hardcoded. The CMake function `zephyr_constants_library()` automates this: it compiles a C source file containing `GEN_ABSOLUTE_SYM()` declarations into an OBJECT library, then runs `gen_offset_header.py` to extract symbols ending in `_OFFSET` or `_SIZEOF` and emit them as `#define` lines in a generated header under `include/generated/zephyr/`.

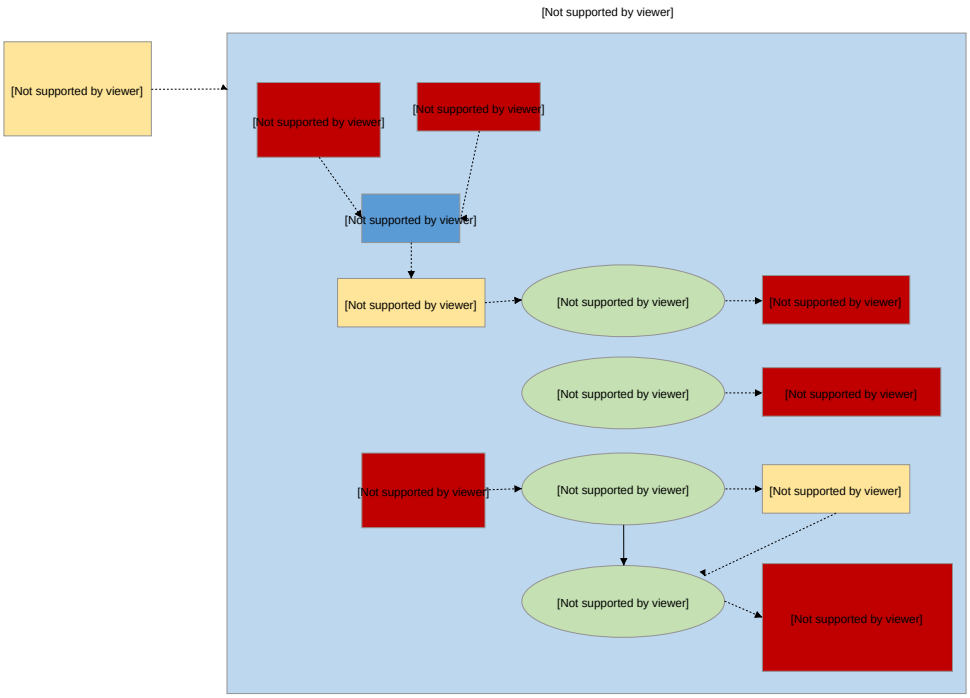
This mechanism is used in two places:

- **offsets.h**: Architecture-specific offsets for assembly code that needs access to high-level data structures when the definitions of those structures are not immediately accessible. The source file is `arch/<arch>/core/offsets/offsets.c`.
- **heap_constants.h**: Heap sizing constants (`Z_HEAP_MIN_SIZE`, `Z_HEAP_MIN_SIZE_FOR`) computed from actual heap struct layouts, replacing previously hardcoded values in `kernel.h`. The source file is `lib/heap/heap_constants.c`.

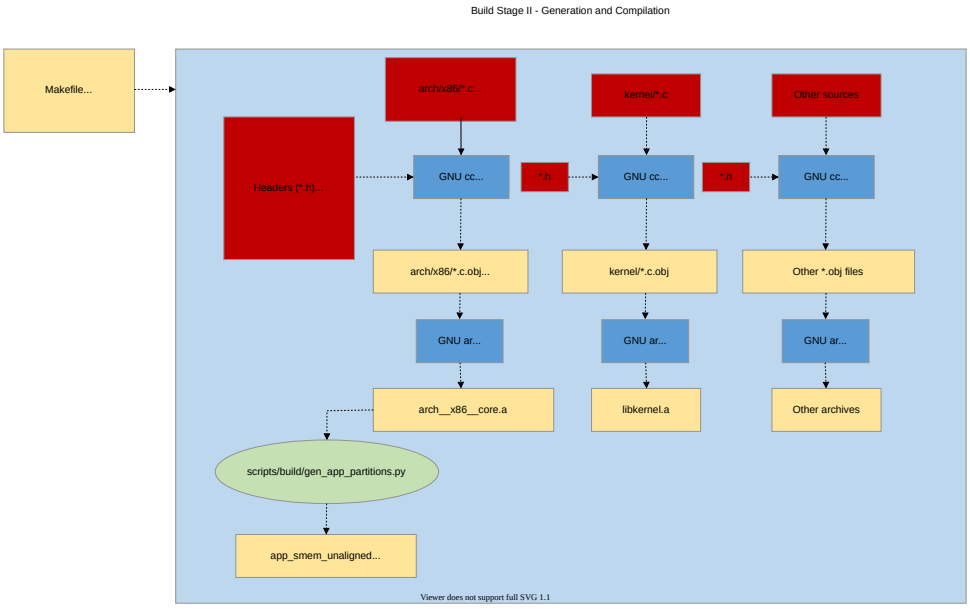
To add new build-time constants, create a C source file that includes `<gen_offset.h>` and uses `GEN_ABSOLUTE_SYM()` to declare each constant, then call `zephyr_constants_library()` from your `CMakeLists.txt`. If the new library's source includes a header generated by another constants library, use the `DEPENDS` parameter to declare that ordering (e.g., `DEPENDS heap_constants`). The resulting OBJECT library can also be linked into the final binary (as offsets.h does via `$<TARGET_OBJECTS:offsets>` in linker scripts).

System call boilerplate

The *gen_syscalls.py* and *parse_syscalls.py* scripts work together to bind potential system call functions with their implementations.



Intermediate binaries Compilation proper begins with the first intermediate binary. Source files (C and assembly) are collected from various subsystems (which ones is decided during the configuration phase), and compiled into archives (with reference to header files in the tree, as well as those generated during the configuration phase and the pre-build stage(s)).



The exact number of intermediate binaries is decided during the configuration phase.

If memory protection is enabled, then:

Partition grouping

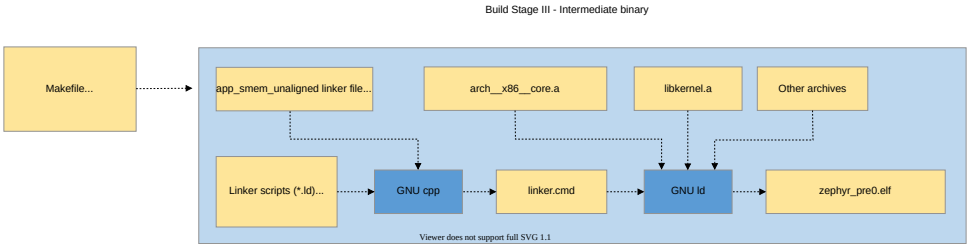
The *gen_app_partitions.py* script scans all the generated archives and outputs linker scripts

to ensure that application partitions are properly grouped and aligned for the target's memory protection hardware.

Then *cpp* is used to combine linker script fragments from the target's architecture/SoC, the kernel tree, optionally the partition output if memory protection is enabled, and any other fragments selected during the configuration process, into a *linker.cmd* file. The compiled archives are then linked with *ld* as specified in the *linker.cmd*.

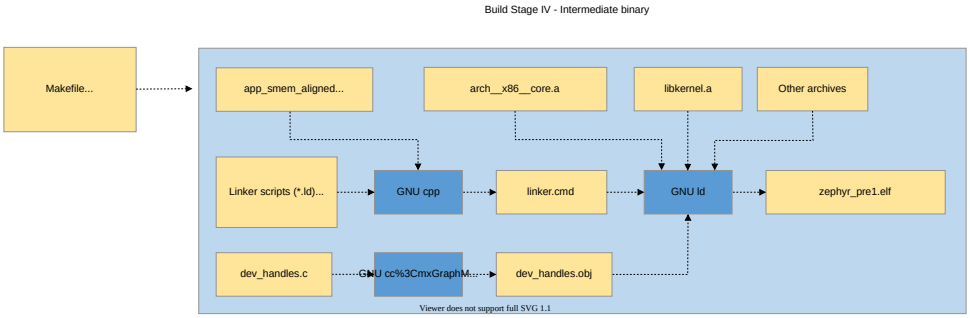
Unfixed size binary

The unfixed size intermediate binary is produced when *User Mode* is enabled or *Devicetree* is in use. It produces a binary where sizes are not fixed and thus it may be used by post-process steps that will impact the size of the final binary.



Fixed size binary

The fixed size intermediate binary is produced when *User Mode* is enabled or when generated IRQ tables are used, `CONFIG_GEN_ISR_TABLES` It produces a binary where sizes are fixed and thus the size must not change between the intermediate binary and the final binary.



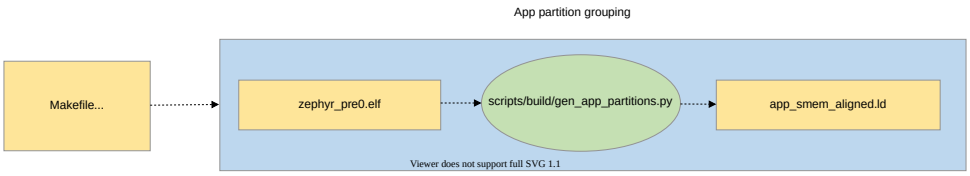
Intermediate binaries post-processing The binaries from the previous stage are incomplete, with empty and/or placeholder sections that must be filled in by, essentially, reflection.

To complete the build procedure the following scripts are executed on the intermediate binaries to produce the missing pieces needed for the final binary.

When *User Mode* is enabled:

Partition alignment

The *gen_app_partitions.py* script scans the unfixed size binary and generates an app shared memory aligned linker script snippet where the partitions are sorted in descending order.

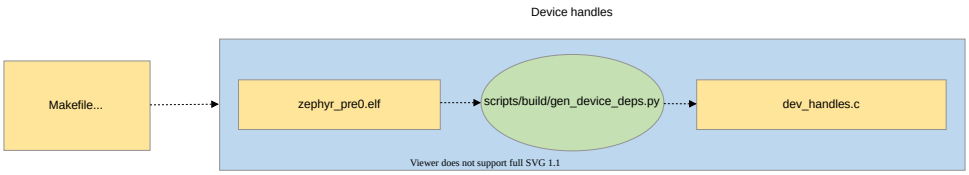


When *Devicetree* is used:

Device dependencies

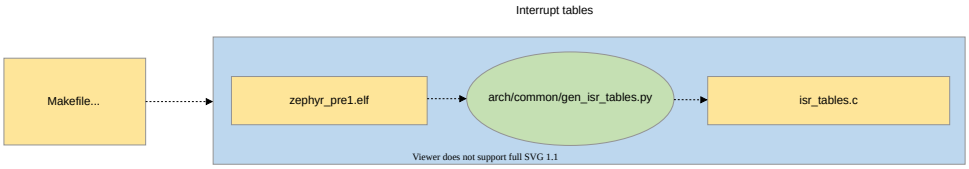
The *gen_device_deps.py* script scans the unfixed size binary to determine relationships be-

tween devices that were recorded from devicetree data, and replaces the encoded relationships with values that are optimized to locate the devices actually present in the application.



When `CONFIG_GEN_ISR_TABLES` is enabled:

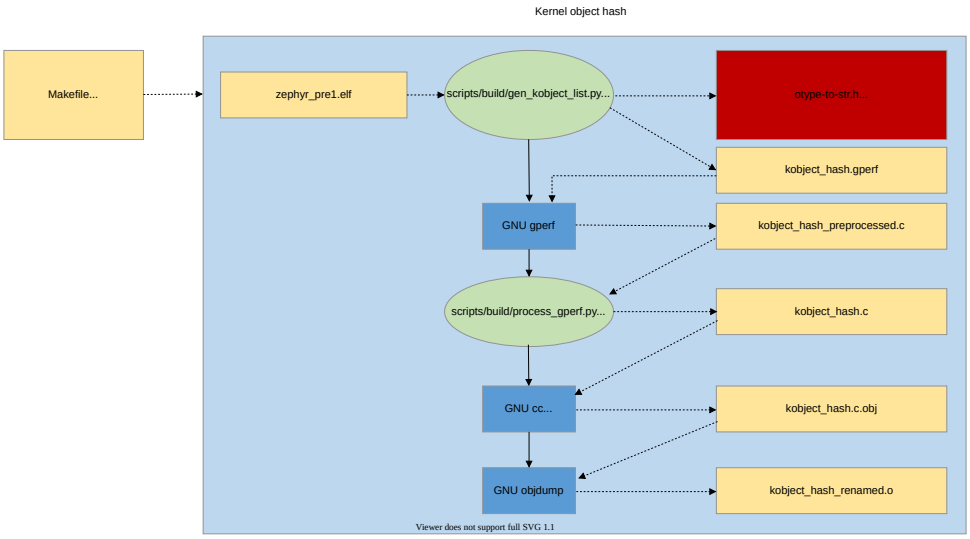
The *gen_isr_tables.py* script scans the fixed size binary and creates an isr_tables.c source file with a hardware vector table and/or software IRQ table.



When *User Mode* is enabled:

Kernel object hashing

The *gen_kobject_list.py* scans the *ELF DWARF* debug data to find the address of the all kernel objects. This list is passed to *gperf*, which generates a perfect hash function and table of those addresses, then that output is optimized by *process_gperf.py*, using known properties of our special case.

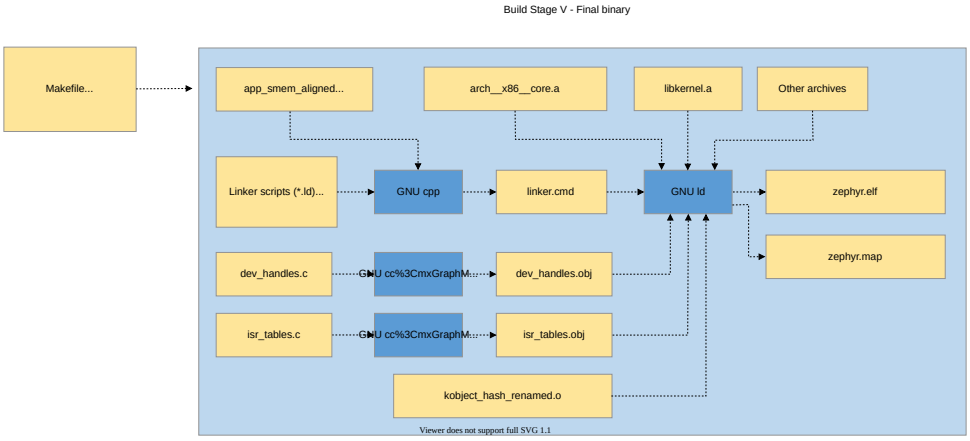


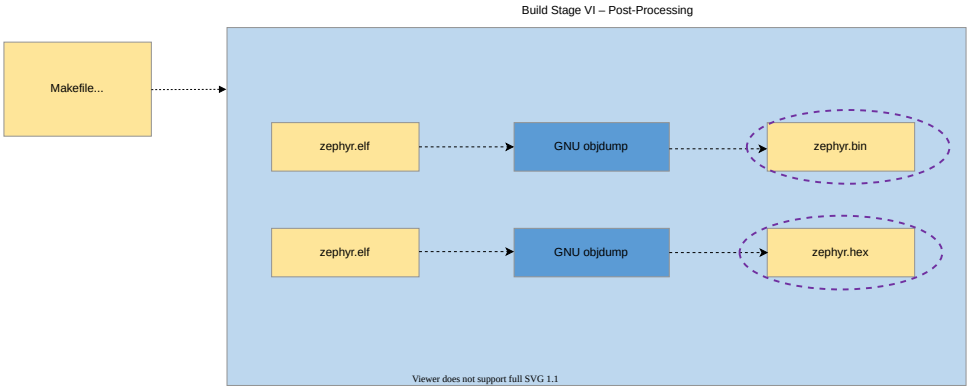
When no intermediate binary post-processing is required then the first intermediate binary will be directly used as the final binary.

Final binary The binary from the previous stage is incomplete, with empty and/or placeholder sections that must be filled in by, essentially, reflection.

The link from the previous stage is repeated, this time with the missing pieces populated.

Post processing Finally, if necessary, the completed kernel is converted from *ELF* to the format expected by the loader and/or flash tool required by the target. This is accomplished in a straightforward manner with *objdump*.





5.1.2 Supporting Scripts and Tools

The following is a detailed description of the scripts used during the build process.

scripts/build/gen_syscalls.py

Script to generate system call invocation macros

This script parses the system call metadata JSON file emitted by parse_syscalls.py to create several files:

- A file containing weak aliases of any potentially unimplemented system calls, as well as the system call dispatch table, which maps system call type IDs to their handler functions.
- A header file defining the system call type IDs, as well as function prototypes for all system call handler functions.
- A directory containing header files. Each header corresponds to a header that was identified as containing system call declarations. These generated headers contain the inline invocation functions for each system call in that header.

scripts/build/gen_device_deps.py

Translate generic handles into ones optimized for the application.

Immutable device data includes information about dependencies, e.g. that a particular sensor is controlled through a specific I2C bus and that it signals event on a pin on a specific GPIO controller. This information is encoded in the first-pass binary using identifiers derived from the devicetree. This script extracts those identifiers and replaces them with ones optimized for use with the devices actually present.

For example the sensor might have a first-pass handle defined by its devicetree ordinal 52, with the I2C driver having ordinal 24 and the GPIO controller ordinal 14. The runtime ordinal is

the index of the corresponding device in the static devicetree array, which might be 6, 5, and 3, respectively.

The output is a C source file that provides alternative definitions for the array contents referenced from the immutable device objects. In the final link these definitions supersede the ones in the driver-specific object file.

scripts/build/gen_kobject_list.py

Script to generate gperf tables of kernel object metadata

User mode threads making system calls reference kernel objects by memory address, as the kernel/driver APIs in Zephyr are the same for both user and supervisor contexts. It is necessary for the kernel to be able to validate accesses to kernel objects to make the following assertions:

- That the memory address points to a kernel object
- The kernel object is of the expected type for the API being invoked
- The kernel object is of the expected initialization state
- The calling thread has sufficient permissions on the object

For more details see the *Kernel Objects* section in the documentation.

The zephyr build generates an intermediate ELF binary, zephyr_prebuilt.elf, which this script scans looking for kernel objects by examining the DWARF debug information to look for instances of data structures that are considered kernel objects. For device drivers, the API struct pointer populated at build time is also examined to disambiguate between various device driver instances since they are all 'struct device'.

This script can generate five different output files:

- A gperf script to generate the hash table mapping kernel object memory addresses to kernel object metadata, used to track permissions, object type, initialization state, and any object-specific data.
- A header file containing generated macros for validating driver instances inside the system call handlers for the driver subsystem APIs.
- A code fragment included by kernel.h with one enum constant for each kernel object type and each driver instance.
- The inner cases of a switch/case C statement, included by kernel/userspace.c, mapping the kernel object types and driver instances to their human-readable representation in the otype_to_str() function.
- The inner cases of a switch/case C statement, included by kernel/userspace.c, mapping kernel object types to their sizes. This is used for allocating instances of them at runtime (CONFIG_DYNAMIC_OBJECTS) in the obj_size_get() function.

scripts/build/gen_offset_header.py

This script scans a specified object file and generates a header file that defined macros for the offsets of various found structure members (particularly symbols ending with `_OFFSET` or `_SIZEOF`), primarily intended for use in assembly code.

scripts/build/parse_syscalls.py

Script to scan Zephyr include directories and emit system call and subsystem metadata

System calls require a great deal of boilerplate code in order to implement completely. This script is the first step in the build system's process of auto-generating this code by doing a text scan of directories containing C or header files, and building up a database of system calls and

their function call prototypes. This information is emitted to a generated JSON file for further processing.

This script also scans for struct definitions such as __subsystem and __net_socket, emitting a JSON dictionary mapping tags to all the struct declarations found that were tagged with them.

If the output JSON file already exists, its contents are checked against what information this script would have outputted; if the result is that the file would be unchanged, it is not modified to prevent unnecessary incremental builds.

arch/x86/gen_idt.py

Generate Interrupt Descriptor Table for x86 CPUs.

This script generates the interrupt descriptor table (IDT) for x86. Please consult the IA Architecture SW Developer Manual, volume 3, for more details on this data structure.

This script accepts as input the zephyr_prebuilt.elf binary, which is a link of the Zephyr kernel without various build-time generated data structures (such as the IDT) inserted into it. This kernel image has been properly padded such that inserting these data structures will not disturb the memory addresses of other symbols. From the kernel binary we read a special section "intList" which contains the desired interrupt routing configuration for the kernel, populated by instances of the IRQ_CONNECT() macro.

This script outputs three binary tables:

1. The interrupt descriptor table itself.
2. A bitfield indicating which vectors in the IDT are free for installation of dynamic interrupts at runtime.
3. An array which maps configured IRQ lines to their associated vector entries in the IDT, used to program the APIC at runtime.

arch/x86/gen_gdt.py

Generate a Global Descriptor Table (GDT) for x86 CPUs.

For additional detail on GDT and x86 memory management, please consult the IA Architecture SW Developer Manual, vol. 3.

This script accepts as input the zephyr_prebuilt.elf binary, which is a link of the Zephyr kernel without various build-time generated data structures (such as the GDT) inserted into it. This kernel image has been properly padded such that inserting these data structures will not disturb the memory addresses of other symbols.

The input kernel ELF binary is used to obtain the following information:

- Memory addresses of the Main and Double Fault TSS structures so GDT descriptors can be created for them
- Memory addresses of where the GDT lives in memory, so that this address can be populated in the GDT pseudo descriptor
- whether userspace or HW stack protection are enabled in Kconfig

The output is a GDT whose contents depend on the kernel configuration. With no memory protection features enabled, we generate flat 32-bit code and data segments. If hardware- based stack overflow protection or userspace is enabled, we additionally create descriptors for the main and double- fault IA tasks, needed for userspace privilege elevation and double-fault handling. If userspace is enabled, we also create flat code/data segments for ring 3 execution.

scripts/build/gen_relocate_app.py

This script will relocate .text, .rodata, .data and .bss sections from required files and places it in the required memory region. This memory region and file are given to this python script in the form of a file. A regular expression filter can be applied to select only the required sections from the file.

Example of content in such an input file would be:

```
SRAM2:COPY:/home/xyz/zephyr/samples/hello_world/src/main.c,.*foo|.*bar
SRAM1:COPY:/home/xyz/zephyr/samples/hello_world/src/main2.c,.*bar
FLASH2:NOCOPY:/home/xyz/zephyr/samples/hello_world/src/main3.c,
```

One can also specify the program header for a given memory region:

SRAM2\ :phdr0:COPY:/home/xyz/zephyr/samples/hello_world/src/main.c,

To invoke this script:

```
python3 gen_relocate_app.py -i input_file -o generated_linker -c generated_code
```

Configuration that needs to be sent to the python script.

- If the memory is like SRAM1/SRAM2/CCD/AON then place full object in the sections
- If the memory type is appended with _DATA / _TEXT/ _RODATA/ _BSS only the selected memory is placed in the required memory region. Others are ignored.
- COPY/NOCOPY defines whether the script should generate the relocation code in code_relocation.c or not
- NOKEEP will suppress the default behavior of marking every relocated symbol with KEEP() in the generated linker script.

Multiple regions can be appended together like SRAM2_DATA_BSS this will place data and bss inside SRAM2.

scripts/build/process_gperf.py

gperf C file post-processor

We use gperf to build up a perfect hashtable of pointer values. The way gperf does this is to create a table 'wordlist' indexed by a string representation of a pointer address, and then doing memcmp() on a string passed in for comparison

We are exclusively working with 4-byte pointer values. This script adjusts the generated code so that we work with pointers directly and not strings. This saves a considerable amount of space.

scripts/build/gen_app_partitions.py

Script to generate a linker script organizing application memory partitions

Applications may declare build-time memory domain partitions with K_APPMEM_PARTITION_DEFINE, and assign globals to them using K_APP_DMEM or K_APP_BMEM macros. For each of these partitions, we need to route all their data into appropriately-sized memory areas which meet the size/alignment constraints of the memory protection hardware.

This linker script is created very early in the build process, before the build attempts to link the kernel binary, as the linker script this tool generates is a necessary pre-condition for kernel linking. We extract the set of memory partitions to generate by looking for variables which have been assigned to input sections that follow a defined naming convention. We also allow entire libraries to be pulled in to assign their globals to a particular memory partition via command line directives.

This script takes as inputs:

- The base directory to look for compiled objects
- key/value pairs mapping static library files to what partitions their globals should end up in.

The output is either a linker script fragment or linker-script generator fragments containing the definition of the app shared memory section, which is further divided, for each partition found, into data and BSS for each partition.

scripts/build/check_init_priorities.py

Checks the initialization priorities

This script parses a Zephyr executable file, creates a list of known devices and their effective initialization priorities and compares that with the device dependencies inferred from the devicetree hierarchy.

This can be used to detect devices that are initialized in the incorrect order, but also devices that are initialized at the same priority but depends on each other, which can potentially break if the linking order is changed.

Optionally, it can also produce a human readable list of the initialization calls for the various init levels.

5.2 CMake Reference

5.2.1 Core Build System & Hardware Modules

arch

Configure arch settings based on Kconfig settings and arch root.

This CMake module will set the following variables in the build system based on board directory and arch root.

If no implementation is available for the current arch, an error will be raised.

Variables The following variables will be defined when this CMake module completes:

- *ARCH*
- *ARCH_DIR*
- *ARCH_ROOT*

boards

Validate board and setup boards target.

This CMake module will validate the *BOARD* argument as well as splitting it into `<BOARD>` and `<BOARD_REVISION>`.

If a board implementation is not found for the specified board an error will be raised and list of valid boards will be printed.

If user provided board is a board alias, the board will be adjusted to real board name.

If board name is deprecated, then board will be adjusted to new board name and a deprecation warning will be printed to the user.

Required variables

- *BOARD*

Optional variables

- *BOARD_ROOT*
- *ARCH_ROOT*

Variables The following variables will be defined when this CMake module completes:

- *BOARD*
- *BOARD_REVISION*
- *BOARD_QUALIFIERS*
- *NORMALIZED_BOARD_QUALIFIERS*
- *NORMALIZED_BOARD_TARGET*
- *BOARD_DIR*
- *ARCH_DIR*
- *BOARD_ROOT*

Targets The following targets will be defined when this CMake module completes:

- `boards`

  When invoked, a list of valid boards will be printed.

Optional environment variables *ZEPHYR_BOARD_ALIASES*

Environment setting pointing to a CMake file containing board aliases.

Example usage

```
# BOARD is normally given on the command line, for example:
#   west build -b nrf52840dk/nrf52840
include(boards)

message(STATUS "Building for ${BOARD} (${NORMALIZED_BOARD_TARGET})")
message(STATUS "Board files are in ${BOARD_DIR}")
```

configuration_files

Locate the Kconfig and DT config files that are to be used. Also, locate the appropriate application config directory.

Variables The following variables will be defined when this CMake module completes:

- *CONF_FILE*
- *EXTRA_CONF_FILE*
- *DTC_OVERLAY_FILE*
- *EXTRA_DTC_OVERLAY_FILE*
- *DTS_EXTRA_CPPFLAGS*
- *APPLICATION_CONFIG_DIR*

If any of the above variables are already set when this CMake module is loaded, then no changes to the variable will happen.

Variables set by this module and not mentioned above are considered internal use only and may be removed, renamed, or re-purposed without prior notice.

soc

Configure SoC settings based on Kconfig settings.

This CMake module will set the following variables in the build system based on Kconfig settings for the selected SoC.

Variables The following variables will be defined when this CMake module completes:

- `SOC_FULL_DIR`: Full directory of where the SoC files are located
- `SOC_DIRECTORIES`: List of directories where SoC files which include this SoC are located
- `SOC_TOOLCHAIN_NAME`: Optional toolchain name of the SoC

Variables set by this module and not mentioned above are considered internal use only and may be removed, renamed, or re-purposed without prior notice.

kernel

Main Zephyr Kernel CMake module.

This module is responsible for creating Zephyr libraries and the Zephyr executable. It creates a CMake project named *Zephyr-Kernel* and defines properties used while configuring libraries to be built.

Variables After loading this module, the following global variables are defined:

`ZEPHYR_LIBS`
: Global list of all Zephyr CMake libraries that should be linked in. *`zephyr_library()`* appends libraries to this list.

`ZEPHYR_INTERFACE_LIBS`
: Global list of all Zephyr interface libraries that should be linked in. *`zephyr_interface_library_named()`* appends libraries to this list.

`GENERATED_APP_SOURCE_FILES`
: Source files that are generated after Zephyr has been linked once. May include dev_handles.c etc.

`GENERATED_KERNEL_OBJECT_FILES`
: Object files that are generated after symbol addresses are fixed. May include mmu tables, etc.

`GENERATED_KERNEL_SOURCE_FILES`
: Source files that are generated after symbol addresses are fixed. May include isr_tables.c etc.

Targets

`code_data_relocation_target`
: Custom target for code data relocation.

`app`
: CMake library containing all the application code. Modified by the entry point ${APPLICATION_SOURCE_DIR}/CMakeLists.txt.

Usage This module is not intended for direct loading, but should be loaded through `find_package(Zephyr)`. It won't load any Zephyr CMake modules by itself.

basic_settings

Setup basic settings for a Zephyr project.

Basic settings are:

- Sysbuild defined configuration settings

Details for sysbuild settings:

Sysbuild is a higher level build system used by Zephyr. Sysbuild allows users to build multiple samples for a given system.

For this to work, sysbuild manages other Zephyr CMake build systems by setting dedicated build variables. This CMake module loads the sysbuild cache variables as target properties on a `sysbuild_cache` target.

This ensures that quotes and lists are correctly preserved.

extensions

Zephyr's CMake extension commands.

This module defines the commands that Zephyr applications, Zephyr modules, and the build system itself use to describe what they build.

Many commands come in an `_ifdef` and an `_ifndef` flavour, taking a Kconfig option as their first argument, so that build rules can be made conditional without wrapping them in an `if()` block.

This module is loaded as part of `find_package(Zephyr)`, which means that every command documented below is available in the `CMakeLists.txt` of any Zephyr application or module.

- *Zephyr-aware extensions*
  - *zephyr_**
  - *zephyr_library_**
    - *zephyr_interface_library_**
  - *generate_inc_**
  - *board_**
    - *Zephyr board runners*
    - *Zephyr board revision*
  - *Misc.*
- *Kconfig-aware extensions*
  - *Misc*
- *CMake-generic extensions*
  - *Debugging CMake*
  - *File system management*
  - *Others*
- *Devicetree extensions*
  - *dt_**
  - **_if_dt_node*
  - *zephyr_dt_**

- *Zephyr linker functions*
  - *zephyr_linker**
- *Function helper macros*
- *Linkable loadable extensions (llext)*
  - *Configuration functions*
  - *Build control functions*

Zephyr-aware extensions

zephyr_* The following methods are for modifying the CMake library called zephyr. zephyr is a catch-all CMake library for source files that can be built purely with the include paths, defines, and other compiler flags that all zephyr source files use.

Example usage:

```
zephyr_sources(
  random_esp32.c
  utils.c
)
```

Is short for:

```
target_sources(zephyr PRIVATE
  ${CMAKE_CURRENT_SOURCE_DIR}/random_esp32.c
  ${CMAKE_CURRENT_SOURCE_DIR}/utils.c
)
```

As a very high-level introduction, here are two call graphs that are purposely minimalistic and incomplete.

```
zephyr_library_cc_option()
         |
         v
zephyr_library_compile_options()  -->  target_compile_options()
```

```
zephyr_cc_option()           --->  target_cc_option()
                                        |
                                        v
zephyr_cc_option_fallback()  --->  target_cc_option_fallback()
                                        |
                                        v
zephyr_compile_options()     --->  target_compile_options()
```

zephyr_sources(<sources>...)

Add sources to the zephyr library.

See target_sources for details.

zephyr_include_directories(<dirs>...)

Add include directories to the zephyr library.

See target_include_directories for details.

`zephyr_system_include_directories(<dirs>...)`

Add system include directories to the `zephyr` library.

See `target_include_directories` for details.

`zephyr_compile_definitions(<defs>...)`

Add compile definitions to the `zephyr` library.

See `target_compile_definitions` for details.

`zephyr_compile_options(<options>...)`

Add compile options to the `zephyr` library.

See `target_compile_options` for details.

`zephyr_link_libraries(<item>...)`

Add link libraries to the `zephyr` library.

See `target_link_libraries` for details.

`zephyr_libc_link_libraries(<item> ...)`

Add link libraries to the `zephyr` library's *`LIBC_LINK_LIBRARIES`* property.

This function allows subsystems to define libraries which get added to the link command after all other libraries and modules. It's useful when using a toolchain library, like libc or libgcc, as those can get added when processing the 'lib' directory, before any module libraries and hence might not get used to resolve symbols from modules.

`zephyr_cc_option(<option>...)`

Add compiler options to the `zephyr` library, if supported by the compiler.

This function checks if the compiler supports each option. If supported, it adds the option to the 'zephyr' library.

`zephyr_cc_option_fallback(<option1> <option2>)`

Add a compiler option to the `zephyr` library, falling back to a second option if the first is not supported.

This function checks if the compiler supports `<option1>`. If so, it is added. Otherwise, it checks `<option2>` and adds it if supported.

`zephyr_ld_options(<options>...)`

Add linker options to the `zephyr` library.

See `target_ld_options()` for details.

`zephyr_get_include_directories_for_lang_as_string(<lang> <var> [STRIP_PREFIX])`

Get include directories for a specific language as a string.

Writes the include directories for language `<lang>` to variable `<var>` as a string.

`<lang>` can be one of `C`, `CXX`, or `ASM`.

`STRIP_PREFIX` can be specified to omit the prefix from the result.

`zephyr_get_system_include_directories_for_lang_as_string(<lang> <var> [STRIP_PREFIX])`

Get system include directories for a specific language as a string.

Writes the system include directories for language `<lang>` to variable `<var>` as a string.

`<lang>` can be one of `C`, `CXX`, or `ASM`.

`STRIP_PREFIX` can be specified to omit the prefix from the result.

zephyr_get_compile_definitions_for_lang_as_string(<lang> <var> [STRIP_PREFIX])

Get compile definitions for a specific language as a string.

Writes the compile definitions for language <lang> to variable <var> as a string.

<lang> can be one of C, CXX, or ASM.

STRIP_PREFIX can be specified to omit the prefix from the result.

zephyr_get_compile_options_for_lang_as_string(<lang> <var> [STRIP_PREFIX])

Get compile options for a specific language as a string.

Writes the compile options for language <lang> to variable <var> as a string.

<lang> can be one of C, CXX, or ASM.

STRIP_PREFIX can be specified to omit the prefix from the result.

zephyr_get_include_directories_for_lang(<lang> <var> [STRIP_PREFIX])

Get include directories for a specific language as a list.

Writes the include directories for language <lang> to variable <var> as a list.

<lang> can be one of C, CXX, or ASM.

STRIP_PREFIX can be specified to omit the prefix from the result.

zephyr_get_system_include_directories_for_lang(<lang> <var> [STRIP_PREFIX])

Get system include directories for a specific language as a list.

Writes the system include directories for language <lang> to variable <var> as a list.

<lang> can be one of C, CXX, or ASM.

STRIP_PREFIX can be specified to omit the prefix from the result.

zephyr_get_compile_definitions_for_lang(<lang> <var> [STRIP_PREFIX])

Get compile definitions for a specific language as a list.

Writes the compile definitions for language <lang> to variable <var> as a list.

<lang> can be one of C, CXX, or ASM.

STRIP_PREFIX can be specified to omit the prefix from the result.

zephyr_get_compile_options_for_lang(<lang> <var> [STRIP_PREFIX])

Get compile options for a specific language as a list.

Writes the compile options for language <lang> to variable <var> as a list.

<lang> can be one of C, CXX, or ASM.

STRIP_PREFIX can be specified to omit the prefix from the result.

zephyr_library_* Zephyr libraries use CMake's library concept and a set of assumptions about how zephyr code is organized to cut down on boilerplate code.

A Zephyr library can be constructed by the function *zephyr_library()* or *zephyr_library_named()*. The constructors create a CMake library with a name accessible through the variable *ZEPHYR_CURRENT_LIBRARY*.

The variable *ZEPHYR_CURRENT_LIBRARY* should seldom be needed since the zephyr libraries have methods that modify the libraries. These methods have the signature: zephyr_library_*<target-function>*.

The methods are wrappers around the CMake target_* functions. See manual:cmake-commands(7) for documentation on the underlying target_* functions.

The methods modify the CMake target_* API to reduce boilerplate; PRIVATE is assumed.

The target is assumed to be *ZEPHYR_CURRENT_LIBRARY*

When a flag that is given through the `zephyr_*` API conflicts with the `zephyr_library_*` API then precedence will be given to the `zephyr_library_*` API. In other words, local configuration overrides global configuration.

zephyr_library()

Create a Zephyr library with a directory-inferred name. This sets the *ZEPHYR_CURRENT_LIBRARY* variable to the inferred name.

zephyr_library_named(<name>)

Create a Zephyr library with an explicitly given name. This sets the *ZEPHYR_CURRENT_LIBRARY* variable to the given name.

zephyr_library_amend([<dir>])

Provides amend functionality to a Zephyr library for out-of-tree usage.

When called from a Zephyr module, the corresponding zephyr library defined within Zephyr will be looked up.

`<dir>`: Use `<dir>` as out-of-tree base directory from where the Zephyr library name shall be generated. `<dir>` can be used in cases where the structure for the library is not placed directly at the ZEPHYR_MODULE's root directory or for cases where the module integration file is located in a 'MODULE_EXT_ROOT'.

Note, in order to ensure correct library when amending, the folder structure in the Zephyr module or `<dir>` base directory must resemble the structure used in Zephyr, as example:

Example: to amend the zephyr library created in `ZEPHYR_BASE/drivers/entropy/CMakeLists.txt`, add a `ZEPHYR_MODULE/drivers/entropy/CMakeLists.txt` file with the following content:

```
zephyr_library_amend()
zephyr_library_sources(...)
```

It is also possible to use generator expression when amending to Zephyr libraries.

For example, in case it is required to expose the Zephyr library's folder as include path then the following is possible:

```
zephyr_library_amend()
zephyr_library_include_directories($<TARGET_PROPERTY:SOURCE_DIR>)
```

See the CMake documentation for more target properties or generator expressions.

zephyr_library_sources(<source> ...)

Add source files to the current Zephyr library.

zephyr_library_include_directories(<include_dir> ...)

Add include directories to the current Zephyr library.

zephyr_library_link_libraries(<item> ...)

Add link libraries to the current Zephyr library.

zephyr_library_compile_definitions(<item> ...)

Add compile definitions to the current Zephyr library.

zephyr_library_compile_options(<item> ...)

Add compile options to the current Zephyr library.

zephyr_library_cc_option(<option> ...)

Add compile options to the current Zephyr library if the C compiler supports them. Unsupported options are ignored.

`zephyr_library_add_dependencies(<item> ...)`

Add dependencies to the current Zephyr library.

See `add_dependencies()` for more information.

`zephyr_append_cmake_library(<library>)`

Add the existing CMake library 'library' to the global list of Zephyr CMake libraries. This is done automatically by the constructor but must be called explicitly on CMake libraries that do not use a zephyr library constructor.

`zephyr_library_import(<library_name> <library_path>)`

Add the imported library 'library_name', located at 'library_path' to the global list of Zephyr CMake libraries.

`zephyr_library_app_memory(<partition>)`

Place the current zephyr library in the application memory partition.

The partition argument is the name of the partition where the library shall be placed.

Note: Ensure the given partition has been defined using `K_APPMEM_PARTITION_DEFINE` in source code.

`zephyr_interface_library_*` A Zephyr interface library is a thin wrapper over a CMake `INTERFACE` library. The most important responsibility of this abstraction is to ensure that when a user KConfig-enables a library then the header files of this library will be accessible to the 'app' library.

This is done because when a user uses Kconfig to enable a library he expects to be able to include its header files and call its functions out-of-the box.

A Zephyr interface library should be used when there exists some build information (include directories, defines, compiler flags, etc.) that should be applied to a set of Zephyr libraries and 'app' might be one of these libraries.

Zephyr libraries must explicitly call *`zephyr_library_link_libraries(<interface_library>)`* to use this build information. 'app' is treated as a special case for usability reasons; a Kconfig option (CONFIG_APP_LINK_WITH_<interface_library>) should exist for each interface_library and will determine if 'app' links with the interface_library.

This API has a constructor like the zephyr_library API has, but it does not have wrappers over the other cmake target functions.

`zephyr_interface_library_named(<name>)`

Create a Zephyr interface library with a given name.

`generate_inc_*` These functions are useful if there is a need to generate a file that can be included into the application at build time. The file can also be compressed automatically when embedding it.

See tests/application_development/gen_inc_file for an example of usage.

`generate_inc_file(<source_file> <generated_file> ...)`

Generate a file that can be included into the application at build time.

`source_file`
: The source file to be converted to hex

`generated_file`
: The generated file

`...`
: Extra arguments are passed to file2hex.py

`generate_inc_file_for_gen_target(<target> <source_file> <generated_file> <gen_target> ...)`

Generate a file that can be included into the application at build time, and make a target depend on the target that generates it.

Use this when a target generating `generated_file` already exists. To have one created, use *`generate_inc_file_for_target()`* instead.

target
The target that depends on the generated file

source_file
The source file to be converted to hex

generated_file
The generated file

gen_target
The target that generates `generated_file`

...
Extra arguments are passed to file2hex.py

`generate_inc_file_for_target(<target> <source_file> <generated_file> ...)`

Generate a file that can be included into the application at build time, and make a target depend on it.

A target is created to run the conversion, and `target` is made to depend on it, so that `generated_file` exists before anything in `target` is built.

target
The target that depends on the generated file

source_file
The source file to be converted to hex

generated_file
The generated file

...
Extra arguments are passed to file2hex.py

`board_*` This section is for extensions related to Zephyr board handling.

Zephyr board runners Zephyr board runner extension functions control Zephyr's board runners from the build system. The Zephyr build system has targets for flashing and debugging supported boards. These are wrappers around a "runner" Python subpackage that is part of Zephyr's "west" tool.

This section provides glue between CMake and the Python code that manages the runners.

`board_set_runner(<type> <runner>)`

This function sets the runner for the board unconditionally. It's meant to be used from application's `CMakeLists.txt` files.

> **Note**
>
> Usually `board_set_xxx_ifnset()` is best in `board.cmake` files. This lets the user set the runner at cmake time, or in their own application's `CMakeLists.txt`.

Example usage:

```
board_set_runner(FLASH pyocd)
```

This would set the board's flash runner to "pyocd".

In general, "type" is FLASH, DEBUG, SIM or ROBOT and "runner" is the name of a runner.

board_set_runner_ifnset(<type> <runner>)

This macro is like *board_set_runner()*, but will only make a change if that runner is currently not set.

See also *board_set_flasher_ifnset()* and *board_set_debugger_ifnset()*.

board_set_flasher(<runner>)

A convenience macro for *board_set_runner(FLASH ${runner})*.

board_set_debugger(<runner>)

A convenience macro for *board_set_runner(DEBUG ${runner})*.

board_set_flasher_ifnset(<runner>)

A convenience macro for *board_set_runner_ifnset(FLASH ${runner})*.

board_set_debugger_ifnset(<runner>)

A convenience macro for *board_set_runner_ifnset(DEBUG ${runner})*.

board_set_robot_runner_ifnset(<runner>)

A convenience macro for *board_set_runner_ifnset(ROBOT ${runner})*.

board_set_sim_runner_ifnset(<runner>)

A convenience macro for *board_set_runner_ifnset(SIM ${runner})*.

board_runner_args(<runner> <args>...)

This function is intended for board.cmake files and application CMakeLists.txt files.

Usage from board.cmake files:

```
board_runner_args(runner "--some-arg=val1" "--another-arg=val2")
```

The build system will then ensure the command line used to create the runner contains:

```
--some-arg=val1 --another-arg=val2
```

Within application CMakeLists.txt files, ensure that all calls to *board_runner_args()* are part of a macro named `app_set_runner_args()`, like this, which is defined before calling `find_package(Zephyr)`:.

```
macro(app_set_runner_args)
  board_runner_args(runner "--some-app-setting=value")
endmacro()
```

The build system tests for the existence of the macro and will invoke it at the appropriate time if it is defined.

Any explicitly provided settings given by this function override defaults provided by the build system.

board_set_rimage_target(<target>)

Set the rimage target for the current board.

Zephyr board revision This section provides a function for revision checking.

```
board_check_revision(FORMAT <LETTER | NUMBER | MAJOR.MINOR.PATCH> [EXACT]␣
[DEFAULT_REVISION <revision>] [HIGHEST_REVISION <revision>]␣
[VALID_REVISIONS <revision> ...] )
```

Zephyr board extension function.

This function can be used in `boards/board/revision.cmake` to check a user requested revision against available board revisions.

The function will check the revision from `-DBOARD=board@revision` that is provided by the user according to the arguments. When `EXACT` is not specified, this function will set the Zephyr build system variable *ACTIVE_BOARD_REVISION* with the selected revision.

FORMAT <LETTER | NUMBER | MAJOR.MINOR.PATCH>
Specify the revision format.

LETTER
Revision format is a single letter from A - Z.

NUMBER
Revision format is a single integer number.

MAJOR.MINOR.PATCH
Revision format is three numbers, separated by `.`, `x.y.z`. Trailing zeroes may be omitted on the command line, which means that 1.0.0 == 1.0 == 1.

OPTIONAL
Revision specifier is optional.

If revision is not provided the base board will be used. If both `EXACT` and `OPTIONAL` are given, then specifying the revision is optional, but if it is given then the `EXACT` requirements apply. Mutually exclusive with `DEFAULT_REVISION`.

EXACT
Revision is required to be an exact match.

As example, available revisions are: 0.1.0 and 0.3.0, and user provides 0.2.0, then an error is reported when `EXACT` is given. If `EXACT` is not provided, then closest lower revision will be selected as the active revision, which in the example will be 0.1.0.

DEFAULT_REVISION
Provides a default revision to use when user has not selected a revision number.

If no default revision is provided then user will be printed with an error if no revision is given on the command line.

HIGHEST_REVISION
Allows to specify highest valid revision for a board.

This can be used to ensure that a newer board cannot be used with an older Zephyr. As example, if current board supports revisions 0.x.0-0.99.99 and 1.0.0-1.99.99, and it is expected that current board implementation will not work with board revision 2.0.0, then `HIGHEST_REVISION` can be set to 1.99.99, and user will be printed with an error if using *<board>@2.0.0* or higher. This field is not needed when `EXACT` is used.

VALID_REVISIONS
A list of valid revisions for this board.

If this argument is not provided, then each Kconfig fragment of the form `board_revision.conf` in the board folder will be used as a valid revision for the board.

Misc.

```
pow2round(<variable>)
```

Round number to next power of two.

Example usage:

```
set(test 2)
pow2round(test)
# test is still 2

set(test 5)
pow2round(test)
# test is now 8
```

zephyr_build_string(outvar [BOARD board] [SHORT short_outvar] [BOARD_QUALIFIERS qualifiers] [BOARD_REVISION revision] [MERGE] [REVERSE])

Create a build string based on `BOARD`, `BOARD_REVISION`, and `BOARD_QUALIFIER`. This is a common function to ensure that build strings are always created in a uniform way.

When `MERGE` is supplied a list of build strings will be returned with the full build string as first item in the list. The full order of build strings returned in the list will be:

- Normalized board target build string, this includes qualifiers and revision ;
- Build string with board variants removed in addition ;
- Build string with cpuset removed in addition ;
- Build string with soc removed in addition.

If `REVISION` is supplied or obtained as system wide setting a build string with the sanitized revision string will be added in addition to the non-revisioned entry for each entry.

outvar
Output variable where the build string will be returned.

BOARD board
Board name to use when creating the build string.

SHORT short_outvar
Output variable where the shortened build string will be returned.

BOARD_QUALIFIERS qualifiers
Board qualifiers to use.

BOARD_REVISION revision
Board revision to use.

MERGE
Return a list of build strings instead of a single build string.

REVERSE
Reverse the list before returning it.

Example usage:

```
zephyr_build_string(build_string BOARD alpha)
# Returns "alpha" in build_string

zephyr_build_string(build_string BOARD alpha
                                 BOARD_REVISION 1.0.0)
# Returns "alpha_1_0_0" in build_string

zephyr_build_string(build_string BOARD alpha
                                 BOARD_QUALIFIERS /soc/bar)
# Returns "alpha_soc_bar" in build_string

zephyr_build_string(build_string BOARD alpha
                                 BOARD_REVISION 1.0.0
                                 BOARD_QUALIFIERS /soc/bar
```

(continues on next page)

(continued from previous page)

```
                                  MERGE)
# Returns list "alpha_soc_bar_1_0_0;alpha_soc_bar" in build_string

zephyr_build_string(build_string SHORT short_build_string
                                  BOARD alpha
                                  BOARD_REVISION 1.0.0
                                  BOARD_QUALIFIERS /soc/bar
                                  MERGE)
# Returns list "alpha_soc_bar_1_0_0;alpha_soc_bar" in build_string
# Returns list "alpha_bar_1_0_0;alpha_bar" in short_build_string

zephyr_build_string(build_string BOARD_QUALIFIERS /soc/bar/foo)
# Returns "soc_bar_foo" in build_string
```

`zephyr_syscall_include_directories(<directories>...)`

Add one or more directories to the include list passed to the syscall generator.

`<directories>...`
One or more directories to add.

`zephyr_syscall_header(<headers>...)`

Add one or more header files to the list passed to the syscall generator.

`<headers>...`
One or more header files to add.

`zephyr_syscall_header_ifdef(feature_toggle <headers>...)`

Add one or more header files to the list passed to the syscall generator if“feature_toggle“ is true.

`feature_toggle`
The name of the boolean variable to check.

`<headers>...`
One or more header files to add.

`zephyr_blobs_verify(<MODULE module | FILES file [files...]> [REQUIRED])`

Verify blobs fetched using west. If the sha256 checksum isn't valid, a warning/fatal error message is printed (level depends on `REQUIRED` flag).

`MODULE module`
Verify all blobs in the given module.

`FILES file [files...]`
Verify the specified files.

`REQUIRED`
If specified, a fatal error is raised if the verification fails. Otherwise, a warning is printed.

Example usage:

```
# Verify all blobs in my_module and fail on error
zephyr_blobs_verify(MODULE my_module REQUIRED)

# Verify a single file and print on error
zephyr_blobs_verify(FILES img/file.bin)
```

Kconfig-aware extensions *Kconfig* is a configuration language developed for the Linux kernel. The below functions integrate CMake with Kconfig.

Misc

`import_kconfig(prefix kconfig_fragment [keys] [TARGET target])`

Parse a KConfig fragment (typically with extension .config) and introduce all the symbols that are prefixed with `prefix` into the CMake namespace. List all created variable names in the `keys` output variable if present.

`prefix`
: symbol prefix of settings in the Kconfig fragment.

`kconfig_fragment`
: absolute path to the config fragment file.

`keys`
: output variable which will be populated with variable names loaded from the kconfig fragment.

`TARGET`
: set all symbols on `target` instead of adding them to the CMake namespace.

CMake-generic extensions These functions extend the CMake API in a way that is not particular to Zephyr. Primarily, they work around limitations in the CMake language to allow cleaner build scripts.

`add_subdirectory_ifdef`

This function adds a subdirectory to the build if `feature_toggle` is enabled.

Debugging CMake

`print(arg)`

Print the value of a variable.

Example usage:

```
print(BOARD)
# will print: "BOARD: nrf52dk"
```

`assert(test comment)`

Assert that a condition is true.

This macro will cause a fatal error and print an error message if the first expression is false.

`test`
: The boolean expression to test.

`comment`
: The error message to print if the assertion fails.

Example usage:

```
assert(ZEPHYR_TOOLCHAIN_VARIANT "ZEPHYR_TOOLCHAIN_VARIANT not set.")
```

`assert_not(test comment)`

Assert that a condition is false.

This macro will cause a fatal error and print an error message if the first expression is true.

`test`
: The boolean expression to test.

`comment`
: The error message to print if the assertion fails.

Example usage:

```
assert_not(OBSOLETE_VAR "OBSOLETE_VAR has been removed; use NEW_VAR instead")
```

`assert_exists(var)`

Assert that a file or directory exists.

This macro will cause a fatal error if the file or directory pointed to by the variable does not exist.

`var`
The name of the variable containing the path to check.

Example usage:

```
assert_exists(CMAKE_READELF)
```

File system management

`generate_unique_target_name_from_filename(filename target_name)`

Generate a unique target name from a filename.

`filename`
The filename to generate a unique name from.

`target_name`
The name of the variable to store the generated target name in.

`zephyr_file(APPLICATION_ROOT <path> [BASE_DIR <base-dir>])`

`zephyr_file(CONF_FILES <paths> [DTS <list>] [KCONF <list>] [DEFCONF <list>] [BOARD <board> [BOARD_REVISION <revision>] | NAMES <name> ...] [SUFFIX <suffix>] [REQUIRED])`

Zephyr file function extension. This function currently supports the following modes:

`APPLICATION_ROOT`
Check all paths in provided variable, and convert those paths that are defined with `-D<path>=<val>` to absolute path, relative from `APPLICATION_SOURCE_DIR`. Issue an error for any relative path not specified by user with `-D<path>`.

`BASE_DIR <base-dir>`
Convert paths relative to `<base-dir>` instead of `APPLICATION_SOURCE_DIR`.

Returns an updated list of absolute paths.

`CONF_FILES`
Find all configuration files in the list of paths and return them in a list. If paths is empty then no configuration files are returned.

Configuration files will be:

- `DTS`: Overlay files (`.overlay`)
- `Kconfig`: Config fragments (`.conf`)
- `defconfig`: defconfig files (`_defconfig`)

The conf file search will return existing configuration files for the current board.

`CONF_FILES` takes the following additional arguments:

`BOARD <board>`
Find configuration files for specified board.

`BOARD_REVISION <revision>`
Find configuration files for specified board revision. Requires `BOARD` to be specified. If no board is given the current `BOARD` and `BOARD_REVISION` will be used, unless `NAMES` are specified.

`NAMES <name1> [name2] ...`
List of file names to look for and instead of creating file names based on board settings. Only the first match found in `<paths>` will be returned in the `<list>`.

`DTS <list>`
List to append DTS overlay files in `<path>` to.

`KCONF <list>`
List to append Kconfig fragment files in `<path>` to.

`DEFCONF <list>`
List to append _defconfig files in `<path>` to.

`SUFFIX <name>`
Suffix name to check for instead of the default name but with a fallback to the default name if not found.

For example: `SUFFIX fish`, will look for `<file>_fish.conf` and use if found but will use `<file>.conf` if not found.

`REQUIRED`
Option to indicate that the `<list>` specified by `DTS` or `KCONF` must contain at least one element, else an error will be raised.

`zephyr_file_copy(<oldname> <newname> [ONLY_IF_DIFFERENT])`

Zephyr file copy extension.

Deprecated: this function only existed because `file(COPY_FILE...)` was not available with CMake 3.20; call `file(COPY_FILE ...)` directly instead.

`oldname`
Old file name.

`newname`
New file name.

`ONLY_IF_DIFFERENT`
Only copy if the files are different.

`zephyr_file_suffix(<filename> SUFFIX <suffix>)`

Update the filename(s) with the suffix if a file with the suffix exists.

This function checks the provided filename or list of filenames to see if a file with the `_<suffix>` extension exists. If so, it updates the supplied variable/list with the new path/paths.

`<filename>`
Variable (singular or list) of absolute path filename(s).

`<suffix>`
The suffix to test for and append to the end of the provided filename.

Returns an updated variable of absolute path(s).

Others

`zephyr_string(ESCAPE <out-var> <input> ...)`

`zephyr_string(SANITIZE <out-var> <input> ...)`

`zephyr_string(SANITIZE TOUPPER <out-var> <input> ...)`

Zephyr string function extension.

This function extends the CMake `string` function by providing additional manipulation options for the `<mode>` argument, namely:

`ESCAPE`
Ensure that every character of the input arguments is considered by CMake as a literal by prefixing the escape character \ where appropriate. This is useful for handling

Windows path separators in strings, or when it is desired to write \n as an actual string of four characters instead of a single newline. Note that this operation must be performed exactly once during the lifetime of a string, or previous escape characters will be treated as literals and escaped further.

SANITIZE
Ensure that the output string does not contain any special characters. Special characters, such as -, +, =, $, etc. are converted to underscores _. Multiple arguments are concatenated.

SANITIZE TOUPPER
Ensure that the output string does not contain any special characters. Special characters, such as -, +, =, $, etc. are converted to underscores _. Multiple arguments are concatenated. The sanitized string will be returned in UPPER case.

Returns the updated string in <out-var>.

zephyr_list(TRANSFORM <list> <ACTION> [OUTPUT_VARIABLE <output variable>])

Zephyr list function extension.

Like CMake's list(TRANSFORM ...) this is intended as a placeholder for storing current and future Zephyr-related extensions for list processing.

<ACTION>
This currently must be NORMALIZE_PATHS. This action converts the argument list <list> to a ;-list with CMake path names, after passing its contents through a configure_file transformation. The input list may be whitespace- or semicolon-separated.

OUTPUT_VARIABLE
the result is normally stored in place, but an alternative variable to store the result can be provided with this.

zephyr_get(<variable> [MERGE [REVERSE]] [SYSBUILD [LOCAL|GLOBAL]] [VAR <var1> ...])

Return the value of <variable> as local scoped variable of same name.

If MERGE is supplied, will return a list of found items.

If REVERSE is supplied together with MERGE, the order of the list will be reversed before being returned. Reverse will happen before the list is returned and hence it will not change the order of precedence in which the list itself is constructed.

VAR can be used either to store the result in a variable with a different name, or to look for values from multiple variables.

```
zephyr_get(FOO VAR FOO_A FOO_B)
zephyr_get(FOO MERGE VAR FOO_A FOO_B)
```

zephyr_get() is a common function to provide a uniform way of supporting build settings that can be set from sysbuild, CMakeLists.txt, CMake cache, or in environment.

The order of precedence for variables defined in multiple scopes:

- Sysbuild defined when sysbuild is used. Sysbuild variables can be defined as global or local to specific image. Examples:
  - BOARD is considered a global sysbuild cache variable
  - blinky_BOARD is considered a local sysbuild cache variable only for the blinky image.

  If no sysbuild scope is specified, GLOBAL is assumed. If using MERGE then SYSBUILD GLOBAL will get both the local and global sysbuild scope variables (in that order, if both exist).
- CMake cache, set by -D<var>=<value> or set(<var> <val> CACHE ...)
- Environment

- Locally in CMakeLists.txt before `find_package(Zephyr)`

For example, if *ZEPHYR_TOOLCHAIN_VARIANT* is set in environment but locally overridden by setting `ZEPHYR_TOOLCHAIN_VARIANT` directly in the CMake cache using `-DZEPHYR_TOOLCHAIN_VARIANT=<val>`, then the value from the cache is returned.

zephyr_create_scope(<scope>)

Create a new scope for the creation of scoped variables.

<scope>
Name of new scope.

zephyr_scope_exists(<result> <scope>)

Check if `<scope>` exists.

<result>
Variable to set with result. `TRUE` if scope exists, `FALSE` otherwise.

<scope>
Name of scope.

zephyr_get_scoped(<output> <scope> <var>)

Get the current value of `<var>` in a specific `<scope>`, as defined by a previous *zephyr_set()* call. The value will be stored in the `<output>` var.

<output>
Variable to store the value in

<scope>
Scope for the variable look up

<var>
Name to look up in the specific scope

zephyr_set(<variable> <value> SCOPE <scope> [APPEND])

Zephyr extension of CMake set which allows a variable to be set in a specific scope.

The scope is used on later *zephyr_get()* invocation for precedence handling when a variable it set in multiple scopes.

<variable>
Name of variable

<value>
Value of variable, multiple values will create a list. The `SCOPE` argument identifies the end of value list.

SCOPE <scope>
Name of scope for the variable

APPEND
Append values to the already existing variable in `<scope>`

zephyr_check_cache(<variable> [REQUIRED] [WATCH])

Check the current CMake cache for `<variable>` and warn the user if the value is being modified.

This can be used to ensure the user does not accidentally try to change Zephyr build variables, such as:

- `BOARD`
- `SHIELD`

<variable>
Name of `<variable>` to check and set, for example `BOARD`.

REQUIRED
Optional flag. If specified, then an unset <variable> will be treated as an error.

WATCH
Optional flag. If specified, watch the variable and print a warning if the variable is later being changed.

Details:

<variable> can be set by 3 sources.

- Using CMake argument, -D<variable>
- Using an environment variable
- In the project CMakeLists.txt before find_package(Zephyr).

CLI has the highest precedence, then comes environment variables, and then finally CMakeLists.txt.

The value defined on the first CMake invocation will be stored in the CMake cache as CACHED_<variable>. This allows the Zephyr build system to detect when a user reconfigures a sticky variable.

A user can ignore all the precedence rules if the same source is always used E.g. always specifies -D<variable>= on the command line, always has an environment <variable> set, or always has a set(<variable> foo) line in his CMakeLists.txt and avoids mixing sources.

The selected <variable> can be accessed through the variable '<variable>' in following Zephyr CMake code.

If the user tries to change <variable> to a new value, then a warning will be printed, and the previously cached value (CACHED_<variable>) will be used, as it has precedence.

Together with the warning, user is informed that in order to change <variable> the build directory must be cleaned.

zephyr_get_targets(<directory> <types> <targets>)

Get build targets for a given directory and sub-directories.

This functions will traverse the build tree, starting from <directory>. It will read the BUILDSYSTEM_TARGETS for each directory in the build tree and return the build types matching the <types> list.

Example of types: OBJECT_LIBRARY, STATIC_LIBRARY, INTERFACE_LIBRARY, UTILITY.

Returns a list of targets in <targets> matching the required <types>.

target_byproducts(TARGET <target> BYPRODUCTS <file> ...)

Specify additional BYPRODUCTS that this target produces.

This function allows the build system to specify additional byproducts to target created with add_executable. When linking an executable the linker may produce additional files, like map files. Those files are not known to the build system. This function makes it possible to describe such additional byproducts in an easy manner.

TARGET <target>
The target to add byproducts to.

BYPRODUCTS <file> ...
List of byproducts that the target produces.

topological_sort(TARGETS <target> ... PROPERTY_NAME <property> RESULT␣ <out-variable>)

This function performs topological sorting of CMake targets using a specific <property>, which dictates target dependencies. A fatal error occurs if the provided dependencies cannot be met, e.g., if they contain cycles.

`TARGETS <target> ...`
List of target names.

`PROPERTY_NAME <property>`
Name of the target property to be used when sorting. For every target listed in `TARGETS`, this property must contain a list (possibly empty) of other targets, which this target depends on for a particular purpose. The property must not contain any target which is not also found in `TARGETS`.

`RESULT <out-variable>`
Output variable, where the topologically sorted list of target names will be returned.

`build_info(<tag>... VALUE <value>...)`
`build_info(<tag>... PATH <path>...)`
Populates the `build_info.yml` info file with exchangeable build information related to the current build.

Example usage:

```
# This will update the 'devicetree.files' key in the build info yaml with the␣
↪list of files
# file1.dts, file2.dts, file3.dts
build_info(devicetree files VALUE file1.dts file2.dts file3.dts)
```

```
# This will place the vendor specific 'foo' key with value 'bar' in the␣
↪vendor-specific
# section of the build info file.
build_info(vendor-specific foo VALUE bar)
```

`<tag>...`
One of the pre-defined valid CMake keys supported by build info or vendor-specific. See scripts/schemas/build-schema.yaml, `cmake` section, for valid tags.

`VALUE <value>...`
Value(s) to place in the build_info.yml file.

`PATH <path>...`
Path(s) to place in the build_info.yml file. All paths are converted to CMake style. If no conversion is required, for example when paths are already guaranteed to be CMake style, then `VALUE` can also be used.

Devicetree extensions

`dt_*` The following methods are for retrieving devicetree information in CMake.

Notes:

- In CMake, we refer to the nodes using the node's path, therefore there is no `dt_path(...)` function for obtaining a node identifier like there is in the C devicetree.h API.
- As another difference from the C API, you can generally use an alias at the beginning of a path interchangeably with the full path to the aliased node in these functions. The usage comments will make this clear in each case.
- These methods are also available to sysbuild. To retrieve the DT information of some <image>, after its CMake configuration step, the `dt_*` function call must include a `TARGET <image>` argument.

`dt_nodelabel(<var> NODELABEL <label> [REQUIRED] [TARGET <target>])`
Function for retrieving the node path for the node having nodelabel `<label>`.

The node's path will be returned in the <var> parameter. <var> will be undefined if node does not exist.

<var>
Return variable where the node path will be stored

NODELABEL <label>
Node label

REQUIRED
Generate a fatal error if the node-label is not found

TARGET <target>
Optional target to retrieve devicetree information from

Example devicetree fragment:

```
/ {
   soc {
       nvic: interrupt-controller@e000e100  { ... };
   };
};
```

Example usage:

```
# Sets 'nvic_path' to "/soc/interrupt-controller@e000e100"
dt_nodelabel(nvic_path NODELABEL "nvic")
```

dt_alias(<var> PROPERTY <prop> [REQUIRED] [TARGET <target>])

Get a node path for an /aliases node property.

The node's path will be returned in the <var> parameter. The variable will be left undefined if the alias does not exist.

<var>
Return variable where the node path will be stored

PROPERTY <prop>
The alias to check

REQUIRED
Generate a fatal error if the alias is not found

TARGET <target>
Optional target to retrieve devicetree information from

Example usage:

```
# The full path to the 'led0' alias is returned in 'path'.
dt_alias(path PROPERTY "led0")

# The variable 'path' will be left undefined for a nonexistent
# alias "does-not-exist".
dt_alias(path PROPERTY "does-not-exist")
```

dt_node_exists(<var> PATH <path> [TARGET <target>])

Tests whether a node with path <path> exists in the devicetree.

The <path> value may be any of these:

- absolute path to a node, like /foo/bar
- a node alias, like my-alias
- a node alias followed by a path to a child node, like my-alias/child-node

The result of the check, either TRUE or FALSE, will be returned in the <var> parameter.

<var>
Return variable where the check result will be returned

PATH <path>
Node path

TARGET <target>
Optional target to retrieve devicetree information from

dt_node_has_status(<var> PATH <path> STATUS <status> [TARGET <target>])

Tests whether <path> refers to a node which:

- exists in the devicetree, and
- has a status property matching the <status> argument (a missing status or an "ok" status is treated as if it were "okay" instead)

The <path> value may be any of these:

- absolute path to a node, like '/foo/bar'
- a node alias, like 'my-alias'
- a node alias followed by a path to a child node, like 'my-alias/child-node'

The result of the check, either TRUE or FALSE, will be returned in the <var> parameter.

<var>
Return variable where the check result will be returned

PATH <path>
Node path

STATUS <status>
Status to check

TARGET <target>
Optional target to retrieve devicetree information from

dt_prop(<var> PATH <path> PROPERTY <prop> [INDEX <idx>] [REQUIRED]␣
[TARGET <target>])

Get a devicetree property value. The value will be returned in the <var> parameter.

The <path> value may be any of these:

- absolute path to a node, like '/foo/bar'
- a node alias, like 'my-alias'
- a node alias followed by a path to a child node, like 'my-alias/child-node'

This function currently only supports properties with the following devicetree binding types: string, int, boolean, array, uint8-array, string-array, path.

For array valued properties (including uint8-array and string-array), the entire array is returned as a CMake list unless INDEX is given. If INDEX is given, just the array element at index <idx> is returned.

The property value will be returned in the <var> parameter if the node exists and has a property <prop> with one of the above types. <var> will be undefined otherwise.

<var>
Return variable where the property value will be stored

PATH <path>
Node path

PROPERTY <prop>
Property for which a value should be returned, as it appears in the DTS source

`INDEX <idx>`
Optional index when retrieving a value in an array property

`REQUIRED`
Generate a fatal error if the property is not found

`TARGET <target>`
Optional target to retrieve devicetree information from

To test if the property is defined before using it, use `DEFINED` on the return `<var>`, like this:

```
dt_prop(reserved_ranges PATH "/soc/gpio@deadbeef" PROPERTY
↪"gpio-reserved-ranges")
if(DEFINED reserved_ranges)
  # Node exists and has the "gpio-reserved-ranges" property.
endif()
```

To distinguish a missing node from a missing property, combine *dt_prop()* and *dt_node_exists()*, like this:

```
dt_node_exists(node_exists PATH "/soc/gpio@deadbeef")
dt_prop(reserved_ranges    PATH "/soc/gpio@deadbeef" PROPERTY
↪"gpio-reserved-ranges")
if(DEFINED reserved_ranges)
  # Node "/soc/gpio@deadbeef" exists and has the "gpio-reserved-ranges"␣
↪property
elseif(node_exists)
  # Node exists, but doesn't have the property, or the property has an␣
↪unsupported type.
endif()
```

`dt_comp_path(<var> COMPATIBLE <compatible> [INDEX <idx>] [TARGET <target>])`
Get a list of paths for the nodes with the given compatible. The value will be returned in the `<var>` parameter. `<var>` will be undefined if no such compatible exists.

For details and considerations about the format of `<path>` and the returned parameter refer to *dt_prop()*.

`<var>`
Return variable where the property value will be stored

`COMPATIBLE <compatible>`
Compatible for which the list of paths should be returned, as it appears in the DTS source

`INDEX <idx>`
Optional index when retrieving a value in an array property

`TARGET <target>`
Optional target to retrieve devicetree information from

`dt_num_regs(<var> PATH <path> [TARGET <target>])`
Get the number of register blocks in the node's `reg` property; this may be zero.

The value will be returned in the `<var>` parameter.

The `<path>` value may be any of these:

- absolute path to a node, like '/foo/bar'
- a node alias, like 'my-alias'
- a node alias followed by a path to a child node, like 'my-alias/child-node'

`<var>`
Return variable where the property value will be stored

`PATH <path>`
Node path

`TARGET <target>`
Optional target to retrieve devicetree information from

`dt_reg_addr(<var> PATH <path> [INDEX <idx>] [NAME <name>] [TARGET <target>])`

Get the base address of the register block at index `<idx>`, or with name `<name>`. If `<idx>` and `<name>` are both omitted, the value at index 0 will be returned. Do not give both `<idx>` and `<name>`.

The value will be returned in the `<var>` parameter.

The `<path>` value may be any of these:

- absolute path to a node, like '/foo/bar'
- a node alias, like 'my-alias'
- a node alias followed by a path to a child node, like 'my-alias/child-node'

Results can be:

- The base address of the register block
- `<var>` will be undefined if node does not exists or does not have a register block at the requested index or with the requested name

`<var>`
Return variable where the address value will be stored

`PATH <path>`
Node path

`INDEX <idx>`
Register block index number

`NAME <name>`
Register block name

`TARGET <target>`
Optional target to retrieve devicetree information from

`dt_reg_size(<var> PATH <path> [INDEX <idx>] [NAME <name>] [TARGET <target>])`

Get the size of the register block at index `<idx>`, or with name `<name>`. If `<idx>` and `<name>` are both omitted, the value at index 0 will be returned. Do not give both `<idx>` and `<name>`.

The value will be returned in the `<value>` parameter.

The `<path>` value may be any of these:

- absolute path to a node, like '/foo/bar'
- a node alias, like 'my-alias'
- a node alias followed by a path to a child node, like 'my-alias/child-node'

`<var>`
Return variable where the size value will be stored

`PATH <path>`
Node path

`INDEX <idx>`
Register block index number

`NAME <name>`
Register block name

`TARGET <target>`
Optional target to retrieve devicetree information from

`dt_has_chosen(<var> PROPERTY <prop> [TARGET <target>])`
Test if the devicetree's /chosen node has a given property `<prop>` which contains the path to a node.

The result of the check, either TRUE or FALSE, will be stored in the `<var>` parameter.

`<var>`
Return variable

`PROPERTY <prop>`
Chosen property

`TARGET <target>`
Optional target to retrieve devicetree information from

Example devicetree fragment:

```
chosen {
        foo = &bar;
};
```

Example usage:

```
# Sets 'result' to TRUE
dt_has_chosen(result PROPERTY "foo")

# Sets 'result' to FALSE
dt_has_chosen(result PROPERTY "baz")
```

`dt_chosen(<var> PROPERTY <prop> [TARGET <target>])`
Get a node path for a `/chosen` node property.

The node's path will be returned in the `<var>` parameter. The variable will be left undefined if the chosen node does not exist.

`<var>`
Return variable where the node path will be stored

`PROPERTY <prop>`
Chosen property

`TARGET <target>`
Optional target to retrieve devicetree information from

`*_if_dt_node` This section is similar to the extensions named `*_ifdef`, except actions are performed if the devicetree contains some node. `*_if_dt_node` functions may be added as needed, or if they are likely to be useful for user applications.

`target_sources_if_dt_node(<path> <target> <scope> <item>...)`
Add item(s) to a target's SOURCES list if a devicetree node exists.

`<path>`
Path to devicetree node to check

`<target>`
Build system target whose sources to add to

`<scope>`
Scope to add items to

`<item>`
Item (or items) to add to the target

Example usage:

```
# If the devicetree alias "led0" refers to a node, this
# adds "blink_led.c" to the sources list for the "app" target.
target_sources_if_dt_node("led0" app PRIVATE blink_led.c)

# If the devicetree path "/soc/serial@4000" is a node, this
# adds "uart.c" to the sources list for the "lib" target,
target_sources_if_dt_node("/soc/serial@4000" lib PRIVATE uart.c)
```

`zephyr_dt_*` The following methods are common code for dealing with devicetree related files in CMake.

Note that functions related to accessing the *contents* of the devicetree belong in section 4.1. This section is just for DT file processing at configuration time.

`zephyr_dt_preprocess(CPP <path> [<argument...>] SOURCE_FILES <file...>␣ OUT_FILE <file> [DEPS_FILE <file> [EXTRA_CPPFLAGS <flag...>]␣ [INCLUDE_DIRECTORIES <dir...>] [WORKING_DIRECTORY <dir>])`

Preprocess one or more devicetree source files. The preprocessor symbol `__DTS__` will be defined. If the preprocessor command fails, a fatal error occurs.

Mandatory arguments:

`CPP <path> [<argument...>]`
path to C preprocessor, followed by any additional arguments

`SOURCE_FILES <file...>`
The source files to run the preprocessor on. These will, in effect, be concatenated in order and used as the preprocessor input.

`OUT_FILE <file>`
Where to store the preprocessor output.

Optional arguments:

`DEPS_FILE <file>`
If set, generate a dependency file here.

`EXTRA_CPPFLAGS <flag...>`
Additional flags to pass the preprocessor.

`INCLUDE_DIRECTORIES <dir...>`
Additional #include file directories.

`WORKING_DIRECTORY <dir>`
where to run the preprocessor.

`zephyr_dt_import(EDT_PICKLE_FILE <file> TARGET <target>)`

Parse devicetree information and make it available to CMake, so that it can be accessed by the `dt_*` CMake extensions from section 4.1.

This requires running a Python script, which can take the output of edtlib and generate a CMake source file from it. If that script fails, a fatal error occurs.

`EDT_PICKLE_FILE <file>`
Input edtlib.EDT object in pickle format

`TARGET <target>`
Target to populate with devicetree properties

Zephyr linker functions

`zephyr_linker*` The following methods are for defining linker structure using CMake functions.

This allows Zephyr developers to define linker sections and their content and have this configuration rendered into an appropriate linker script based on the toolchain in use.

For example:

- ld linker scripts with GNU ld
- ARM scatter files with ARM linker.

Example usage:

```
zephyr_linker_section(
  NAME my_data
  VMA  RAM
  LMA  FLASH
)

# and to configure special input sections for the section
zephyr_linker_section_configure(
  SECTION my_data
  INPUT   "my_custom_data"
  KEEP
)
```

`zephyr_linker([FORMAT <format>] [ENTRY <entry_symbol>])`

Zephyr linker general settings. This function specifies general settings for the linker script to be generated.

`FORMAT <format>`
The output format of the linked executable.

`ENTRY <entry_symbol>`
The code entry symbol.

`zephyr_linker_memory(NAME <name> START <address> SIZE <size> [FLAGS <flags>])`

Zephyr linker memory. This function specifies a memory region for the platform in use.

> **Note**
>
> This function should generally be called with values obtained from devicetree or Kconfig.

`NAME <name>`
Name of the memory region, for example FLASH.

`START <address>`
Start address of the memory region. Start address can be given as decimal or hex value.

`SIZE <size>`
Size of the memory region.

Size can be given as decimal value, hex value, or decimal with postfix k or m. All the following are valid values: `1048576`, `0x10000`, `1024k`, `1024K`, `1m`, and `1M`.

`FLAGS <flags>`
Flags describing properties of the memory region.

- `r`: Read-only region
- `w`: Read-write region

- x: Executable region
- a: Allocatable region
- i: Initialized region
- l: Same as i
- !: Invert the sense of any of the attributes that follow

The flags may be combined like: rx, rx!w.

zephyr_linker_memory_ifdef(<setting> NAME <name> START <address> SIZE <size> [FLAGS <flags>])

Will create memory region if <setting> is enabled.

<setting>
Setting to check for True value before invoking *zephyr_linker_memory()*

See *zephyr_linker_memory()* description for other supported arguments.

zephyr_linker_dts_section(PATH <path>)

Zephyr linker devicetree memory section from path.

This function specifies an output section for the platform in use based on its devicetree configuration.

The section will only be defined if the devicetree exists and has status okay.

PATH <path>
Devicetree node path.

zephyr_linker_dts_memory(PATH <path>)
zephyr_linker_dts_memory(NODELABEL <nodelabel>)
zephyr_linker_dts_memory(CHOSEN <prop>)

Zephyr linker devicetree memory.

This function specifies a memory region for the platform in use based on its devicetree configuration.

The memory will only be defined if the devicetree node or a devicetree node matching the nodelabel exists and has status okay.

Only one of PATH, NODELABEL, and CHOSEN parameters may be given.

PATH <path>
Devicetree node identifier.

NODELABEL <label>
Node label

CHOSEN <prop>
Chosen property, add memory section described by the /chosen property if it exists.

zephyr_linker_group(NAME <name> [VMA <region|group>] [LMA <region|group>] [GROUP <group>] [SYMBOL <SECTION>])

Zephyr linker group. This function specifies a group inside a memory region or another group.

The group ensures that all section inside the group are located together inside the specified group.

This also allows for section placement inside a given group without the section itself needing the precise knowledge regarding the exact memory region this section will be placed in, as that will be determined by the group setup.

Each group will define the following linker symbols:

- `__name_start`: Start address of the group
- `__name_end`: End address of the group
- `__name_size`: Size of the group

Note: `<name>` will be converted to lower casing for linker symbols definitions.

NAME <name>
Name of the group.

VMA <region|group>
VMA Memory region or group to be used for this group. If a group is used then the VMA region of that group will be used.

LMA <region|group>
Memory region or group to be used for this group.

GROUP <group>
Place the new group inside the existing group `<group>`

SYMBOL <SECTION>
Specify that start symbol of the region should be identical to the start address of the first section in the group.

Note: `VMA` and `LMA` are mutual exclusive with `GROUP`

Example:

```
zephyr_linker_memory(NAME memA START ... SIZE ... FLAGS ...)
zephyr_linker_group(NAME groupA LMA memA)
zephyr_linker_group(NAME groupB LMA groupA)
```

will create two groups in same memory region as groupB will inherit the LMA from groupA:

```
+-----------------+
| memory region A |
|                 |
| +-------------+ |
| | groupA      | |
| +-------------+ |
|                 |
| +-------------+ |
| | groupB      | |
| +-------------+ |
|                 |
| +-------------+ |
```

whereas

```
zephyr_linker_memory(NAME memA START ... SIZE ... FLAGS ...)
zephyr_linker_group(NAME groupA LMA memA)
zephyr_linker_group(NAME groupB GROUP groupA)
```

will create groupB inside groupA:

```
+---------------------+
| memory region A     |
|                     |
| +-----------------+ |
| | groupA          | |
| |                 | |
| | +-------------+ | |
```

(continues on next page)

(continued from previous page)

```
| | | groupB      | | |
| | +-------------+ | |
| |                 | |
| +-----------------+ |
|                     |
+---------------------+
```

`zephyr_linker_section(NAME <name> [GROUP <group>] [VMA <region|group>] [LMA <region|group>] [ADDRESS <address>] [ALIGN <alignment>] [SUBALIGN <alignment>] [FLAGS <flags>] [MIN_SIZE <size>] [MAX_SIZE <size>] [HIDDEN] [NOINPUT] [NOINIT] [PASS [NOT] <name>...])`

Zephyr linker output section. This function specifies an output section for the linker.

When using `zephyr_linker_section(NAME <name>)` an output section with `<name>` will be configured. This section will default include input sections of the same name, unless `NOINPUT` is specified. This means the section named `foo` will default include the sections matching `foo` and `foo.*` Each output section will define the following linker symbols:

- *__name*_start: Start address of the section
- *__name*_end: End address of the section
- *__name*_size: Size of the section
- *__name*_load_start: Load address of the section, if VMA = LMA then this value will be identical to *__name*_start

The location of the output section can be controlled using LMA, VMA, and address parameters

NAME <name>
Name of the output section.

VMA <region|group>
VMA Memory region or group where code / data is located runtime (VMA) If `<group>` is used here it means the section will use the same VMA memory region as `<group>` but will not be placed inside the group itself, see also `GROUP` argument.

KVMA <region|group>
Kernel VMA Memory region or group where code / data is located runtime (VMA) When MMU is active and Kernel VM base and offset is different from SRAM base and offset, then the region defined by KVMA will be used as VMA. If `<group>` is used here it means the section will use the same VMA memory region as `<group>` but will not be placed inside the group itself, see also `GROUP` argument.

LMA <region|group>
Memory region or group where code / data is loaded (LMA) If VMA is different from LMA, the code / data will be loaded from LMA into VMA at bootup, this is usually the case for global or static variables that are loaded in rom and copied to ram at boot time. If `<group>` is used here it means the section will use the same LMA memory region as `<group>` but will not be placed inside the group itself, see also `GROUP` argument.

GROUP <group>
Place this section inside the group `<group>`

ADDRESS <address>
Specific address to use for this section.

ALIGN_WITH_INPUT
The alignment difference between VMA and LMA is kept intact for this section.

ALIGN <alignment>
Align the execution address with alignment.

`SUBALIGN <alignment>`
Align input sections with alignment value.

`ENDALIGN <alignment>`
Align the end so that next output section will start aligned. This only has effect on Scatter scripts.

Note: Regarding all alignment attributes. Not all linkers may handle alignment in identical way. For example the Scatter file will align both load and execution address (LMA and VMA) to be aligned when given the ALIGN attribute.

`MIN_SIZE <size>`
Pad section so that it at least `<size>` bytes in size.

`MAX_SIZE <size>`
Check that the sections is not larger than `<size>` bytes.

`NOINPUT`
No default input sections will be defined, to setup input sections for section `<name>`, the corresponding *zephyr_linker_section_configure()* must be used.

`PASS [NOT] <name>...`
Linker pass where this section should be active. By default a section will be present during all linker passes.

`PASS <p1> <p2>...` makes the section present only in the given passes. Empty list means no passes.

`PASS NOT <p1> <p2>...` makes the section present in all but the given passes. Empty list means all passes.

`TYPE <type>`
Tag section for special treatment.

- `NOLOAD`, `BSS` - Ensure that the section is NOLOAD
- `LINKER_SCRIPT_FOOTER` - One single section to be generated last

Note: `VMA` and `LMA` are mutual exclusive with `GROUP`

```
zephyr_linker_section_ifdef(<setting> NAME <name> [GROUP <group>]␣
[VMA <region|group>] [LMA <region|group>] [ADDRESS <address>] [ALIGN <alignment>]␣
[SUBALIGN <alignment>] [FLAGS <flags>] [MIN_SIZE <size>] [MAX_SIZE <size>]␣
[HIDDEN] [NOINPUT] [NOINIT] [PASS [NOT] <name>...])
```

Will create an output section if `<setting>` is enabled.

`<setting>`
Setting to check for True value before invoking *zephyr_linker_section()*

See *zephyr_linker_section()* description for other supported arguments.

```
zephyr_iterable_section(NAME <name> [GROUP <group>]␣
[VMA <region|group>] [LMA <region|group>] [ALIGN_WITH_INPUT] [SUBALIGN <alignment>])
```

Define an output section which will set up an iterable area of equally-sized data structures. For use with `STRUCT_SECTION_ITERABLE`. Input sections will be sorted by name in lexicographical order.

Each list for an input section will define the following linker symbols:

- `_<name>_list_start`: Start of the iterable list
- `_<name>_list_end`: End of the iterable list

The output section will be named *<name>_area* and define the following linker symbols:

Symbol	Description
`__name_area_start`	Start address of the section
`__name_area_end`	End address of the section
`__name_area_size`	Size of the section
`__name_area_load_`	Load address of the section, if VMA = LMA then this value will be identical to `__name_area_start`

The options are:

`NAME <name>`
Name of the struct type, the output section be named accordingly as: <name>_area.

`VMA <region|group>`
VMA Memory region where code / data is located runtime (VMA)

`LMA <region|group>`
Memory region where code / data is loaded (LMA) If VMA is different from LMA, the code / data will be loaded from LMA into VMA at bootup, this is usually the case for global or static variables that are loaded in rom and copied to ram at boot time.

`GROUP <group>`
Place this section inside the group <group>

`ADDRESS <address>`
Specific address to use for this section.

`ALIGN_WITH_INPUT`
The alignment difference between VMA and LMA is kept in tact for this section.

`NUMERIC`
Use numeric sorting.

`SUBALIGN <alignment>`
Force input alignment with size <alignment>

> **Note**
>
> Regarding all alignment attributes. Not all linkers may handle alignment in identical way. For example the Scatter file will align both load and execution address (LMA and VMA) to be aligned when given the ALIGN attribute.

`zephyr_linker_section_obj_level(SECTION <section> LEVEL <level>)`
Generate a symbol to mark the start of the objects array for the specified object and level, then link all of those objects (sorted by priority). Ensure the objects aren't discarded if there is no direct reference to them.

This is useful content such as struct devices.

For example: `zephyr_linker_section_obj_level(SECTION init LEVEL PRE_KERNEL_1)` will create an input section matching `.z_init_PRE_KERNEL_P_1_SUB_?_`, `.z_init_PRE_KERNEL_P_1_SUB_??_`, and `.z_init_PRE_KERNEL_P_1_SUB_???_`.

`SECTION <section>`
Section in which the objects shall be placed

`LEVEL <level>`
Priority level, all input sections matching the level will be sorted.

```
zephyr_linker_section_configure(SECTION <section> [ALIGN <alignment>]␣
[PASS [NOT] <name>] [PRIO <no>] [SORT <sort>] [MIN_SIZE <size>] [MAX_SIZE <size>]␣
[ANY] [FIRST] [KEEP] [INPUT <input>] [SYMBOLS [<start> [<end>]]])
```

Configure an output section with additional input sections. An output section can be configured with additional input sections besides its default section. For example, to add the input section `foo` to the output section bar, with `KEEP` attribute, call: `zephyr_linker_section_configure(SECTION bar INPUT foo KEEP)`

SECTION <section>
The output section to configure

ALIGN <alignment>
Will align the input section placement inside the load region with `<alignment>`

FIRST
The first input section in the list should be marked as first section in output.

SORT <NAME>
Sort the input sections according to `<type>`. Currently only `NAME` is supported.

MIN_SIZE <size>
Pad section so that it at least `<size>` bytes in size.

MAX_SIZE <size>
Check that the sections is not larger than `<size>` bytes.

KEEP
Do not eliminate input section during linking

PRIO
The priority of the input section. Per default, input sections order is not guaranteed by all linkers, but using priority Zephyr CMake linker will create sections such that order can be guaranteed. All unprioritized sections will internally be given a CMake process order priority counting from 100, so first unprioritized section is handled internal prio 100, next 101, and so on. To ensure a specific input section come before those, you may use `PRIO 50`, `PRIO 20` and so on. To ensure an input section is at the end, it is advised to use `PRIO 200` and above.

PASS [NOT] <pass>...
Control in which linker passes this piece is present See *zephyr_linker_section()* PASS for details.

FLAGS <flags>
Special section flags such as "+RO", +XO, "+ZI". The FLAGS and ANY arguments only has effect for scatter files.

ANY
ANY section flag in scatter file. The FLAGS and ANY arguments only has effect for scatter files.

INPUT <input>
Input section name or list of input section names. `<input>` is either just a section name ".data*" or `<file-pattern>(<section-patterns>... )` `<file-pattern>` is `[library.a:]file` e.g. `foo.a:bar.o(.data*)`

SYMBOLS <start> <end>
Generate start and end symbols for the input section.

zephyr_linker_symbol(SYMBOL <name> EXPR <expr>)

Add additional user defined symbol to the generated linker script.

SYMBOL <name>
Symbol name to be available.

EXPR <expr>
Expression that defines the symbol. Due to linker limitations all expressions should only contain simple math, such as `+`, `-`, `*` and similar. The expression will go directly into the linker, and all `@<symbol>@` will be replaced with the referred symbol.

Example:
To create a new symbol bar pointing to the start VMA address of section foo + 1024, one can write:

```
zephyr_linker_symbol(SYMBOL bar EXPR "(@foo@ + 1024)")
```

`zephyr_linker_include_generated((CMAKE | KCONFIG | HEADER) <name> [PASS [NOT] <pass>...])`

Add file that is generated at build-time to be included when running the linker script generator.

`CMAKE <name>`
includes the given cmake file

`KCONFIG <name>`
import_kconfig() the given Kconfig file. gives `@variable@` access to all the CONFIG_FOO settings

`HEADER <name>`
finds all #define FOO value in name. Plain regex, no proper preprocessing.

`PASS [NOT] <pass>...`
Rule for which PASSES to include file. see *zephyr_linker_section()* PASS

`zephyr_linker_include_var(VAR <name> [VALUE <value>] [PASS [NOT] <pass>...])`

Save the value of `<name>` for when the generator is running at build-time. If `VALUE` isn't set, the current value of the variable is used.

`VAR <name>`
Variable to be set

`VALUE <value>`
The value

`PASS [NOT] <pass>...`
Rule for which PASSES to include variable see *zephyr_linker_section()* PASS for details.

`zephyr_linker_generate_linker_settings_file(<file_name>)`

Generate a file for the settings to the linker file generator script.

`<file_name>`
File to be created

Function helper macros Set of CMake macros to facilitate argument processing when defining functions.

`zephyr_check_arguments_required(<function_name> <prefix> <arg1> [<arg2> ...])`

Helper macro for verifying that at least one of the required arguments has been provided by the caller.

A `FATAL_ERROR` will be raised if not one of the required arguments has been passed by the caller.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<prefix>`
Prefix used in `cmake_parse_arguments`, usually `arg` or the function name in uppercase.

`<arg>`
One or more arguments to check.

`zephyr_check_arguments_required_allow_empty(<function_name> <prefix> <arg1> [<arg2> ...])`

Helper macro for verifying that at least one of the required arguments has been provided by the caller. Arguments with empty values are allowed.

A `FATAL_ERROR` will be raised if not one of the required arguments has been passed by the caller.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<prefix>`
Prefix used in `cmake_parse_arguments`, usually `arg` or the function name in uppercase.

`<arg>`
One or more arguments to check.

`zephyr_check_flags_required(<function_name> <prefix> <flag1> [<flag2> ...])`

Helper macro for verifying that at least one of the required flags has been provided by the caller.

A `FATAL_ERROR` will be raised if not one of the required arguments has been passed by the caller.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<prefix>`
Prefix used in `cmake_parse_arguments`, usually `arg` or the function name in uppercase.

`<flag>`
One or more flags to check.

`zephyr_check_arguments_required_all(<function_name> <prefix> <arg1> [<arg2> ...])`

Helper macro for verifying that all the required arguments have been provided by the caller.

A `FATAL_ERROR` will be raised if one of the required arguments is missing.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<prefix>`
Prefix used in `cmake_parse_arguments`, usually `arg` or the function name in uppercase.

`<arg>`
One or more arguments to check.

`zephyr_check_arguments_exclusive(<function_name> <prefix> <arg1> <arg2> [<arg3> ...])`

Helper macro for verifying that none of the mutual exclusive arguments are provided together.

A `FATAL_ERROR` will be raised if any of the arguments are given together.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<prefix>`
Prefix used in `cmake_parse_arguments`, usually `arg` or the function name in uppercase.

`<arg>`
One or more arguments to check.

`zephyr_check_flags_exclusive(<function_name> <prefix> <flag1> <flag2> [<flag3> ... ])`

Helper macro for verifying that none of the mutual exclusive flags are provided together.

A `FATAL_ERROR` will be raised if any of the flags are given together.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<prefix>`
Prefix used in `cmake_parse_arguments`, usually `arg` or the function name in uppercase.

`<flag>`
One or more flags to check.

`zephyr_check_no_arguments(<function_name> <arg1> [<arg2> ...])`

Helper macro for verifying that no unexpected arguments are provided.

A `FATAL_ERROR` will be raised if any unexpected argument is given.

`<function_name>`
Name of the function the check is performed for. Used in error message generation.

`<arg>`
One or more arguments to check.

Linkable loadable extensions (llext) These functions simplify the creation and management of *Linkable Loadable Extensions (LLEXT)*.

Configuration functions The following functions simplify access to the compilation/link stage properties of an llext using the same API as CMake's `target_*` functions.

`llext_compile_definitions(target_name <defs>...)`

Add compile definitions to the llext target.

`target_name`
Name of the llext target.

`<defs>`
Arguments passed to `target_compile_definitions`.

`llext_compile_features(target_name <features>...)`

Add compile features to the llext target.

`target_name`
Name of the llext target.

`<features>`
Arguments passed to `target_compile_features`.

`llext_compile_options(target_name <options>...)`

Add compile options to the llext target.

`target_name`
Name of the llext target.

`<options>`
Arguments passed to `target_compile_options`.

`llext_include_directories(target_name <dirs>...)`

Add include directories to the llext target.

`target_name`
Name of the llext target.

`<dirs>`
Arguments passed to `target_include_directories`.

`llext_link_options(target_name <options>...)`
Add link options to the llext target.

`target_name`
Name of the llext target.

`<options>`
Arguments passed to `target_link_options`.

Build control functions The following functions add targets and subcommands to the build system to compile and link an llext.

`add_llext_target(target_name OUTPUT <output_file> SOURCES <source_files>)`
Add a custom target that compiles a set of source files to a .llext file.

Output and source files must be specified using the `OUTPUT` and `SOURCES` arguments. Only one source file is supported when `CONFIG_LLEXT_TYPE_ELF_OBJECT` is selected, since there is no linking step in that case.

The llext code will be compiled with mostly the same C compiler flags used in the Zephyr build, but with some important modifications. The list of flags to remove and flags to append is controlled respectively by the *LLEXT_REMOVE_FLAGS* and *LLEXT_APPEND_FLAGS* global variables.

The following custom properties of `target_name` are defined and can be retrieved using the `get_target_property` function:

- `lib_target`: Target name for the source compilation and/or link step.
- `lib_output`: The binary file resulting from compilation and/or linking steps.
- `pkg_input`: The file to be used as input for the packaging step.
- `pkg_output`: The final .llext file.

Example usage:

```
add_llext_target(hello_world
  OUTPUT  ${PROJECT_BINARY_DIR}/hello_world.llext
  SOURCES ${PROJECT_SOURCE_DIR}/src/llext/hello_world.c
)
```

This will compile the source file `src/llext/hello_world.c` to a file named *$PROJECT_BINARY_DIR*`/hello_world.llext`.

`add_llext_command(TARGET <target_name> PRE_BUILD <command> [...])`
`add_llext_command(TARGET <target_name> POST_BUILD <command> [...])`
`add_llext_command(TARGET <target_name> POST_PKG <command> [...])`
Add a custom command to an llext target that will be executed during the build. The command will be executed at the specified build step and can refer to `<target>`'s properties for build-specific details.

The different build steps are:

`PRE_BUILD`
Before the llext code is linked, if the architecture uses dynamic libraries. This step can access `lib_target` and its own properties.

`POST_BUILD`
After the llext code is built, but before packaging it in an .llext file. This step is expected to create a `pkg_input` file by reading the contents of `lib_output`.

POST_PKG
After the .llext file has been created. This can operate on the final llext file pkg_output.

Anything else after COMMAND will be passed to add_custom_command as-is (including multiple commands and other options).

kconfig

This module contains the Kconfig configuration logic.

It provides the following targets:

- menuconfig: Run the Kconfig menuconfig tool.
- guiconfig: Run the Kconfig guiconfig tool.
- hardenconfig: Run the Kconfig hardenconfig tool.
- traceconfig: Run the Kconfig traceconfig tool.

Variables After loading this module, the following global variables are defined:

KCONFIG_ROOT
Path to the Kconfig root file. Default: ${ZEPHYR_BASE}/Kconfig.

AUTOCONF_H
Path to the generated autoconf header file. Default: ${PROJECT_BINARY_DIR}/include/generated/zephyr/autoconf.h.

DOTCONFIG
Path to the generated .config file. Default: ${PROJECT_BINARY_DIR}/.config.

EXTRA_KCONFIG_TARGETS
List of extra Kconfig targets to add.

EXTRA_KCONFIG_TARGET_COMMAND_FOR_<target>
Command to run for the extra Kconfig target.

The Kconfig fragments this module reads are selected by *CONF_FILE* and *EXTRA_CONF_FILE*, which are resolved by the *configuration_files* module.

5.2.2 Sysbuild Modules

sysbuild_extensions

Sysbuild's CMake extension commands.

This module defines the commands used to describe a *sysbuild* configuration: which images take part in the build, how they are configured, and in which order they are configured and flashed.

The commands documented below are available in the sysbuild.cmake file of any application or Zephyr module that participates in a sysbuild build.

Commands not documented here are used by sysbuild itself and may be removed, renamed, or re-purposed without prior notice.

- *Setting Kconfig values on an image*

`sysbuild_get(<variable> IMAGE <image> [VAR <image-variable>] <KCONFIG|CACHE>)`

Read a CMake cache variable or Kconfig setting from an image.

Sysbuild can only read an image once that image has been configured, for example from the post-CMake hook of a *sysbuild module*.

If `VAR` is provided, the name given as parameter will be looked up, but if `VAR` is not given, the `<variable>` name provided will be used both for lookup and value return.

The result will be returned in `<variable>`.

`<variable>`
Variable used for returning the value. Also used as lookup variable if `VAR` is not provided.

`IMAGE <image>`
Name of the image to read the variable from.

`VAR <image-variable>`
Name of the variable to look up.

`KCONFIG`
Flag indicating that a Kconfig setting should be fetched.

`CACHE`
Flag indicating that a CMake cache variable should be fetched.

Example usage:

```
# Look up PROJECT_NAME from the CMake cache of `my_sample` and return it
# in the local variable `PROJECT_NAME`.
sysbuild_get(PROJECT_NAME IMAGE my_sample CACHE)

# Same lookup, but returned in `my_sample_PROJECT_NAME`.
sysbuild_get(my_sample_PROJECT_NAME IMAGE my_sample VAR PROJECT_NAME CACHE)

# Look up the Kconfig setting CONFIG_FOO of `my_sample`.
sysbuild_get(my_sample_CONFIG_FOO IMAGE my_sample VAR CONFIG_FOO KCONFIG)
```

`ExternalZephyrProject_Add(APPLICATION <name> SOURCE_DIR <dir>␣`
`[BOARD <board> [BOARD_REVISION <revision>]]␣`
`[APP_TYPE <MAIN|BOOTLOADER|FIRMWARE_LOADER>] [BUILD_ONLY <bool>] )`

Include a Zephyr based build system into the multi-image build system.

`APPLICATION <name>`
Name of the application. The name will also be used for the build folder of the application.

`SOURCE_DIR <dir>`
Source directory of the application.

`BOARD <board>`
Use `<board>` for the application build instead of the user defined *BOARD*.

`BOARD_REVISION <revision>`
Use `<revision>` of `<board>` for the application. Only valid if `BOARD` is also supplied.

`APP_TYPE <MAIN|BOOTLOADER|FIRMWARE_LOADER>`
Application type.

`MAIN` indicates this application is the main application, and where user defined settings should be passed on as-is except for multi image build flags. For example, `-DCONF_FILE=<files>` will be passed on to the main application unmodified.

`BOOTLOADER` indicates this application is a bootloader.

FIRMWARE_LOADER indicates this application is a firmware loader image for MCUboot.

BUILD_ONLY <bool>
Mark the application as build-only. If <bool> evaluates to true, then this application will be excluded from flashing and debugging.

See *Adding Zephyr applications to sysbuild* for a worked example.

```
ExternalZephyrVariantProject_Add(APPLICATION <name> SOURCE_APP <name>
[SNIPPET <snippet>] [EXTRA_DTC_OVERLAY_FILE <file>] [EXTRA_CONF_FILE <file>]
[BUILD_ONLY <bool>] )
```

Duplicate an existing Zephyr based build system into the multi-image build system with a specified modification.

This will not create the extra build targets that *ExternalZephyrProject_Add()* adds, for example <app>_menuconfig.

Note

The variant image must have a *CONF_FILE*, *EXTRA_CONF_FILE*, *EXTRA_DTC_OVERLAY_FILE* or SNIPPET added to it, or it will be invalid and image configuration will result in a fatal error.

APPLICATION <name>
Name of the application. The name will also be used for the build folder of the application.

SOURCE_APP <name>
Name of the existing image to use for duplication.

SNIPPET <snippet>
List of default snippets to apply for the variant image.

EXTRA_DTC_OVERLAY_FILE <file>
List of default extra devicetree overlay files to apply for the variant image.

EXTRA_CONF_FILE <file>
List of default extra Kconfig fragments to apply for the variant image.

BUILD_ONLY <bool>
Mark the application as build-only. If <bool> evaluates to true, then this application will be excluded from flashing and debugging.

```
sysbuild_cache_set(VAR <variable> [APPEND [REMOVE_DUPLICATES]] <value>)
```

Set a sysbuild cache variable, which can then be accessed by images.

The result will be returned in <variable>.

VAR <variable>
Name of the variable in the CMake cache.

APPEND
If specified, append the supplied data to the existing value as a list.

REMOVE_DUPLICATES
If specified, remove duplicate entries contained within the list before saving to the cache.

<value>
Value to set or update.

Example usage:

```
# Add `battery` to the `ATTRIBUTES` list in the CMake cache, removing
# any duplicates from the list.
sysbuild_cache_set(VAR ATTRIBUTES APPEND REMOVE_DUPLICATES battery)
```

Setting Kconfig values on an image The following commands add a Kconfig fragment line to <image>. They are the supported way for a sysbuild.cmake file to configure an image it has added.

The collected fragment is passed to the image as a forced input configuration, so it is applied last and takes precedence over the image's own Kconfig fragments. See *Kconfig namespacing* for how these relate to SB_CONFIG_ options.

set_config_bool(<image> <setting> <value>)

Set a boolean Kconfig option to y if <value> evaluates to true, and to n otherwise.

set_config_string(<image> <setting> <value>)

Set a string Kconfig option to <value>.

set_config_int(<image> <setting> <value>)

Set an int or hex Kconfig option to <value>.

Example usage:

```
set_config_bool(${DEFAULT_IMAGE} CONFIG_BOOTLOADER_MCUBOOT y)
set_config_string(${DEFAULT_IMAGE} CONFIG_MCUBOOT_SIGNATURE_KEY_FILE "$
↪{keyfile}")
set_config_int(mcuboot CONFIG_LOG_MAX_LEVEL 2)
```

sysbuild_add_subdirectory(<source_dir> [<binary_dir>])

Add a subdirectory, and recursively process the sysbuild images it adds.

This command extends add_subdirectory with additional, recursive processing of the sysbuild images added via <source_dir>.

After exiting <source_dir>, this command will take every image added so far and include() its sysbuild.cmake file, if found. If more images get added at this stage, their sysbuild.cmake files will be included as well, and so on. This continues until all expected images have been added, before returning.

sysbuild_add_dependencies(<CONFIGURE | FLASH> <image> [<image-dependency> ...])

Make an image depend on other images in the configuration or flashing order.

Each image named <image-dependency> will be ordered before the image named <image>.

CONFIGURE

Add CMake configuration dependencies. This will determine the order in which images are configured.

FLASH

Add flashing dependencies. This will determine the order in which all images will appear in domains.yaml.

See *Adding dependencies among Zephyr applications* for a worked example.

sysbuild_images_order(<variable> <CONFIGURE | FLASH> IMAGES <images>)

Sort the provided <images> to satisfy the dependencies specified using *sysbuild_add_dependencies()*.

The result will be returned in <variable>.

sysbuild_kconfig

Sysbuild's own Kconfig configuration.

Sysbuild has a Kconfig tree of its own, rooted at the application's `Kconfig.sysbuild` file and namespaced under `SB_CONFIG_` so that it cannot collide with an image's own `CONFIG_` options. See *Kconfig namespacing*.

This module locates the sysbuild configuration files and runs Kconfig over them. It also defines the `sysbuild_menuconfig` and `sysbuild_guiconfig` build targets, which edit sysbuild's own configuration rather than that of any image.

Configuration input files are looked up in *SB_APPLICATION_CONFIG_DIR*, falling back to *APPLICATION_CONFIG_DIR*, and are controlled by:

- *SB_CONF_FILE*
- *SB_EXTRA_CONF_FILE*

sysbuild_root

Convert Zephyr roots to absolute paths to be used by sysbuild.

This module converts all relative paths in the following root lists to absolute paths, relative from *APP_DIR*:

- *ARCH_ROOT*
- *BOARD_ROOT*
- *MODULE_EXT_ROOT*
- *SCA_ROOT*
- *SNIPPET_ROOT*
- *SOC_ROOT*

If a root is defined, this module checks the list of paths in the root, converts any relative path to an absolute path, and updates the root list. If a root is undefined, it is still undefined once this module has loaded.

Converted paths are placed in the CMake cache so that they are propagated correctly to image builds.

native_simulator_sb_extensions

Sysbuild extension commands for native_sim based targets.

These commands are used by the `sysbuild.cmake` file of a board or application that builds several images into a single native simulator executable. They are no-ops unless the board target is a native simulator or bsim based one.

`native_simulator_set_final_executable(<final_image>)`

Add a build target which copies the executable produced by `<final_image>` to the top level, as `zephyr/zephyr.exe`.

`<final_image>` is expected to have been set to assemble other dependent images into itself if necessary, by calling *native_simulator_set_child_images()*.

This allows other tools, like twister or the bsim test scripts, as well as users, to find this final executable in the same place as for non-sysbuild builds.

`native_simulator_set_child_images(<final_image> <child_image>)`

Set `<child_image>` as a dependency of `<final_image>`, and configure the final image to assemble the child image into its final executable.

`native_simulator_set_primary_mcu_index(<image> [<image2> ...])`

Propagate the `SB_CONFIG_NATIVE_SIMULATOR_PRIMARY_MCU_INDEX` setting, if it is set, to the `CONFIG_NATIVE_SIMULATOR_PRIMARY_MCU_INDEX` of each given image.

5.2.3 Utility & Tooling Modules

doc

This CMake module will load all Zephyr CMake modules required for a documentation build.

The following CMake modules will be loaded:

- *extensions*
- *python*
- *west*
- `root`
- `zephyr_module`

Outcome: The Zephyr package required for documentation build setup.

python

Configure Python for use in Zephyr build system.

This module finds and configures Python for use in the Zephyr build system. It ensures a compatible Python version is available and sets up necessary environment variables and CMake variables.

Variables The *PYTHON_MINIMUM_REQUIRED* variable is used to specify the minimum required Python version.

The following variables will be defined when this CMake module completes:

- *PYTHON_EXECUTABLE*

Search Process The module searches for Python in the following order:

1. Using the `Python3_EXECUTABLE` if already defined
2. Using the `WEST_PYTHON` if defined
3. Searching for "python" or "python3" in the system PATH and virtual environment
4. Using CMake's `find_package(Python3)` mechanism

The module verifies that any found Python meets the minimum version requirement.

Windows-Specific Configuration On Windows systems, the module sets the `PYTHONIOENCODING` environment variable to "utf-8" to ensure consistent encoding behavior when Python is invoked from CMake.

Example Usage

```
include(python)

# Now PYTHON_EXECUTABLE can be used to invoke Python scripts
execute_process(COMMAND ${PYTHON_EXECUTABLE} my_script.py)
```

west

This module finds and configures the `west` meta-tool for use in the Zephyr build system. It ensures that west is available and correctly configured.

Variables The following variables will be defined when this CMake module completes:

- *WEST*
- *WEST_TOPDIR*

git

Provides Git-related functionality for the Zephyr build system.

This module provides functions for interacting with Git repositories within the Zephyr build system.

Commands

git_describe

Get a short Git description associated with a directory or file.

```
git_describe(<file|dir> OUTPUT)
```

This function runs `git describe --abbrev=12 --always` on the provided directory, or if provided a file, the directory containing that file. In both cases, the result is stored in the provided output variable.

file|dir
The directory or file to run the git command for.

OUTPUT
The variable name where the git description will be stored. If the git command fails or Git is not found, this variable will not be set.

The function will output status messages if:

- The **git** command fails (error message)
- The **git** command produces warnings (warning message)

Example Usage

```
include(git)

git_describe(${CMAKE_CURRENT_SOURCE_DIR} GIT_DESCRIPTION)
if(DEFINED GIT_DESCRIPTION)
  message(STATUS "Git description: ${GIT_DESCRIPTION}")
endif()
```

yaml

Provides YAML file handling functionality for the Zephyr build system.

This module offers support for reading, writing, and manipulating YAML files within the Zephyr build system. It supports basic key-value pairs, nested objects, lists, and lists of maps.

Supported YAML Features The module supports the following YAML constructs:

Basic key-value pairs

```
foo: bar
```

Nested objects

```
foo:
  bar: baz
```

Simple lists

```
foos:
  - foo1
  - foo2
  - foo3
```

Lists of maps

```
foo:
  - bar: val1
    baz: val1
  - bar: val2
    baz: val2
```

Example Usage

```
include(yaml)

# Create a new YAML context
yaml_create(NAME my_yaml FILE my_file.yaml)

# Set some values
yaml_set(NAME my_yaml KEY foo VALUE bar)
yaml_set(NAME my_yaml KEY list LIST item1 item2 item3)

# Append a map to a list
yaml_set(NAME my_yaml KEY list APPEND LIST MAP "name: test, value: 42")

# Save the context to file
yaml_save(NAME my_yaml)

# my_file.yaml now contains:
#
# foo: bar
# list:
#   - item1
#   - item2
#   - item3
#   - name: test
#     value: 42
```

Commands

`yaml_context`

Query the status of a YAML context.

```
yaml_context(EXISTS NAME <name> <result>)
```

`EXISTS`

Check if the YAML context exists in the current scope. If the context exists, then TRUE

is returned in `<result>`.

`NAME <name>`
Name of the YAML context.

`<result>`
Variable to store the result of the query.

`yaml_create`
Create a new empty YAML context.

```
yaml_create(NAME <name> [FILE <file>])
```

`NAME <name>`
Name of the YAML context.

`FILE <file>`
Path to file to be used together with this YAML context. This file will be used when *yaml_save()* is called.

`yaml_load`
Load an existing YAML file into a YAML context.

```
yaml_load(FILE <file> NAME <name>)
```

`FILE <file>`
Path to file to load.

`NAME <name>`
Name of the YAML context.

`yaml_get`
Get the value of a key from a YAML context.

```
yaml_get(<out-var> NAME <name> KEY <key>...)
```

`NAME <name>`
Name of the YAML context.

`KEY <key>...`
Name of key(s) to retrieve. Multiple keys can be specified for nested access.

`<out-var>`
Variable to store the retrieved value.

If the key represents a list, the list is returned. Behavior is undefined if the key points to a complex object.

`yaml_length`
Get the length of an array defined by a key.

```
yaml_length(<out-var> NAME <name> KEY <key>...)
```

`NAME <name>`
Name of the YAML context.

`KEY <key>...`
Name of key defining the list.

`<out-var>`
Variable to store the length.

Returns -1 if the key does not define an array.

yaml_set

Set a value or list of values in a YAML context.

```
yaml_set(NAME <name> KEY <key>... [GENEX] VALUE <value>)
yaml_set(NAME <name> KEY <key>... [APPEND] [GENEX] LIST <value>...)
yaml_set(NAME <name> KEY <key>... [APPEND] LIST MAP <map1> MAP <map2> MAP ...)
```

NAME <name>
Name of the YAML context.

KEY <key>...
Name of key(s) to set.

VALUE <value>
New value for the key.

LIST <values>
New list of values for the key.

APPEND
Append the list of values to the existing list.

GENEX
The value(s) contain generator expressions.

MAP <map>
Map with key-value pairs where pairs are separated by ',' and key-value by ':'. Format: `"<key1>: <value1>, <key2>: <value2>, ..."`. Multiple MAP arguments can be given to separate maps when adding to a list. Cannot be used with GENEX.

Note: If a map value contains commas, the value must be quoted in single quotes and commas must be double escaped: `'A \\,string'`

yaml_remove

Remove a key from a YAML context.

```
yaml_remove(NAME <name> KEY <key>...)
```

NAME <name>
Name of the YAML context.

KEY <key>...
Name of key(s) to remove. Multiple keys can be specified for nested access.

yaml_save

Write a YAML context to a file.

```
yaml_save(NAME <name> [FILE <file>])
```

NAME <name>
Name of the YAML context.

FILE <file>
Path to file to write the context. If not given, uses the FILE property of the YAML context.

If the context uses generator expressions, keys containing them will initially be written as comments. The full contents will be available at build time. Build steps depending on the file being complete must depend on the *name*_yaml_saved target.

5.2.4 Target properties

LIBC_LINK_LIBRARIES

A list of libraries that are linked to the C standard library when it is built. This property is used to define additional dependencies that the C standard library may have on other libraries.

See also: *zephyr_libc_link_libraries()*.

5.2.5 Variables

ACTIVE_BOARD_REVISION

The revision of the active board.

See *boards*.

APPLICATION_CONFIG_DIR

Root folder for application configuration.

See *configuration_files*.

APP_BUILD_VERSION

This variable is used in all contexts as the build version of the application.

By default, if the application is in a Git repository, its contents are generated from the output of the command `git describe --abbrev=12 --always` run in the application folder. This results in a string which contains the most recent tag, the number of commits since that tag, and the abbreviated commit hash.

This default behavior can also be overridden by setting the CMake variable `APP_BUILD_VERSION` to the desired string value. This can either be done in the `CMakeLists.txt` file, or on the command line when invoking CMake, e.g. via `cmake -DAPP_BUILD_VERSION="v3.3.0-18-g2c85d92" ...`

Input to `version` Custom application build version to use for this build, if set externally.

Output from `version` Final value of the build version of the application: either the value of `APP_BUILD_VERSION`, if set, or the value generated from the Git repository.

See *Application version management* for details; see also *BUILD_VERSION*.

APP_DIR

Source directory of the main application in a sysbuild build.

Set from the `-DAPP_DIR=<path>` build option or the matching environment variable. Sysbuild resolves the sysbuild configuration files and root lists relative to this directory.

See *Sysbuild (System build)*.

ARCH

Name of the architecture in use.

See *arch*.

ARCH_DIR

Directory containing the architecture implementation.

See *arch*.

ARCH_ROOT

List of directories containing architecture implementations. This variable is updated to include *ZEPHYR_BASE* appended.

See *arch*.

BOARD

Input to *boards* Board name, including any optional revision field, for example: `foo` or `foo@1.0.0`.

Output from *boards* Board name, without revision field.

See *boards*.

BOARD_DIR

Board directory with the implementation for selected board.

See *boards*.

BOARD_QUALIFIERS

Board qualifiers.

See *boards*.

BOARD_REVISION

Board revision.

See *boards*.

BOARD_ROOT

Input to *boards* List of directories containing board implementations.

Output from *boards* List of directories containing board implementations. This variable is updated to include *ZEPHYR_BASE* appended.

See *boards*.

BSIM_COMPONENTS_PATH

Path to the BabbleSim components source folder.

See *BabbleSim*.

BSIM_OUT_PATH

Path to the BabbleSim build output root path (under which libraries and binaries are kept).

See *BabbleSim*.

BUILD_VERSION

This variable is used in all contexts as the build version of the Zephyr kernel.

By default, if the Zephyr project is in a Git repository and the `git` tool is available, its contents are generated from the output of the command `git describe --abbrev=12 --always` run in the Zephyr folder. This results in a string which contains the most recent tag, the number of commits since that tag, and the abbreviated commit hash.

This default behavior can also be overridden by setting the CMake variable `BUILD_VERSION` to the desired string value. This can either be done in the `CMakeLists.txt` file, or on the command line when invoking CMake, e.g. via `cmake -DBUILD_VERSION="v3.3.0-18-g2c85d92" ...`

Input to `version` Custom Zephyr build version to use for this build, if set externally.

Output from `version` Final value of the build version of the Zephyr project: either the value of `BUILD_VERSION`, if already set, or the value generated from the Git repository.

See also *`APP_BUILD_VERSION`*.

CONF_FILE

List of Kconfig fragments.

See *`configuration_files`*.

DEFAULT_IMAGE

Name of the main application image in a sysbuild build.

This is the image added with `APP_TYPE MAIN`, and the one reported as the default domain in `domains.yaml`. Use it from a `sysbuild.cmake` file to refer to the main image without hardcoding its name.

See *`ExternalZephyrProject_Add()`*.

DTC_OVERLAY_FILE

List of devicetree overlay files.

See *`configuration_files`*.

DTS_EXTRA_CPPFLAGS

List of additional devicetree preprocessor defines.

See *`configuration_files`*.

EXTRA_CONF_FILE

List of additional Kconfig fragments.

See *`configuration_files`*.

EXTRA_DTC_OVERLAY_FILE

List of additional devicetree overlay files.

See *`configuration_files`*.

LLEXT_APPEND_FLAGS

Global list of compiler flags to be added to compiler invocations when compiling extensions. Target and compiler dependent, usually set in toolchain files.

See *add_llext_target()*.

LLEXT_REMOVE_FLAGS

Global list of regular expressions describing compiler flags to be removed from compiler invocations when compiling extensions. Target and compiler dependent, usually set in toolchain files.

See *add_llext_target()*.

MODULE_EXT_ROOT

List of directories containing module external roots, which hold the CMake and Kconfig glue code for Zephyr modules that do not carry it themselves.

See *sysbuild_root* for how sysbuild resolves this variable for image builds.

NORMALIZED_BOARD_QUALIFIERS

Board qualifiers in lower-case format where slashes have been replaced with underscores.

See *boards*.

NORMALIZED_BOARD_TARGET

Board target in lower-case format where slashes have been replaced with underscores.

See *boards*.

PYTHON_EXECUTABLE

Path to the Python executable that will be used by Zephyr. This is an alias to `Python3_EXECUTABLE` for backward compatibility.

See *python*.

PYTHON_MINIMUM_REQUIRED

The minimum required Python version (currently 3.12).

See *python*.

SB_APPLICATION_CONFIG_DIR

Directory searched for sysbuild's own configuration files.

When set, it takes precedence over *APPLICATION_CONFIG_DIR* for locating *SB_CONF_FILE*.

See *sysbuild_kconfig*.

SB_CONF_FILE

Sysbuild Kconfig configuration file, or list of files.

Defaults to `sysbuild.conf` in *APPLICATION_CONFIG_DIR*, if that file exists. Sysbuild is an opt-in feature, so the file is optional.

Settings in these files override the defaults of sysbuild's own Kconfig tree. Paths are resolved relative to *APP_DIR*.

See *sysbuild_kconfig*.

SB_EXTRA_CONF_FILE

Additional sysbuild Kconfig fragment files, applied on top of *SB_CONF_FILE*.

Paths are resolved relative to *APP_DIR*.

See *sysbuild_kconfig*.

SB_SNIPPET

List of snippets to apply to the sysbuild build itself.

Defaults to the value of `SNIPPET` when that is set and `SB_SNIPPET` is not.

See *Snippets*.

SCA_ROOT

List of directories containing static code analysis tool integrations.

See *sysbuild_root* for how sysbuild resolves this variable for image builds.

SNIPPET_ROOT

List of directories searched for snippet definitions.

See *sysbuild_root* for how sysbuild resolves this variable for image builds.

SOC_ROOT

List of directories containing SoC implementations.

See *sysbuild_root* for how sysbuild resolves this variable for image builds.

USE_CCACHE

This variable can be set to `0` to explicitly disable the use of ccache, even if it is installed on the system. If set to any other value (or unset) and ccache is found on the system, it will be used for compilation and linking.

WEST

Path to the west executable.

See *west*.

WEST_PYTHON

Path to the Python interpreter used by west.

See *west*.

WEST_TOPDIR

Top-level directory of the west workspace.

See *west*.

ZEPHYR_CURRENT_LIBRARY

The name of the current Zephyr library.

5.3 Devicetree

A *devicetree* is a hierarchical data structure primarily used to describe hardware. Zephyr uses devicetree in two main ways:

- to describe hardware to the *Device Driver Model*
- to provide that hardware's initial configuration

This page links to a high level guide on devicetree as well as reference material.

5.3.1 Devicetree Guide

The pages in this section are a high-level guide to using devicetree for Zephyr development.

Introduction to devicetree

> **Tip**
>
> This is a conceptual overview of devicetree and how Zephyr uses it. For step-by-step guides and examples, see *Devicetree HOWTOs*.

The following pages introduce general devicetree concepts and how they apply to Zephyr.

Scope and purpose A *devicetree* is primarily a hierarchical data structure that describes hardware. The Devicetree specification defines its source and binary representations.

Zephyr uses devicetree to describe:

- the hardware available on its boards
- that hardware's initial configuration

As such, devicetree is both a hardware description language and a configuration language for Zephyr. See *Devicetree versus Kconfig* for some comparisons between devicetree and Zephyr's other main configuration language, Kconfig.

There are two types of devicetree input files: *devicetree sources* and *devicetree bindings*. The sources contain the devicetree itself. The bindings describe its contents, including data types. The *build system* uses devicetree sources and bindings to produce a generated C header. The generated header's contents are abstracted by the `devicetree.h` API, which you can use to get information from your devicetree.

Here is a simplified view of the process:

All Zephyr and application source code files can include and use `devicetree.h`. This includes *device drivers*, *applications*, *tests*, the kernel, etc.

The API itself is based on C macros. The macro names all start with `DT_`. In general, if you see a macro that starts with `DT_` in a Zephyr source file, it's probably a `devicetree.h` macro. The generated C header contains macros that start with `DT_` as well; you might see those in compiler error messages. You always can tell a generated- from a non-generated macro: generated macros have some lowercased letters, while the `devicetree.h` macro names have all capital letters.

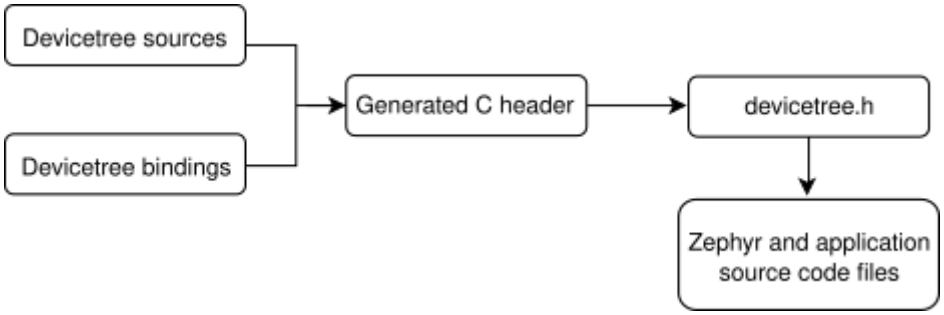


Fig. 1: Devicetree build flow

Syntax and structure As the name indicates, a devicetree is a tree. The human-readable text format for this tree is called DTS (for devicetree source), and is defined in the Devicetree specification.

This page's purpose is to introduce devicetree in a more gradual way than the specification. However, you may still need to refer to the specification to understand some detailed cases.

Contents

- *Example*
- *Nodes*
- *Properties*
- *Devicetrees reflect hardware*
- *Properties in practice*
- *Unit addresses*
- *Important properties*
- *Writing property values*
- *Aliases and chosen nodes*

Example Here is an example DTS file:

```
/dts-v1/;

/ {
        a-node {
                subnode_nodelabel: a-sub-node {
                        foo = <3>;
                };
        };
};
```

The `/dts-v1/;` line means the file's contents are in version 1 of the DTS syntax, which has replaced a now-obsolete "version 0".

Nodes Like any tree data structure, a devicetree has a hierarchy of *nodes*. The above tree has three nodes:

1. A root node: `/`
2. A node named `a-node`, which is a child of the root node

3. A node named `a-sub-node`, which is a child of `a-node`

Nodes can be assigned *node labels*, which are unique shorthands that refer to the labeled node. Above, `a-sub-node` has the node label `subnode_nodelabel`. A node can have zero, one, or multiple node labels. You can use node labels to refer to the node elsewhere in the devicetree.

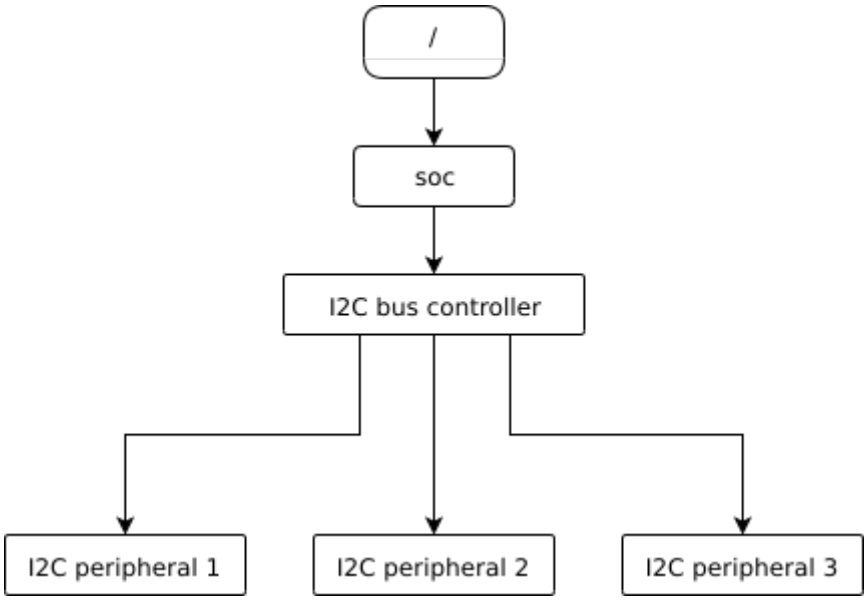
Devicetree nodes have *paths* identifying their locations in the tree. Like Unix file system paths, devicetree paths are strings separated by slashes (/), and the root node's path is a single slash: `/`. Otherwise, each node's path is formed by concatenating the node's ancestors' names with the node's own name, separated by slashes. For example, the full path to `a-sub-node` is `/a-node/a-sub-node`.

Properties Devicetree nodes can also have *properties*. Properties are name/value pairs. Property values can be any sequence of bytes. In some cases, the values are an array of what are called *cells*. A cell is just a 32-bit unsigned integer.

Node `a-sub-node` has a property named `foo`, whose value is a cell with value 3. The size and type of `foo`'s value are implied by the enclosing angle brackets (< and >) in the DTS.

See *Writing property values* below for more example property values.

Devicetrees reflect hardware In practice, devicetree nodes usually correspond to some hardware, and the node hierarchy reflects the hardware's physical layout. For example, let's consider a board with three I2C peripherals connected to an I2C bus controller on an SoC, like this:



Nodes corresponding to the I2C bus controller and each I2C peripheral would be present in the devicetree. Reflecting the hardware layout, the I2C peripheral nodes would be children of the bus controller node. Similar conventions exist for representing other types of hardware.

The DTS would look something like this:

```
/dts-v1/;

/ {
        soc {
                i2c-bus-controller {
                        i2c-peripheral-1 {
                        };
                        i2c-peripheral-2 {
                        };
```

(continues on next page)

(continued from previous page)

```
                        i2c-peripheral-3 {
                        };
                };
        };
};
```

Properties in practice In practice, properties usually describe or configure the hardware the node represents. For example, an I2C peripheral's node has a property whose value is the peripheral's address on the bus.

Here's a tree representing the same example, but with real-world node names and properties you might see when working with I2C devices.

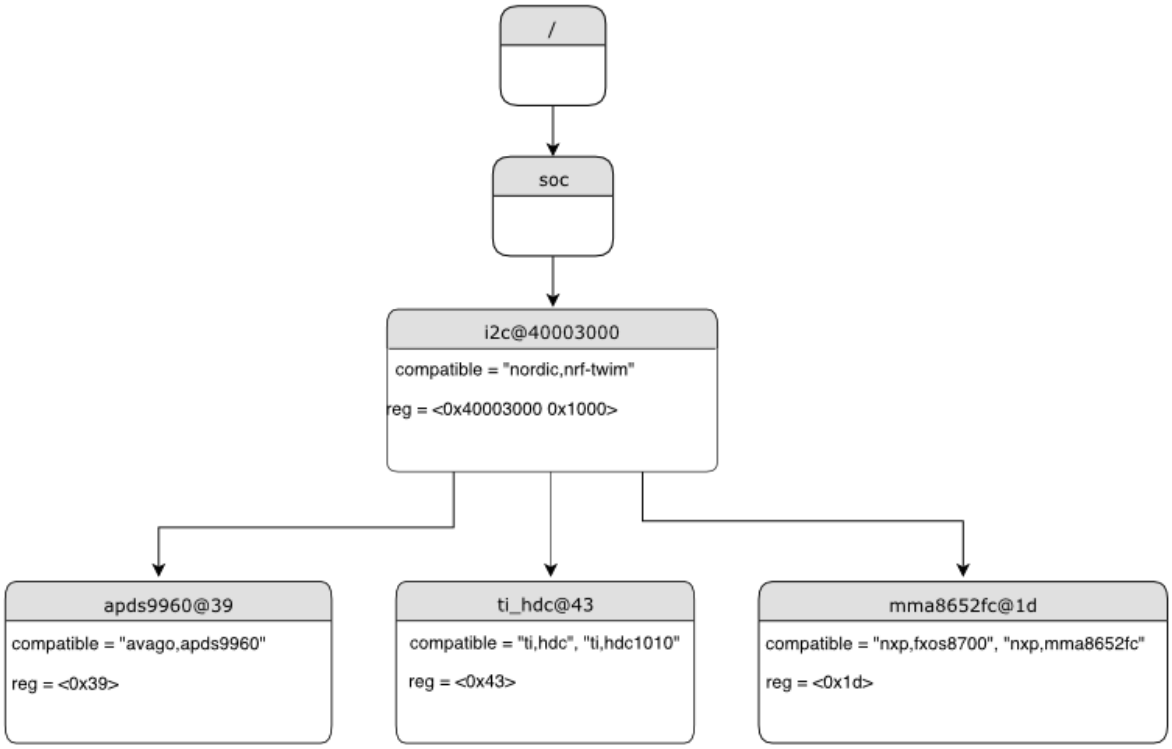


Fig. 2: I2C devicetree example with real-world names and properties. Node names are at the top of each node with a gray background. Properties are shown as "name=value" lines.

This is the corresponding DTS:

```
/dts-v1/;

/ {
        soc {
                i2c@40003000 {
                        compatible = "nordic,nrf-twim";
                        reg = <0x40003000 0x1000>;

                        apds9960@39 {
                                compatible = "avago,apds9960";
                                reg = <0x39>;
                        };
                        ti_hdc@43 {
```

(continues on next page)

(continued from previous page)

```
                                compatible = "ti,hdc", "ti,hdc1010";
                                reg = <0x43>;
                        };
                        mma8652fc@1d {
                                compatible = "nxp,fxos8700", "nxp,mma8652fc";
                                reg = <0x1d>;
                        };
                };
        };
};
```

Unit addresses In addition to showing more real-world names and properties, the above example introduces a new devicetree concept: unit addresses. Unit addresses are the parts of node names after an "at" sign (`@`), like `40003000` in `i2c@40003000`, or `39` in `apds9960@39`. Unit addresses are optional: the `soc` node does not have one.

In devicetree, unit addresses give a node's address in the address space of its parent node. Here are some example unit addresses for different types of hardware.

Memory-mapped peripherals
The peripheral's register map base address. For example, the node named `i2c@40003000` represents an I2C controller whose register map base address is 0x40003000.

I2C peripherals
The peripheral's address on the I2C bus. For example, the child node `apds9960@39` of the I2C controller in the previous section has I2C address 0x39.

SPI peripherals
An index representing the peripheral's chip select line number. (If there is no chip select line, 0 is used.)

Memory
The physical start address. For example, a node named `memory@2000000` represents RAM starting at physical address 0x2000000.

Memory-mapped flash
Like RAM, the physical start address. For example, a node named `flash@8000000` represents a flash device whose physical start address is 0x8000000.

Fixed flash partitions
This applies when the devicetree is used to store a flash partition table. The unit address is the partition's start offset within the flash memory. For example, take this flash device and its partitions:

```
flash@8000000 {
    /* ... */
    partitions {
            partition@0 { /* ... */ };
            partition@20000 {  /* ... */ };
            /* ... */
    };
};
```

The node named `partition@0` has offset 0 from the start of its flash device, so its base address is 0x8000000. Similarly, the base address of the node named `partition@20000` is 0x8020000.

Important properties The devicetree specification defines several standard properties. Some of the most important ones are:

compatible

The name of the hardware device the node represents.

The recommended format is `"vendor,device"`, like `"avago,apds9960"`, or a sequence of these, like `"ti,hdc", "ti,hdc1010"`. The `vendor` part is an abbreviated name of the vendor. The file dts/bindings/vendor-prefixes.txt contains a list of commonly accepted `vendor` names. The `device` part is usually taken from the datasheet.

It is also sometimes a value like `gpio-keys`, `mmio-sram`, or `fixed-clock` when the hardware's behavior is generic.

The build system uses the compatible property to find the right *bindings* for the node. Device drivers use `devicetree.h` to find nodes with relevant compatibles, in order to determine the available hardware to manage.

The `compatible` property can have multiple values. Additional values are useful when the device is a specific instance of a more general family, to allow the system to match from most- to least-specific device drivers.

Within Zephyr's bindings syntax, this property has type `string-array`.

reg

Information used to address the device. The value is specific to the device (i.e. is different depending on the compatible property).

The `reg` property is a sequence of `(address, length)` pairs. Each pair is called a "register block". Values are conventionally written in hex.

Here are some common patterns:

- Devices accessed via memory-mapped I/O registers (like `i2c@40003000`): `address` is usually the base address of the I/O register space, and `length` is the number of bytes occupied by the registers.
- I2C devices (like `apds9960@39` and its siblings): `address` is a slave address on the I2C bus. There is no `length` value.
- SPI devices: `address` is a chip select line number; there is no `length`.

You may notice some similarities between the `reg` property and common unit addresses described above. This is not a coincidence. The `reg` property can be seen as a more detailed view of the addressable resources within a device than its unit address.

status

A string which describes whether the node is enabled.

The devicetree specification allows this property to have values `"okay"`, `"disabled"`, `"reserved"`, `"fail"`, and `"fail-sss"`. Zephyr treats any value other than `"okay"` as disabled. In particular, `"reserved"` is used to document that a node exists but is enabled elsewhere (e.g. by another core or domain in a multi-domain application); it is handled the same as `"disabled"` by `edtlib`. Use of the remaining values (`"fail"` and `"fail-sss"`) is not currently used by Zephyr.

A node is considered enabled if its status property is either `"okay"` or not defined (i.e. does not exist in the devicetree source). Nodes with status `"disabled"` are explicitly disabled. Devicetree nodes which correspond to physical devices must be enabled for the corresponding `struct device` in the Zephyr driver model to be allocated and initialized.

Note that a child node is not implicitly disabled when its parent node is disabled, i.e. child nodes should be disabled explicitly if desired.

interrupts

Information about interrupts generated by the device, encoded as an array of one or more *interrupt specifiers*. Each interrupt specifier has some number of cells. See section 2.4, *Interrupts and Interrupt Mapping*, in the Devicetree Specification release v0.3 for more details.

Zephyr's devicetree bindings language lets you give a name to each cell in an interrupt specifier.

> **Note**
>
> Earlier versions of Zephyr made frequent use of the `label` property, which is distinct from the standard *node label*. Use of the label *property* in new devicetree bindings, as well as use of the `DT_LABEL` macro in new code, are actively discouraged. Label properties continue to persist for historical reasons in some existing bindings and overlays, but should not be used in new bindings or device implementations.

Writing property values This section describes how to write property values in DTS format. The property types in the table below are described in detail in *Devicetree bindings*.

Some specifics are skipped in the interest of keeping things simple; if you're curious about details, see the devicetree specification.

Property type	How to write	Example
string	Double quoted	`a-string = "hello, world!";`
int	between angle brackets (< and >)	`an-int = <1>;`
boolean	for `true`, with no value (for `false`, use `/delete-property/`)	`my-true-boolean;`
array	between angle brackets (< and >), separated by spaces	`foo = <0xdeadbeef 1234 0>;`
uint8-array	in hexadecimal *without* leading `0x`, between square brackets ([and]).	`a-byte-array = [00 01 ab];`
string-array	separated by commas	`a-string-array = "string one", "string two", "string three";`
phandle	between angle brackets (< and >)	`a-phandle = <&mynode>;`
phandles	between angle brackets (< and >), separated by spaces	`some-phandles = <&mynode0 &mynode1 &mynode2>;`
phandle-array	between angle brackets (< and >), separated by spaces	`a-phandle-array = <&mynode0 1 2>, <&mynode1 3 4>;`

Additional notes on the above:

- The values in the `phandle`, `phandles`, and `phandle-array` types are described further in *Phandles*
- Boolean properties are true if present. They should not have a value. A boolean property is only false if it is completely missing in the DTS.
- The `foo` property value above has three *cells* with values 0xdeadbeef, 1234, and 0, in that order. Note that hexadecimal and decimal numbers are allowed and can be intermixed. Since Zephyr transforms DTS to C sources, it is not necessary to specify the endianness of an individual cell here.
- 64-bit integers are written as two 32-bit cells in big-endian order. The value 0xaaaa0000bbbb1111 would be written `<0xaaaa0000 0xbbbb1111>`.
- The `a-byte-array` property value is the three bytes 0x00, 0x01, and 0xab, in that order.
- Parentheses, arithmetic operators, and bitwise operators are allowed. The `bar` property contains a single cell with value 64:

```
bar = <(2 * (1 << 5))>;
```

Note that the entire expression must be parenthesized.

- A cell whose parenthesized expression evaluates to a negative value, such as `<(-1)>` or `<(4 - 6)>`, is preserved as a signed value for `int` and `array` typed properties. Cells written as positive decimal or hexadecimal literals (including `0xffffffff`), or whose expression evaluates to a non-negative value, remain unsigned.
- Property values refer to other nodes in the devicetree by their *phandles*. You can write a phandle using `&foo`, where `foo` is a *node label*. Here is an example devicetree fragment:

```
foo: device@0 { };
device@1 {
        sibling = <&foo 1 2>;
};
```

The `sibling` property of node `device@1` contains three cells, in this order:

1. The `device@0` node's phandle, which is written here as `&foo` since the `device@0` node has a node label `foo`
2. The value 1
3. The value 2

In the devicetree, a phandle value is a cell – which again is just a 32-bit unsigned int. However, the Zephyr devicetree API generally exposes these values as *node identifiers*. Node identifiers are covered in more detail in *Devicetree access from C/C++*.

- Array and similar type property values can be split into several <> blocks, like this:

```
foo = <1 2>, <3 4>;                          // Okay for 'type: array'
foo = <&label1 &label2>, <&label3 &label4>; // Okay for 'type: phandles'
foo = <&label1 1 2>, <&label2 3 4>;          // Okay for 'type: phandle-array'
```

This is recommended for readability when possible if the value can be logically grouped into blocks of sub-values.

Aliases and chosen nodes There are two additional ways beyond *node labels* to refer to a particular node without specifying its entire path: by alias, or by chosen node.

Here is an example devicetree which uses both:

```
/dts-v1/;

/ {
     chosen {
             zephyr,console = &uart0;
     };

     aliases {
             my-uart = &uart0;
     };

     soc {
             uart0: serial@12340000 {
                     ...
             };
     };
};
```

The `/aliases` and `/chosen` nodes do not refer to an actual hardware device. Their purpose is to specify other nodes in the devicetree.

Above, `my-uart` is an alias for the node with path `/soc/serial@12340000`. Using its node label `uart0`, the same node is set as the value of the chosen `zephyr,console` node.

Zephyr sample applications sometimes use aliases to allow overriding the particular hardware device used by the application in a generic way. For example, blinky uses this to abstract the LED to blink via the `led0` alias.

The `/chosen` node's properties are used to configure system- or subsystem-wide values. See *Chosen nodes* for more information.

Input and output files This section describes the input and output files shown in the figure in *Scope and purpose* in more detail.

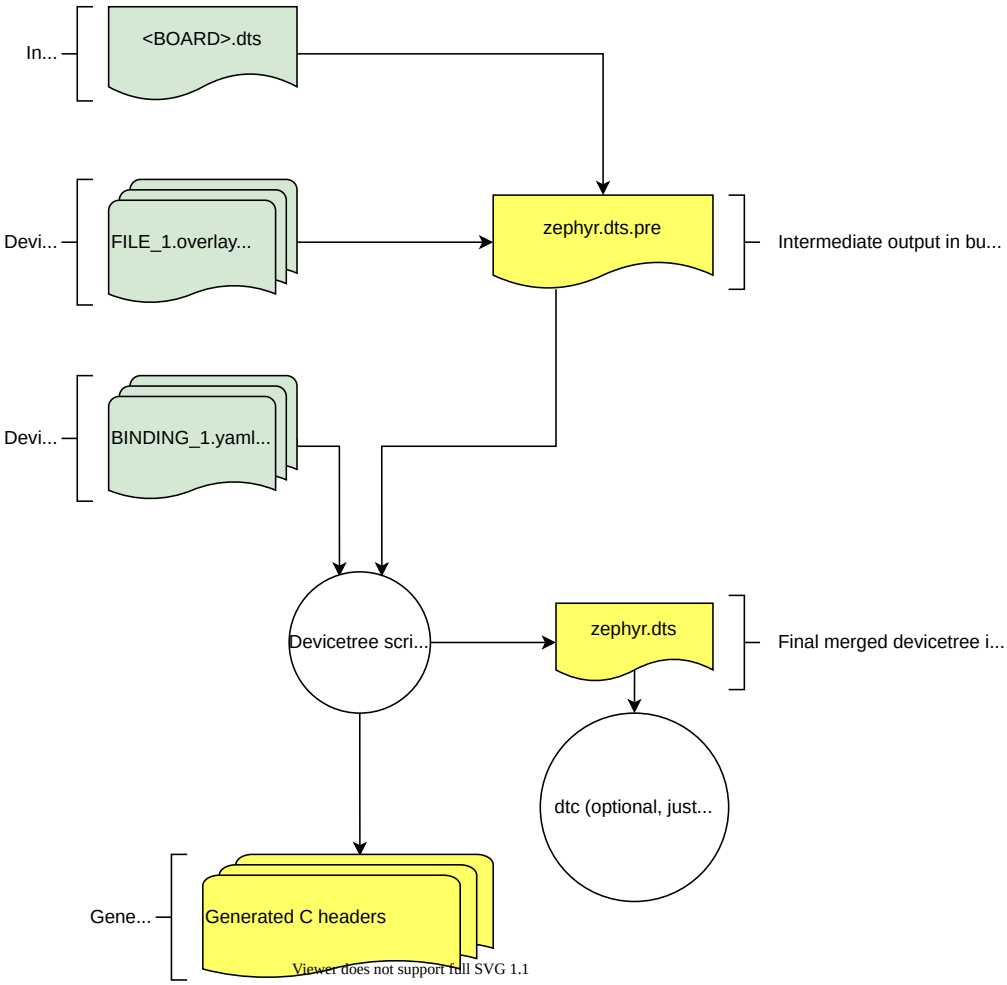


Fig. 3: Devicetree input (green) and output (yellow) files

Input files There are four types of devicetree input files:

- sources (`.dts`)
- includes (`.dtsi`)
- overlays (`.overlay`)
- bindings (`.yaml`)

The devicetree files inside the `zephyr` directory look like this:

```
boards/<ARCH>/<BOARD>/<BOARD>.dts
dts/common/skeleton.dtsi
dts/<ARCH>/.../<SOC>.dtsi
dts/bindings/.../binding.yaml
```

Generally speaking, every supported board has a `BOARD.dts` file describing its hardware. For example, the `reel_board` has boards/phytec/reel_board/reel_board.dts.

`BOARD.dts` includes one or more `.dtsi` files. These `.dtsi` files describe the CPU or system-on-chip Zephyr runs on, perhaps by including other `.dtsi` files. They can also describe other common hardware features shared by multiple boards. In addition to these includes, `BOARD.dts` also describes the board's specific hardware.

The `dts/common` directory contains `skeleton.dtsi`, a minimal include file for defining a complete devicetree. Architecture-specific subdirectories (`dts/<ARCH>`) contain `.dtsi` files for CPUs or SoCs which extend `skeleton.dtsi`.

The C preprocessor is run on all devicetree files to expand macro references, and includes are generally done with `#include <filename>` directives, even though DTS has a `/include/ "<filename>"` syntax.

`BOARD.dts` can be extended or modified using *overlays*. Overlays are also DTS files; the `.overlay` extension is just a convention which makes their purpose clear. Overlays adapt the base devicetree for different purposes:

- Zephyr applications can use overlays to enable a peripheral that is disabled by default, select a sensor on the board for an application specific purpose, etc. Along with *Configuration System (Kconfig)*, this makes it possible to reconfigure the kernel and device drivers without modifying source code.
- Overlays are also used when defining *Shields*.

The build system automatically picks up `.overlay` files stored in certain locations. It is also possible to explicitly list the overlays to include, via the **`DTC_OVERLAY_FILE`** CMake variable. See *Set devicetree overlays* for details.

The build system combines `BOARD.dts` and any `.overlay` files by concatenating them, with the overlays put last. This relies on DTS syntax which allows merging overlapping definitions of nodes in the devicetree. See *Example: FRDM-K64F and Hexiwear K64* for an example of how this works (in the context of `.dtsi` files, but the principle is the same for overlays). Putting the contents of the `.overlay` files last allows them to override `BOARD.dts`.

Devicetree bindings (which are YAML files) are essentially glue. They describe the contents of devicetree sources, includes, and overlays in a way that allows the build system to generate C macros usable by device drivers and applications. The `dts/bindings` directory contains bindings.

Scripts and tools The following libraries and scripts, located in scripts/dts/, create output files from input files. Their sources have extensive documentation.

dtlib.py
: A low-level DTS parsing library.

edtlib.py
: A library layered on top of dtlib that uses bindings to interpret properties and give a higher-level view of the devicetree. Uses dtlib to do the DTS parsing.

gen_defines.py
: A script that uses edtlib to generate C preprocessor macros from the devicetree and bindings.

In addition to these, the standard `dtc` (devicetree compiler) tool is run on the final devicetree if it is installed on your system. This is just to catch errors or warnings. The output is unused.

Boards may need to pass `dtc` additional flags, e.g. for warning suppression. Board directories can contain a file named `pre_dt_board.cmake` which configures these extra flags, like this:

```
list(APPEND EXTRA_DTC_FLAGS "-Wno-simple_bus_reg")
```

Shield directories can contain a file named `pre_dt_shield.cmake` which has the same functionality as the aforementioned `pre_dt_board.cmake`.

Output files These are created in your application's build directory.

> **Warning**
>
> Don't include the header files directly. *Devicetree access from C/C++* explains what to do instead.

`<build>/zephyr/zephyr.dts.pre`
: The preprocessed DTS source. This is an intermediate output file, which is input to `gen_defines.py` and used to create `zephyr.dts` and `devicetree_generated.h`.

`<build>/zephyr/include/generated/zephyr/devicetree_generated.h`
: The generated macros and additional comments describing the devicetree. Included by `devicetree.h`.

`<build>/zephyr/zephyr.dts`
: The final merged devicetree. This file is output by `gen_defines.py`. It is useful for debugging any issues. If the devicetree compiler `dtc` is installed, it is also run on this file, to catch any additional warnings or errors.

Design goals

Zephyr's use of devicetree has evolved significantly over time, and further changes are expected. The following are the general design goals, along with specific examples about how they impact Zephyr's source code, and areas where more work remains to be done.

Single source for hardware information Zephyr's built-in device drivers and sample applications shall obtain configurable hardware descriptions from devicetree.

Examples

- New device drivers shall use devicetree APIs to determine which *devices to create*.
- In-tree sample applications shall use *aliases* to determine which of multiple possible generic devices of a given type will be used in the current build. For example, the blinky sample uses this to determine the LED to blink.
- Boot-time pin muxing and pin control for new SoCs shall be accomplished via a devicetree-based pinctrl driver

Example remaining work

- Zephyr's *Test Runner (Twister)* currently use `board.yaml` files to determine the hardware supported by a board. This should be obtained from devicetree instead.
- Legacy device drivers currently use Kconfig to determine which instances of a particular compatible are enabled. This can and should be done with devicetree overlays instead.
- Board-level documentation still contains tables of hardware support which are generated and maintained by hand. This can and should be obtained from the board level devicetree instead.

- Runtime determination of `struct device` relationships should be done using information obtained from devicetree, e.g. for device power management.

Source compatibility with other operating systems Zephyr's devicetree tooling is based on a generic layer which is interoperable with other devicetree users, such as the Linux kernel.

Zephyr's binding language *semantics* can support Zephyr-specific attributes, but shall not express Zephyr-specific relationships.

Examples

- Zephyr's devicetree source parser, *dtlib.py*, is source-compatible with other tools like dtc in both directions: `dtlib.py` can parse `dtc` output, and `dtc` can parse `dtlib.py` output.
- Zephyr's "extended dtlib" library, `edtlib.py`, shall not include Zephyr-specific features. Its purpose is to provide a higher-level view of the devicetree for common elements like interrupts and buses.

  Only the high-level `gen_defines.py` script, which is built on top of `edtlib.py`, contains Zephyr-specific knowledge and features.

Example remaining work

- Zephyr has a custom *Devicetree bindings* language *syntax*. While Linux's dtschema does not yet meet Zephyr's needs, we should try to follow what it is capable of representing in Zephyr's own bindings.
- Due to inflexibility in the bindings language, Zephyr cannot support the full set of bindings supported by Linux.
- Devicetree source sharing between Zephyr and Linux is not done.

Modeling hardware exposed through multiple Zephyr APIs When a hardware block can be used through different Zephyr APIs, devicetree should keep hardware identity separate from the way software drivers and subsystems use each hardware instance.

Guidelines

- Keep the `compatible` property focused on the device programming model. Avoid changing it only to route software API selection. A different `compatible` is appropriate when the hardware interfaces are architecturally distinct, for example because they use separate register blocks.
- Use `/chosen` for singleton system-wide selection. If the system must select exactly one instance for a global function, such as the system timer, console, or entropy source, represent that choice with a platform-agnostic chosen property, even if other identical hardware instances are used by other APIs. See also *Zephyr-specific chosen nodes*.
- Place the selection at the right integration level. If it is a silicon-level fixed attribute, describe it in the SoC `.dtsi`. If it varies by board, product, or application, describe it in the board `.dts` or an overlay.

If different uses of the hardware device have different required properties, child nodes, or bus semantics, then they do not have to share one binding schema. In that case, use different `compatible` values or model the device as a multifunction one.

Examples **Singleton selection with `/chosen`**

The system timer is a singleton function. The `zephyr,system-timer` chosen property selects which hardware timer instance provides it, while other identical instances remain available for other APIs.

For example, two identical LPTMR timer instances can share the same `compatible` in the hardware description:

```
lptmr1: timer@44300000 {
    compatible = "nxp,lptmr";
    reg = <0x44300000 0x1000>;
    /* ... */
};

lptmr2: timer@424d0000 {
    compatible = "nxp,lptmr";
    reg = <0x424d0000 0x1000>;
    /* ... */
};
```

An integration layer can then select which one provides the singleton system timer function:

```
/ {
    chosen {
        zephyr,system-timer = &lptmr1;
    };
};
```

The system-timer driver binds to the `/chosen` node. The counter driver skips that instance and uses the remaining one.

Devicetree bindings

A devicetree on its own is only half the story for describing hardware, as it is a relatively unstructured format. *Devicetree bindings* provide the other half.

A devicetree binding declares requirements on the contents of nodes, and provides semantic information about the contents of valid nodes. Zephyr devicetree bindings are YAML files in a custom format (Zephyr does not use the dt-schema tools used by the Linux kernel).

These pages introduce bindings, describe what they do, note where they are found, and explain their data format.

> **Note**
>
> See the *Bindings index* for reference information on bindings built in to Zephyr.

> **Note**
>
> For a detailed syntax reference, see *Devicetree bindings syntax*.

Introduction to Devicetree Bindings Devicetree nodes are matched to bindings using their *compatible properties*.

During the *Configuration Phase*, the build system tries to match each node in the devicetree to a binding file. When this succeeds, the build system uses the information in the binding file both when validating the node's contents and when generating macros for the node.

A simple example Here is an example devicetree node:

```
/* Node in a DTS file */
bar-device {
     compatible = "foo-company,bar-device";
     num-foos = <3>;
};
```

Here is a minimal binding file which matches the node:

```
# A YAML binding matching the node

compatible: "foo-company,bar-device"

properties:
  num-foos:
    type: int
    required: true
```

The build system matches the `bar-device` node to its YAML binding because the node's `compatible` property matches the binding's `compatible:` line.

What the build system does with bindings The build system uses bindings both to validate devicetree nodes and to convert the devicetree's contents into the generated *devicetree_generated.h* header file.

For example, the build system would use the above binding to check that the required `num-foos` property is present in the `bar-device` node, and that its value, `<3>`, has the correct type.

The build system will then generate a macro for the `bar-device` node's `num-foos` property, which will expand to the integer literal `3`. This macro lets you get the value of the property in C code using the API which is discussed later in this guide in *Devicetree access from C/C++*.

For another example, the following node would cause a build error, because it has no `num-foos` property, and this property is marked required in the binding:

```
bad-node {
     compatible = "foo-company,bar-device";
};
```

Other ways nodes are matched to bindings If a node has more than one string in its `compatible` property, the build system looks for compatible bindings in the listed order and uses the first match.

Take this node as an example:

```
baz-device {
     compatible = "foo-company,baz-device", "generic-baz-device";
};
```

The `baz-device` node would get matched to a binding with a `compatible: "generic-baz-device"` line if the build system can't find a binding with a `compatible: "foo-company,baz-device"` line.

Nodes without compatible properties can be matched to bindings associated with their parent nodes. These are called "child bindings". If a node describes hardware on a bus, like I2C or SPI, then the bus type is also taken into account when matching nodes to bindings. (See *On-bus* for details).

See *The /zephyr,user node* for information about a special node that doesn't require any binding.

Where bindings are located Binding file names usually match their `compatible:` lines. For example, the above example binding would be named `foo-company,bar-device.yaml` by convention.

The build system looks for bindings in `dts/bindings` subdirectories of the following places:

- the zephyr repository
- your *application source directory*
- your *board directory*
- any *shield directories*
- any directories manually included in the *DTS_ROOT* CMake variable
- any *module* that defines a `dts_root` in its *Build settings*

The build system will consider any YAML file in any of these, including in any subdirectories, when matching nodes to bindings. A file is considered YAML if its name ends with `.yaml` or `.yml`.

Warning

The binding files must be located somewhere inside the `dts/bindings` subdirectory of the above places.

For example, if `my-app` is your application directory, then you must place application-specific bindings inside `my-app/dts/bindings`. So `my-app/dts/bindings/serial/my-company,my-serial-port.yaml` would be found, but `my-app/my-company,my-serial-port.yaml` would be ignored.

Devicetree bindings syntax This page documents the syntax of Zephyr's bindings format. Zephyr bindings files are YAML files. A *simple example* was given in the introduction page.

Contents

- *Top level keys*
- *Title*
- *Description*
- *Compatible*
- *Properties*
  - *Property entry syntax*
  - *Example property definitions*
  - *required*
  - *type*
  - *deprecated*
  - *default*
  - *enum*
  - *min and max*
  - *min-len and max-len*
  - *const*

- *specifier-space*
- *dependency-mode*
  - *Possible values*
  - *Using dependency-mode*
  - *Default behavior*
- *Child-binding*
- *Bus*
- *On-bus*
- *Class*
- *Examples*
- *Specifier cell names (*-cells)*
- *Include*
- *Nexus nodes and maps*

Top level keys The top level of a bindings file maps keys to values. The top-level keys look like this:

```
# When the description text is too long, this field can
# be used to improve readability, e.g.:
#
# title: Binding the device's hardware model.
#
# description |
#   A piece of content with 20 lines.
#   ...
title: Concise title for the long description [optional]

# A high level description of the device the binding applies to:
description: |
   This is the Vendomatic company's foo-device.

   Descriptions which span multiple lines (like this) are OK,
   and are encouraged for complex bindings.

   See https://yaml-multiline.info/ for formatting help.

# You can include definitions from other bindings using this syntax:
include: other.yaml

# Used to match nodes to this binding:
compatible: "manufacturer,foo-device"

properties:
  # Requirements for and descriptions of the properties that this
  # binding's nodes need to satisfy go here.

child-binding:
  # You can constrain the children of the nodes matching this binding
  # using this key.
```

(continues on next page)

(continued from previous page)

```
# If the node describes bus hardware, like an SPI bus controller
# on an SoC, use 'bus:' to say which one, like this:
bus: spi

# If the node instead appears as a device on a bus, like an external
# SPI memory chip, use 'on-bus:' to say what type of bus, like this.
# Like 'compatible', this key also influences the way nodes match
# bindings.
on-bus: spi

examples:
  # You can put a sample node here showing how to use the binding.
  # - |
  #  ...
  # or
  # - ...

foo-cells:
  # "Specifier" cell names for the 'foo' domain go here; example 'foo'
  # values are 'gpio', 'pwm', and 'dma'. See below for more information.
```

These keys are explained in the following sections.

Title An *optional*, short description of the bound device, typically the hardware model. It should typically be of the format "Vendor Family Model". If acronyms are used, they should be spelled out in parentheses. The naming should stay as close to the vendor datasheet as possible.

Titles should not exceed 100 characters. The description field should be used for longer descriptions. The words "binding", "schema" or "driver" should not be used in the title, everything is a binding.

```
title: Acme Foo UART (Universal Asynchronous Receiver/Transmitter)
```

Description A free-form description of node hardware goes here. You can put links to datasheets or example nodes or properties as well.

Compatible This key is used to match nodes to this binding as described in *Introduction to Devicetree Bindings*. It should look like this in a binding file:

```
# Note the comma-separated vendor prefix and device name
compatible: "manufacturer,device"
```

This devicetree node would match the above binding:

```
device {
     compatible = "manufacturer,device";
};
```

Assuming no binding has `compatible: "manufacturer,device-v2"`, it would also match this node:

```
device-2 {
    compatible = "manufacturer,device-v2", "manufacturer,device";
};
```

Each node's `compatible` property is tried in order. The *bindings* are uniquely identified by a pair of (*compatible*, *on-bus*), where the *on-bus* may be unspecified. A specified *on-bus* takes precedence over unspecified. The first matching binding is used.

For the following device:

```
spi-bus {
        device-3 {
                compatible = "manufacturer,device";
        };
};
```

The following two bindings can coexist and would match in the following order:

`manufacturer,device-spi.yaml`

```
compatible: "manufacturer,device"
on-bus: spi
```

`manufacturer,device.yaml`

```
compatible: "manufacturer,device"
```

The following binding can coexist but would not match.

`manufacturer,device-i2c.yaml`

```
compatible: "manufacturer,device"
on-bus: i2c
```

If more than one matching binding for a compatible is found, an error is raised.

The `manufacturer` prefix identifies the device vendor. See dts/bindings/vendor-prefixes.txt for a list of accepted vendor prefixes. The `device` part is usually from the datasheet.

Some bindings apply to a generic class of devices which do not have a specific vendor. In these cases, there is no vendor prefix. One example is the `gpio-leds` compatible which is commonly used to describe board LEDs connected to GPIOs.

Properties The `properties:` key describes properties that nodes which match the binding contain. For example, a binding for a UART peripheral might look something like this:

```
compatible: "manufacturer,serial"

properties:
  reg:
    type: array
    description: UART peripheral MMIO register space
    required: true
  current-speed:
    type: int
    description: current baud rate
    required: true
```

In this example, a node with compatible `"manufacturer,serial"` must contain a property named `current-speed`. The property's value must be a single integer. Similarly, the node must contain a `reg` property.

The build system uses bindings to generate C macros for devicetree properties that appear in DTS files. You can read more about how to get property values in source code from these macros in *Devicetree access from C/C++*. Generally speaking, the build system only generates macros for

properties listed in the `properties:` key for the matching binding. Properties not mentioned in the binding are generally ignored by the build system.

The one exception is that the build system will always generate macros for standard properties, like *reg*, whose meaning is defined by the devicetree specification. This happens regardless of whether the node has a matching binding or not.

Property entry syntax Property entries in `properties:` are written in this syntax:

```
<property name>:
  required: <true | false>
  type: <string | int | boolean | array | uint8-array | string-array |
         phandle | phandles | phandle-array | path | compound>
  deprecated: <true | false>
  default: <default>
  description: <description of the property>
  enum:
    - <item1>
    - <item2>
    ...
    - <itemN>
  const: <string | int | array | uint8-array | string-array>
  min: <int>
  max: <int>
  min-len: <int>
  max-len: <int>
  specifier-space: <space-name>
  dependency-mode: <normal | reverse | ignore | child-ignore>
```

Example property definitions Here are some more examples.

```
properties:
    # Describes a property like 'current-speed = <115200>;'. We pretend that
    # it's obligatory for the example node and set 'required: true'.
    current-speed:
        type: int
        required: true
        description: Initial baud rate for bar-device

    # Describes an optional property like 'keys = "foo", "bar";'
    keys:
        type: string-array
        description: Keys for bar-device

    # Describes an optional property like 'maximum-speed = "full-speed";'
    # the enum specifies known values that the string property may take
    maximum-speed:
        type: string
        description: Configures USB controllers to work up to a specific speed.
        enum:
           - "low-speed"
           - "full-speed"
           - "high-speed"
           - "super-speed"

    # Describes an optional property like 'resolution = <16>;'
    # the enum specifies known values that the int property may take
```

(continues on next page)

(continued from previous page)

```
    resolution:
      type: int
      enum:
       - 8
       - 16
       - 24
       - 32

    # Describes a required property '#address-cells = <1>';  the const
    # specifies that the value for the property is expected to be the value 1
    "#address-cells":
        type: int
        required: true
        const: 1

    int-with-default:
        type: int
        default: 123
        description: Value for int register, default is power-up configuration.

    array-with-default:
        type: array
        default: [1, 2, 3] # Same as 'array-with-default = <1 2 3>'

    string-with-default:
        type: string
        default: "foo"

    string-array-with-default:
        type: string-array
        default: ["foo", "bar"] # Same as 'string-array-with-default = "foo",
↪"bar"'

    uint8-array-with-default:
        type: uint8-array
        default: [0x12, 0x34] # Same as 'uint8-array-with-default = [12 34]'
```

required Adding `required: true` to a property definition will fail the build if a node matches the binding, but does not contain the property.

The default setting is `required: false`; that is, properties are optional by default. Using `required: false` is therefore redundant and strongly discouraged.

type The type of a property constrains its values. The following types are available. See *Writing property values* for more details about writing values of each type in a DTS file. See *Phandles* for more information about the `phandle*` type properties.

Type	Description	Example in DTS
`string`	exactly one string	`status = "disabled";`
`int`	exactly one 32-bit value (cell)	`current-speed = <115200>;`
`boolean`	flags that don't take a value when true, and are absent if false	`hw-flow-control;`
`array`	zero or more 32-bit values (cells)	`offsets = <0x100 0x200 0x300>;`
`uint8-array`	zero or more bytes, in hex ('bytestring' in the Devicetree specification)	`local-mac-address = [de ad be ef 12 34];`
`string-array`	zero or more strings	`dma-names = "tx", "rx";`
`phandle`	exactly one phandle	`interrupt-parent = <&gic>;`
`phandles`	zero or more phandles	`pinctrl-0 = <&usart2_tx_pd5 &usart2_rx_pd6>;`
`phandle-array`	a list of phandles and 32-bit cells (usually specifiers)	`dmas = <&dma0 2>, <&dma0 3>;`
`path`	a path to a node as a phandle path reference or path string	`zephyr,bt-c2h-uart = &uart0;` or `foo = "/path/to/some/node";`
`compound`	a catch-all for more complex types (no macros will be generated)	`foo = <&label>, [01 02];`

deprecated A property with `deprecated: true` indicates to both the user and the tooling that the property is meant to be phased out.

The tooling will report a warning if the devicetree includes the property that is flagged as deprecated. (This warning is upgraded to an error in the *Test Runner (Twister)* for upstream pull requests.)

The default setting is `deprecated: false`. Using `deprecated: false` is therefore redundant and strongly discouraged.

default The optional `default:` setting gives a value that will be used if the property is missing from the devicetree node.

For example, with this binding fragment:

```
properties:
  foo:
    type: int
    default: 3
```

If property `foo` is missing in a matching node, then the output will be as if `foo = <3>;` had appeared in the DTS (except YAML data types are used for the default value).

Note that a binding which declares both `default:` and `required: true` for the same property will produce a warning, since the two settings are redundant: `required: true` already fails the build when the property is missing, so the default can never apply. A binding may still override an inherited `default:` with `required: true` to force an explicit value; this is well defined and is not reported. A binding which is only ever reached through `include:` is not loaded on its own during a build, so the warning for it is seen when the documentation is generated rather than when an application is built.

For rules related to `default` in upstream Zephyr bindings, see *Rules for default values*.

See *Example property definitions* for examples. Putting `default:` on any property type besides those used in *Example property definitions* will raise an error.

enum The `enum:` line is followed by a list of values the property may contain. If a property value in DTS is not in the `enum:` list in the binding, an error is raised. See *Example property definitions* for examples.

min and max The `min:` and `max:` keys constrain the range of valid values for properties with `type: int` or `type: array`. If a property value in DTS is outside the `[min, max]` range, an error is raised.

Both keys are optional and independent; you may specify just `min:`, just `max:`, or both. They cannot be combined with `enum:` on the same property.

For `type: array`, each element of the array is checked against the range.

Example:

```
properties:
  # A brightness percentage between 0 and 100
  brightness:
    type: int
    min: 0
    max: 100
    description: LED brightness as a percentage

  # A timeout in milliseconds, with a minimum of 1 ms (no upper bound)
  timeout-ms:
    type: int
    min: 1
    description: Timeout in milliseconds
```

min-len and max-len The `min-len:` and `max-len:` keys constrain the number of elements (length) for array-type properties (`array`, `uint8-array`, `string-array`, `phandles`, and `phandle-array`). If the length of a property value in DTS is outside the `[min-len, max-len]` range, an error is raised.

Both keys are optional and independent; you may specify just `min-len:`, just `max-len:`, or both.

Example:

```
properties:
  # An array of exactly 3 integers
  coordinates:
    type: array
    min-len: 3
    max-len: 3
    description: 3D coordinates

  # A list of up to 4 GPIO phandles
  gpios:
    type: phandle-array
    max-len: 4
    description: Up to 4 GPIOs
```

const This specifies a constant value the property must take. It is mainly useful for constraining the values of common properties for a particular piece of hardware.

Warning

It is an abuse of this feature to use it to name properties in unconventional ways.

For example, this feature is not meant for cases like naming a property `my-pin`, then assigning it to the "gpio" specifier space using this feature. Properties which refer to GPIOs should use conventional names, i.e. end in `-gpios` or `-gpio`.

specifier-space This property, if present, manually sets the specifier space associated with a property with type `phandle-array`.

Normally, the specifier space is encoded implicitly in the property name. A property named `foos` with type `phandle-array` implicitly has specifier space `foo`. As a special case, `*-gpios` properties have specifier space "gpio", so that `foo-gpios` will have specifier space "gpio" rather than "foo-gpio".

You can use `specifier-space` to manually provide a space if using this convention would result in an awkward or unconventional name.

For example:

```
compatible: ...
properties:
  bar:
    type: phandle-array
    specifier-space: my-custom-space
```

Above, the `bar` property's specifier space is set to "my-custom-space".

You could then use the property in a devicetree like this:

```
controller1: custom-controller@1000 {
        #my-custom-space-cells = <2>;
};

controller2: custom-controller@2000 {
        #my-custom-space-cells = <1>;
};

my-node {
        bar = <&controller1 10 20>, <&controller2 30>;
};
```

Generally speaking, you should reserve this feature for cases where the implicit specifier space naming convention doesn't work. One appropriate example is an `mboxes` property with specifier space "mbox", not "mboxe". You can write this property as follows:

```
properties:
  mboxes:
    type: phandle-array
    specifier-space: mbox
```

dependency-mode The `dependency-mode` setting controls how phandle properties are treated when calculating the dependency graph in Zephyr.

By default, any *phandle property* creates a dependency between the node containing the property and the node it references. The `dependency-mode` setting allows you to override this behavior.

Possible values The `dependency-mode` setting accepts the following values:

`normal` or unspecified
The default behavior. The referencing node depends on the referenced node (the node pointed to by the phandle).

`reverse`
Reverses the dependency direction. Instead of the referencing node depending on the referenced node, the referenced node depends on the referencing node.

`ignore`
The phandle property does not create a dependency.

`child-ignore`
Similar to `ignore`, but the dependency is only ignored if the referenced node is a child of the referencing node. If the referenced node is not a child, a normal dependency is created.

Use this when a phandle may reference either a child node (no dependency needed) or an external node (dependency required).

Using dependency-mode The `dependency-mode` setting is specified within a property definition in a binding file. Here's an example:

```
compatible: "vendor,ethernet-controller"

properties:
  phy-handle:
    type: phandle
    description: |
      Specifies a reference to a node representing a PHY device.
    dependency-mode: ignore
```

Another example using `child-ignore`:

```
compatible: "vendor,node-with-optional-phandle"

properties:
  optional-ref:
    type: phandle
    description: |
      References either a child node or an external device.
      Child node references create no dependency, but external
      device references do.
    dependency-mode: child-ignore
```

Example DTS using that binding:

```
root {
        external_dev: external-device@2000 {
                compatible = "vendor,external-device";
                reg = <0x2000 0x100>;
        };

        parent_dev: parent-device@1000 {
                compatible = "vendor,node-with-optional-phandle";
                reg = <0x1000 0x100>;
                optional-ref = <&internal_child>; /* child: ignored dependency */

                internal_child: internal-device {
                        compatible = "vendor,internal-device";
                };
        };
```

(continues on next page)

(continued from previous page)

```
        peer_dev: peer-device@3000 {
                compatible = "vendor,node-with-optional-phandle";
                reg = <0x3000 0x100>;
                optional-ref = <&external_dev>; /* external: normal dependency */
        };
};
```

Default behavior If `dependency-mode` is not specified, the default value is `normal`, which means the referencing node depends on the referenced node.

Child-binding `child-binding` can be used when a node has children that all share the same properties. Each child gets the contents of `child-binding` as its binding, though an explicit `compatible = ...` on the child node takes precedence, if a binding is found for it.

Consider a binding for a PWM LED node like this one, where the child nodes are required to have a `pwms` property:

```
pwmleds {
        compatible = "pwm-leds";

        red_pwm_led {
                pwms = <&pwm3 4 15625000>;
        };
        green_pwm_led {
                pwms = <&pwm3 0 15625000>;
        };
        /* ... */
};
```

The binding would look like this:

```
compatible: "pwm-leds"

child-binding:
  description: LED that uses PWM

  properties:
    pwms:
      type: phandle-array
      required: true
```

`child-binding` also works recursively. For example, this binding:

```
compatible: foo

child-binding:
  child-binding:
    properties:
      my-property:
        type: int
        required: true
```

will apply to the `grandchild` node in this DTS:

```
parent {
        compatible = "foo";
        child {
                grandchild {
                        my-property = <123>;
                };
        };
};
```

Bus If the node is a bus controller, use `bus:` in the binding to say what type of bus. For example, a binding for a SPI peripheral on an SoC would look like this:

```
compatible: "manufacturer,spi-peripheral"
bus: spi
# ...
```

The presence of this key in the binding informs the build system that the children of any node matching this binding appear on this type of bus.

This in turn influences the way `on-bus:` is used to match bindings for the child nodes.

For a single bus supporting multiple protocols, e.g. I3C and I2C, the `bus:` in the binding can have a list as value:

```
compatible: "manufacturer,i3c-controller"
bus: [i3c, i2c]
# ...
```

On-bus If the node appears as a device on a bus, use `on-bus:` in the binding to say what type of bus.

For example, a binding for an external SPI memory chip should include this line:

```
on-bus: spi
```

And a binding for an I2C based temperature sensor should include this line:

```
on-bus: i2c
```

When looking for a binding for a node, the build system checks if the binding for the parent node contains `bus: <bus type>`. If it does, then only bindings with a matching `on-bus: <bus type>` and bindings without an explicit `on-bus` are considered. Bindings with an explicit `on-bus: <bus type>` are searched for first, before bindings without an explicit `on-bus`. The search repeats for each item in the node's `compatible` property, in order.

This feature allows the same device to have different bindings depending on what bus it appears on. For example, consider a sensor device with compatible `manufacturer,sensor` which can be used via either I2C or SPI.

The sensor node may therefore appear in the devicetree as a child node of either an SPI or an I2C controller, like this:

```
spi-bus@0 {
   /* ... some compatible with 'bus: spi', etc. ... */

   sensor@0 {
       compatible = "manufacturer,sensor";
       reg = <0>;
       /* ... */
```

(continues on next page)

(continued from previous page)

```
    };
};

i2c-bus@0 {
    /* ... some compatible with 'bus: i2c', etc. ... */

    sensor@79 {
        compatible = "manufacturer,sensor";
        reg = <79>;
        /* ... */
    };
};
```

You can write two separate binding files which match these individual sensor nodes, even though they have the same compatible:

```
# manufacturer,sensor-spi.yaml, which matches sensor@0 on the SPI bus:
compatible: "manufacturer,sensor"
on-bus: spi

# manufacturer,sensor-i2c.yaml, which matches sensor@79 on the I2C bus:
compatible: "manufacturer,sensor"
properties:
  uses-clock-stretching:
    type: boolean
on-bus: i2c
```

Only `sensor@79` can have a `use-clock-stretching` property. The bus-sensitive logic ignores `manufacturer,sensor-i2c.yaml` when searching for a binding for `sensor@0`.

Class If nodes matching the binding can be used as devices of one or more Zephyr device classes, use `class:` to declare those classes:

```
compatible: "manufacturer,adc"
class: adc
```

The value is a device class name or a list of names. Names use lowercase letters, digits, `-` and `_`; malformed or duplicate names are rejected. Each name should match a device API class registered with `DEVICE_API` (like adc); lowercase-and-hyphens names are converted to lowercase-and-underscores C tokens the same way compatibles are.

The key declares which device API class(es) the node's drivers *can* implement; which driver and API are actually built for a node remains a Kconfig decision. A list value covers hardware whose drivers can implement more than one API depending on configuration:

```
compatible: "manufacturer,rtc"
class: [rtc, counter]
```

`class:` is normally declared once in a device class base binding (for example dts/bindings/adc/adc-controller.yaml) so that every binding including it inherits the class. When a binding and its included files declare classes, the values are unioned: a binding that includes several class base bindings belongs to all of their classes.

Class membership can be queried at build time with `DT_NODE_HAS_CLASS`, iterated with `DT_FOREACH_CLASS_STATUS_OKAY`, counted with `DT_NUM_CLASS_STATUS_OKAY`, and tested from Kconfig with `$(dt_class_enabled,<class name>)`.

To add a class to a device whose binding cannot be changed, for example to classify an upstream device from a downstream module, define a more specific compatible whose binding includes

the original binding and declares the class, and list both compatibles on the node:

```
# vnd,foo-classed.yaml
compatible: "vnd,foo-classed"
include: vnd,foo.yaml
class: xyz
```

```
compatible = "vnd,foo-classed", "vnd,foo";
```

The node's binding, and with it its classes, comes from the first compatible that has a binding; drivers match on any of the node's compatibles, so the original driver still binds. Two bindings for the same compatible are an error, so an existing binding cannot be shadowed.

Examples If you feel you want to provide a minimal example for your binding, you can use it like this:

```
description: ...

properties:
 ...

examples:
  - |
    leds {
      compatible = "gpio-leds";

      uled: led {
      gpios = <&gpioe 12 GPIO_ACTIVE_HIGH>;
      };
    };
```

Specifier cell names (*-cells) This section documents how to name the cells in a specifier within a binding. These concepts are discussed in detail later in this guide in *phandle-array properties*.

Consider a binding for a node whose phandle may appear in a `phandle-array` property, like the PWM controllers `pwm1` and `pwm2` in this example:

```
pwm1: pwm@deadbeef {
    compatible = "foo,pwm";
    #pwm-cells = <2>;
};

pwm2: pwm@deadbeef {
    compatible = "bar,pwm";
    #pwm-cells = <1>;
};

my-node {
    pwms = <&pwm1 1 2000>, <&pwm2 3000>;
};
```

The bindings for compatible `"foo,pwm"` and `"bar,pwm"` must give a name to the cells that appear in a PWM specifier using `pwm-cells:`, like this:

```
# foo,pwm.yaml
compatible: "foo,pwm"
```

(continues on next page)

(continued from previous page)

```
...
pwm-cells:
  - channel
  - period

# bar,pwm.yaml
compatible: "bar,pwm"
...
pwm-cells:
  - period
```

A `*-names` (e.g. `pwm-names`) property can appear on the node as well, giving a name to each entry.

This allows the cells in the specifiers to be accessed by name, e.g. using APIs like `DT_PWMS_CHANNEL_BY_NAME`.

If the specifier is empty (e.g. `#clock-cells = <0>`), then `*-cells` can either be omitted (recommended) or set to an empty array. Note that an empty array is specified as e.g. `clock-cells: []` in YAML.

Include Bindings can include other files, which can be used to share common property definitions between bindings. Use the `include:` key for this. Its value is either a string or a list.

In the simplest case, you can include another file by giving its name as a string, like this:

```
include: foo.yaml
```

If any file named `foo.yaml` is found (see *Where bindings are located* for the search process), it will be included into this binding.

Included files are merged into bindings with a simple recursive dictionary merge. The build system will check that the resulting merged binding is well-formed. It is allowed to include at any level, including `child-binding`, like this:

```
# foo.yaml will be merged with content at this level
include: foo.yaml

child-binding:
  # bar.yaml will be merged with content at this level
  include: bar.yaml
```

It is an error if a key appears with a different value in a binding and in a file it includes, with two exceptions: a binding can have `required: true` for a *property definition* for which the included file has `required: false` (the `required: true` takes precedence, allowing bindings to strengthen requirements from included files), and `class:` values are unioned as described in *Class*.

Note that weakening requirements by having `required: false` where the included file has `required: true` is an error. This is meant to keep the organization clean.

The file base.yaml contains definitions for many common properties. When writing a new binding, it is a good idea to check if `base.yaml` already defines some of the needed properties, and include it if it does.

Note that you can make a property defined in base.yaml obligatory like this, taking *reg* as an example:

```
reg:
  required: true
```

This relies on the dictionary merge to fill in the other keys for `reg`, like `type`.

To include multiple files, you can use a list of strings:

```
include:
  - foo.yaml
  - bar.yaml
```

This includes the files `foo.yaml` and `bar.yaml`. (You can write this list in a single line of YAML as `include: [foo.yaml, bar.yaml]`.)

When including multiple files, any overlapping `required` keys on properties in the included files are ORed together. This makes sure that a `required: true` is always respected.

In some cases, you may want to include some property definitions from a file, but not all of them. In this case, `include:` should be a list, and you can filter out just the definitions you want by putting a mapping in the list, like this:

```
include:
  - name: foo.yaml
    property-allowlist:
      - i-want-this-one
      - and-this-one
  - name: bar.yaml
    property-blocklist:
      - do-not-include-this-one
      - or-this-one
```

Each map element must have a `name` key which is the filename to include, and may have `property-allowlist` and `property-blocklist` keys that filter which properties are included.

You cannot have a single map element with both `property-allowlist` and `property-blocklist` keys. A map element with neither `property-allowlist` nor `property-blocklist` is valid; no additional filtering is done.

You can freely intermix strings and mappings in a single `include:` list:

```
include:
  - foo.yaml
  - name: bar.yaml
    property-blocklist:
      - do-not-include-this-one
      - or-this-one
```

Finally, you can filter from a child binding like this:

```
include:
  - name: bar.yaml
    child-binding:
      property-allowlist:
        - child-prop-to-allow
```

Nexus nodes and maps All `phandle-array` type properties support mapping through `*-map` properties, e.g. `gpio-map`, as defined by the Devicetree specification.

This is used, for example, to define connector nodes for common breakout headers, such as the `arduino_header` nodes that are conventionally defined in the devicetrees for boards with Arduino compatible expansion headers.

Rules for upstream bindings This section includes general rules for writing bindings that you want to submit to the upstream Zephyr Project. (You don't need to follow these rules for bindings you don't intend to contribute to the Zephyr Project, but it's a good idea.)

Decisions made by the Zephyr devicetree maintainer override the contents of this section. If that happens, though, please let them know so they can update this page, or you can send a patch yourself.

Contents

- *Always check for existing bindings*
- *General rules*
  - *File names*
  - *Recommendations are requirements*
  - *Descriptions*
  - *Naming conventions*
- *Rules for vendor prefixes*
- *Rules for default values*
- *The `zephyr,` prefix*

Always check for existing bindings Zephyr aims for devicetree *Source compatibility with other operating systems*. Therefore, if there is an existing binding for your device in an authoritative location, you should try to replicate its properties when writing a Zephyr binding, and you must justify any Zephyr-specific divergences.

In particular, this rule applies if:

- There is an existing binding in the mainline Linux kernel. See `Documentation/devicetree/bindings` in Linus's tree for existing bindings and the Linux devicetree documentation for more information.
- Your hardware vendor provides an official binding outside of the Linux kernel.

General rules Wherever possible, when writing Devicetree bindings for Zephyr, try to follow the same design guidelines laid out by Linux.

File names Bindings which match a compatible must have file names based on the compatible.

- For example, a binding for compatible `vnd,foo` must be named `vnd,foo.yaml`.
- If the binding is bus-specific, you can append the bus to the file name; for example, if the binding YAML has `on-bus: bar`, you may name the file `vnd,foo-bar.yaml`.

Recommendations are requirements All recommendations in *default* are requirements when submitting the binding.

In particular, if you use the `default:` feature, you must justify the value in the property's description.

Descriptions There are only two acceptable ways to write property `description:` strings.

If your description is short, it's fine to use this style:

```
description: my short string
```

If your description is long or spans multiple lines, you must use this style:

```
title: I'm sure you need a short title.

description: |
  My very long string
  goes here.
  Look at all these lines!
```

This | style prevents YAML parsers from removing the newlines in multi-line descriptions. This in turn makes these long strings display properly in the *Bindings index*.

If using the binding's properties gets complicated, you can use examples to provide a minimal node. e.g.:

```
description: ...

properties:
 ...

examples:
  - |
    leds {
      compatible = "gpio-leds";

      uled: led {
      gpios = <&gpioe 12 GPIO_ACTIVE_HIGH>;
      };
    };
```

Naming conventions Do not use uppercase letters (A through Z) or underscores (_) in property names. Use lowercase letters (a through z) instead of uppercase. Use dashes (-) instead of underscores. (The one exception to this rule is if you are replicating a well-established binding from somewhere like Linux.)

Rules for vendor prefixes The following general rules apply to vendor prefixes in *compatible* properties.

- If your device is manufactured by a specific vendor, then its compatible should have a vendor prefix.

  If your binding describes hardware with a well known vendor from the list in dts/bindings/vendor-prefixes.txt, you must use that vendor prefix.

- If your device is not manufactured by a specific hardware vendor, do **not** invent a vendor prefix. Vendor prefixes are not mandatory parts of compatible properties, and compatibles should not include them unless they refer to an actual vendor. There are some exceptions to this rule, but the practice is strongly discouraged.

- Do not submit additions to Zephyr's `dts/bindings/vendor-prefixes.txt` file unless you also include users of the new prefix. This means at least a binding and a devicetree using the vendor prefix, and should ideally include a device driver handling that compatible.

  For custom bindings, you can add a custom `dts/bindings/vendor-prefixes.txt` file to any directory in your *DTS_ROOT*. The devicetree tooling will respect these prefixes, and will not generate warnings or errors if you use them in your own bindings or devicetrees.

- We sometimes synchronize Zephyr's vendor-prefixes.txt file with the Linux kernel's equivalent file; this process is exempt from the previous rule.
- If your binding is describing an abstract class of hardware with Zephyr specific drivers handling the nodes, it's usually best to use `zephyr` as the vendor prefix. See *Zephyr Project (zephyr)* for examples.

Rules for default values In any case where `default:` is used in a devicetree binding, the `description:` for that property **must** explain *why* the value was selected and any conditions that would make it necessary to provide a different value. Additionally, if changing one property would require changing another to create a consistent configuration, then those properties should be made required.

There is no need to document the default value itself; this is already present in the *Bindings index* output.

There is a risk in using `default:` when the value in the binding may be incorrect for a particular board or hardware configuration. For example, defaulting the capacity of the connected power cell in a charging IC binding is likely to be incorrect. For such properties it's better to make the property `required: true`, forcing the user to make an explicit choice.

Driver developers should use their best judgment as to whether a value can be safely defaulted. Candidates for default values include:

- delays that would be different only under unusual conditions (such as intervening hardware)
- configuration for devices that have a standard initial configuration (such as a USB audio headset)
- defaults which match the vendor-specified power-on reset value (as long as they are independent from other properties)

Default values for `status`, `#address-cells` and `#size-cells` cannot be defined in the bindings. The default behavior for these properties is already defined in the devicetree specification §2.3.4 and §2.3.5.

Examples of how to write descriptions according to these rules:

```
properties:
  cs-interval:
    type: int
    default: 0
    description: |
      Minimum interval between chip select deassertion and assertion.
      The default corresponds to the reset value of the register field.
  hold-time-ms:
    type: int
    default: 20
    description: |
      Amount of time to hold the power enable GPIO asserted before
      initiating communication. The default was recommended in the
      manufacturer datasheet, and would only change under very
      cold temperatures.
```

Some examples of what **not** to do, and why:

```
properties:
  # Description doesn't mention anything about the default
  foo:
    type: int
    default: 1
```

(continues on next page)

(continued from previous page)

```
    description: number of foos

  # Description mentions the default value instead of why it
  # was chosen
  bar:
    type: int
    default: 2
    description: bar size; default is 2

  # Explanation of the default value is in a comment instead
  # of the description. This won't be shown in the bindings index.
  baz:
    type: int
    # This is the recommended value chosen by the manufacturer.
    default: 2
    description: baz time in milliseconds
```

The `zephyr,` prefix You must add this prefix to property names in the following cases:

- Zephyr-specific extensions to bindings we share with upstream Linux. One example is the `zephyr,vref-mv` ADC channel property which is common to ADC controllers defined in dts/bindings/adc/adc-controller.yaml. This channel binding is partially shared with an analogous Linux binding, and Zephyr-specific extensions are marked as such with the prefix.
- Configuration values that are specific to a Zephyr device driver. One example is the `zephyr,lazy-load` property in the `ti,bq274xx` binding. Though devicetree in general is a hardware description and configuration language, it is Zephyr's only mechanism for configuring driver behavior for an individual `struct device`. Therefore, as a compromise, we do allow some software configuration in Zephyr's devicetree bindings, as long as they use this prefix to show that they are Zephyr specific.

You may use the `zephyr,` prefix when naming a devicetree compatible that is specific to Zephyr. One example is `zephyr,ipc-openamp-static-vrings`. In this case, it's permitted but not required to add the `zephyr,` prefix to properties defined in the binding.

Devicetree access from C/C++

This guide describes Zephyr's `<zephyr/devicetree.h>` API for reading the devicetree from C source files. It assumes you're familiar with the concepts in *Introduction to devicetree* and *Devicetree bindings*. See *Devicetree Reference* for reference material.

A note for Linux developers Linux developers familiar with devicetree should be warned that the API described here differs significantly from how devicetree is used on Linux.

Instead of generating a C header with all the devicetree data which is then abstracted behind a macro API, the Linux kernel would instead read the devicetree data structure in its binary form. The binary representation is parsed at runtime, for example to load and initialize device drivers.

Zephyr does not work this way because the size of the devicetree binary and associated handling code would be too large to fit comfortably on the relatively constrained devices Zephyr supports.

Node identifiers To get information about a particular devicetree node, you need a *node identifier* for it. This is a just a C macro that refers to the node.

These are the main ways to get a node identifier:

By path
Use `DT_PATH()` along with the node's full path in the devicetree, starting from the root node. This is mostly useful if you happen to know the exact node you're looking for.

By node label
Use `DT_NODELABEL()` to get a node identifier from a *node label*. Node labels are often provided by SoC `.dtsi` files to give nodes names that match the SoC datasheet, like `i2c1`, `spi2`, etc.

By alias
Use `DT_ALIAS()` to get a node identifier for a property of the special `/aliases` node. This is sometimes done by applications (like blinky, which uses the `led0` alias) that need to refer to *some* device of a particular type ("the board's user LED") but don't care which one is used. You may also use `DT_HAS_ALIAS()` to verify whether an alias node exists.

By instance number
This is done primarily by device drivers, as instance numbers are a way to refer to individual nodes based on a matching compatible. Get these with `DT_INST()`, but be careful doing so. See below.

By chosen node
Use `DT_CHOSEN()` to get a node identifier for `/chosen` node properties.

By parent/child
Use `DT_PARENT()` and `DT_CHILD()` to get a node identifier for a parent or child node, starting from a node identifier you already have.

Two node identifiers which refer to the same node are identical and can be used interchangeably.

Here's a DTS fragment for some imaginary hardware we'll return to throughout this file for examples:

```
/dts-v1/;

/ {
        aliases {
                sensor-controller = &i2c1;
        };

        soc {
                i2c1: i2c@40002000 {
                        compatible = "vnd,soc-i2c";
                        label = "I2C_1";
                        reg = <0x40002000 0x1000>;
                        status = "okay";
                        clock-frequency = <100000>;
                };
        };
};
```

Here are a few ways to get node identifiers for the `i2c@40002000` node:

- `DT_PATH(soc, i2c_40002000)`
- `DT_NODELABEL(i2c1)`
- `DT_ALIAS(sensor_controller)`
- `DT_INST(x, vnd_soc_i2c)` for some unknown number `x`. See the `DT_INST()` documentation for details.

Important

Non-alphanumeric characters like dash (-) and the at sign (@) in devicetree names are converted to underscores (_). The names in a DTS are also converted to lowercase.

Node identifiers are not values There is no way to store one in a variable. You cannot write:

```
/* These will give you compiler errors: */

void *i2c_0 = DT_INST(0, vnd_soc_i2c);
unsigned int i2c_1 = DT_INST(1, vnd_soc_i2c);
long my_i2c = DT_NODELABEL(i2c1);
```

If you want something short to save typing, use C macros:

```
/* Use something like this instead: */

#define MY_I2C DT_NODELABEL(i2c1)

#define INST(i) DT_INST(i, vnd_soc_i2c)
#define I2C_0 INST(0)
#define I2C_1 INST(1)
```

Property access The right API to use to read property values depends on the node and property.

- *Checking properties and values*
- *Simple properties*
- *reg properties*
- *interrupts properties*
- *phandle properties*

Checking properties and values You can use `DT_NODE_HAS_PROP()` to check if a node has a property. For the *example devicetree* above:

```
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), clock_frequency)  /* expands to 1 */
DT_NODE_HAS_PROP(DT_NODELABEL(i2c1), not_a_property)   /* expands to 0 */
```

Simple properties Use `DT_PROP(node_id, property)` to read basic integer, boolean, string, numeric array, and string array properties.

For example, to read the `clock-frequency` property's value in the *above example*:

```
DT_PROP(DT_PATH(soc, i2c_40002000), clock_frequency)  /* This is 100000, */
DT_PROP(DT_NODELABEL(i2c1), clock_frequency)          /* and so is this, */
DT_PROP(DT_ALIAS(sensor_controller), clock_frequency) /* and this. */
```

Important

The DTS property `clock-frequency` is spelled `clock_frequency` in C. That is, properties also need special characters converted to underscores. Their names are also forced to lowercase.

Properties with `string` and `boolean` types work the exact same way. The `DT_PROP()` macro expands to a string literal in the case of strings, and the number 0 or 1 in the case of booleans. For example:

```
#define I2C1 DT_NODELABEL(i2c1)

DT_PROP(I2C1, status)  /* expands to the string literal "okay" */
```

> **Note**
>
> Don't use DT_NODE_HAS_PROP() for boolean properties. Use DT_PROP() instead as shown above. It will expand to either 0 or 1 depending on if the property is present or absent.

Properties with type `array`, `uint8-array`, and `string-array` work similarly, except `DT_PROP()` expands to an array initializer in these cases. Here is an example devicetree fragment:

```
foo: foo@1234 {
        a = <1000 2000 3000>; /* array */
        b = [aa bb cc dd];    /* uint8-array */
        c = "bar", "baz";     /* string-array */
};
```

Its properties can be accessed like this:

```
#define FOO DT_NODELABEL(foo)

int a[] = DT_PROP(FOO, a);           /* {1000, 2000, 3000} */
unsigned char b[] = DT_PROP(FOO, b); /* {0xaa, 0xbb, 0xcc, 0xdd} */
char* c[] = DT_PROP(FOO, c);         /* {"bar", "baz"} */
```

You can use `DT_PROP_LEN()` to get logical array lengths in number of elements.

```
size_t a_len = DT_PROP_LEN(FOO, a); /* 3 */
size_t b_len = DT_PROP_LEN(FOO, b); /* 4 */
size_t c_len = DT_PROP_LEN(FOO, c); /* 2 */
```

`DT_PROP_LEN()` cannot be used with the special `reg` or `interrupts` properties. These have alternative macros which are described next.

reg properties See *Important properties* for an introduction to `reg`.

Given a node identifier `node_id`, `DT_NUM_REGS(node_id)` is the total number of register blocks in the node's `reg` property.

You **cannot** read register block addresses and lengths with `DT_PROP(node, reg)`. Instead, if a node only has one register block, use `DT_REG_ADDR` or `DT_REG_SIZE`:

- `DT_REG_ADDR(node_id)`: the given node's register block address
- `DT_REG_SIZE(node_id)`: its size

Use `DT_REG_ADDR_BY_IDX` or `DT_REG_SIZE_BY_IDX` instead if the node has multiple register blocks:

- `DT_REG_ADDR_BY_IDX(node_id, idx)`: address of register block at index `idx`
- `DT_REG_SIZE_BY_IDX(node_id, idx)`: size of block at index `idx`

The `idx` argument to these must be an integer literal or a macro that expands to one without requiring any arithmetic. In particular, `idx` cannot be a variable. This won't work:

```
/* This will cause a compiler error. */

for (size_t i = 0; i < DT_NUM_REGS(node_id); i++) {
        size_t addr = DT_REG_ADDR_BY_IDX(node_id, i);
}
```

interrupts properties See *Important properties* for a brief introduction to `interrupts`.

Given a node identifier `node_id`, `DT_NUM_IRQS(node_id)` is the total number of interrupt specifiers in the node's `interrupts` property.

The most general purpose API macro for accessing these is `DT_IRQ_BY_IDX`:

```
DT_IRQ_BY_IDX(node_id, idx, val)
```

Here, `idx` is the logical index into the `interrupts` array, i.e. it is the index of an individual interrupt specifier in the property. The `val` argument is the name of a cell within the interrupt specifier. To use this macro, check the bindings file for the node you are interested in to find the `val` names.

Most Zephyr devicetree bindings have a cell named `irq`, which is the interrupt number. You can use `DT_IRQN` as a convenient way to get a processed view of this value.

Warning

Here, "processed" reflects Zephyr's devicetree *Scripts and tools*, which change the `irq` number in *zephyr.dts* to handle hardware constraints on some SoCs and in accordance with Zephyr's multilevel interrupt numbering.

This is currently not very well documented, and you'll need to read the scripts' source code and existing drivers for more details if you are writing a device driver.

Note

See *Phandles* for a detailed guide to phandles.

phandle properties Property values can refer to other nodes using the `&another-node` phandle syntax introduced in *Writing property values*. Properties which contain phandles have type `phandle`, `phandles`, or `phandle-array` in their bindings. We'll call these "phandle properties" for short.

You can convert a phandle to a node identifier using `DT_PHANDLE`, `DT_PHANDLE_BY_IDX`, or `DT_PHANDLE_BY_NAME`, depending on the type of property you are working with.

One common use case for phandle properties is referring to other hardware in the tree. In this case, you usually want to convert the devicetree-level phandle to a Zephyr driver-level *struct device*. See *Get a struct device from a devicetree node* for ways to do that.

Another common use case is accessing specifier values in a phandle array. The general purpose APIs for this are `DT_PHA_BY_IDX` and `DT_PHA`. There are also hardware-specific shorthands like `DT_GPIO_CTLR_BY_IDX`, `DT_GPIO_CTLR`, `DT_GPIO_PIN_BY_IDX`, `DT_GPIO_PIN`, `DT_GPIO_FLAGS_BY_IDX`, and `DT_GPIO_FLAGS`.

See `DT_PHA_HAS_CELL_AT_IDX` and `DT_PROP_HAS_IDX` for ways to check if a specifier value is present in a phandle property.

Other APIs Here are pointers to some other available APIs.

- DT_CHOSEN, DT_HAS_CHOSEN: for properties of the special /chosen node
- DT_HAS_COMPAT_STATUS_OKAY, DT_NODE_HAS_COMPAT: global- and node-specific tests related to the compatible property
- DT_BUS: get a node's bus controller, if there is one
- DT_ENUM_IDX: for properties whose values are among a fixed list of choices
- *Fixed and mapped flash partitions*: APIs for managing fixed flash partitions and mapped flash partitions. Also see *Flash map*, which wraps this in a more user-friendly API.

Device driver conveniences Special purpose macros are available for writing device drivers, which usually rely on *instance identifiers*.

To use these, you must define DT_DRV_COMPAT to the compat value your driver implements support for. This compat value is what you would pass to DT_INST.

If you do that, you can access the properties of individual instances of your compatible with less typing, like this:

```
#include <zephyr/devicetree.h>

#define DT_DRV_COMPAT my_driver_compat

/* This is same thing as DT_INST(0, my_driver_compat): */
DT_DRV_INST(0)

/*
 * This is the same thing as
 * DT_PROP(DT_INST(0, my_driver_compat), clock_frequency)
 */
DT_INST_PROP(0, clock_frequency)
```

See *Instance-based APIs* for a generic API reference.

Hardware specific APIs Convenience macros built on top of the above APIs are also defined to help readability for hardware specific code. See *Hardware specific APIs* for details.

Generated macros While the zephyr/devicetree.h API is not generated, it does rely on a generated C header which is put into every application build directory: *devicetree_generated.h*. This file contains macros with devicetree data.

These macros have tricky naming conventions which the *Devicetree API* API abstracts away. They should be considered an implementation detail, but it's useful to understand them since they will frequently be seen in compiler error messages.

This section contains an Augmented Backus-Naur Form grammar for these generated macros, with examples and more details in comments. See **RFC 7405** (which extends **RFC 5234**) for a syntax specification.

```
; An RFC 7405 ABNF grammar for devicetree macros.
;
; This does *not* cover macros pulled out of DT via Kconfig,
; like CONFIG_SRAM_BASE_ADDRESS, etc. It only describes the
; ones that start with DT_ and are directly generated.

; ---------------------------------------------------------------------
; dt-macro: the top level nonterminal for a devicetree macro
```

(continues on next page)

(continued from previous page)

```
;
; A dt-macro starts with uppercase "DT_", and is one of:
;
; - a <node-macro>, generated for a particular node
; - some <other-macro>, a catch-all for other types of macros
dt-macro = node-macro / other-macro

; ---------------------------------------------------------------------
; node-macro: a macro related to a node

; A macro about a property value
node-macro =  property-macro
; A macro about the pinctrl properties in a node.
node-macro =/ pinctrl-macro
; A macro about the GPIO hog properties in a node.
node-macro =/ gpiohogs-macro
; A macro about the map properties in a node.
node-macro =/ map-macro
; EXISTS macro: node exists in the devicetree
node-macro =/ %s"DT_N" path-id %s"_EXISTS"
; Bus macros: the plain BUS is a way to access a node's bus controller.
; The additional dt-name suffix is added to match that node's bus type;
; the dt-name in this case is something like "spi" or "i2c".
node-macro =/ %s"DT_N" path-id %s"_BUS" ["_" dt-name]
; Device class macros: declares that the node's binding declares
; membership of a device class; the dt-name is something like "adc".
node-macro =/ %s"DT_N" path-id %s"_CLASS_" dt-name
; The reg property is special and has its own macros.
node-macro =/ %s"DT_N" path-id %s"_REG_NUM"
node-macro =/ %s"DT_N" path-id %s"_REG_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_REG_IDX_" DIGIT
              %s"_VAL_" ( %s"ADDRESS" / %s"SIZE")
node-macro =/ %s"DT_N" path-id %s"_REG_NAME_" dt-name
              %s"_VAL_" ( %s"ADDRESS" / %s"SIZE")
node-macro =/ %s"DT_N" path-id %s"_REG_NAME_" dt-name "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_REG"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_REG_SEP"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_REG_VARGS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_REG_SEP_VARGS"
; The interrupts property is also special.
node-macro =/ %s"DT_N" path-id %s"_IRQ_NUM"
node-macro =/ %s"DT_N" path-id %s"_IRQ_LEVEL"
node-macro =/ %s"DT_N" path-id %s"_IRQ_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_IRQ_IDX_" DIGIT
              %s"_VAL_" dt-name [ %s"_EXISTS" ]
node-macro =/ %s"DT_N" path-id %s"_CONTROLLER"
node-macro =/ %s"DT_N" path-id %s"_IRQ_NAME_" dt-name
              %s"_VAL_" dt-name [ %s"_EXISTS" ]
node-macro =/ %s"DT_N" path-id %s"_IRQ_NAME_" dt-name "_CONTROLLER"
; The ranges property is also special.
node-macro =/ %s"DT_N" path-id %s"_RANGES_NUM"
node-macro =/ %s"DT_N" path-id %s"_RANGES_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_RANGES_IDX_" DIGIT
              %s"_VAL_" ( %s"CHILD_BUS_FLAGS" / %s"CHILD_BUS_ADDRESS" /
                          %s"PARENT_BUS_ADDRESS" / %s"LENGTH")
node-macro =/ %s"DT_N" path-id %s"_RANGES_IDX_" DIGIT
```

(continues on next page)

(continued from previous page)

```
              %s"_VAL_CHILD_BUS_FLAGS_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_RANGE"
; Subnodes of the fixed-partitions compatible get macros which contain
; a unique ordinal value for each partition
node-macro =/ %s"DT_N" path-id %s"_PARTITION_ID" DIGIT
; Macros are generated for each of a node's compatibles;
; dt-name in this case is something like "vnd_device".
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MATCHES_" dt-name
node-macro =/ %s"DT_N" path-id %s"_COMPAT_VENDOR_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_COMPAT_VENDOR_IDX_" DIGIT
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MODEL_IDX_" DIGIT "_EXISTS"
node-macro =/ %s"DT_N" path-id %s"_COMPAT_MODEL_IDX_" DIGIT
; The binding compatible (the compatible that matched a binding) as a token,
; uppercased token, or unquoted string. Only defined for nodes with a
; matching binding.
node-macro =/ %s"DT_N" path-id %s"_BINDING_COMPAT_TOKEN"
node-macro =/ %s"DT_N" path-id %s"_BINDING_COMPAT_UPPER_TOKEN"
node-macro =/ %s"DT_N" path-id %s"_BINDING_COMPAT_UNQUOTED"
; Every non-root node gets one of these macros, which expands to the node
; identifier for that node's parent in the devicetree.
node-macro =/ %s"DT_N" path-id %s"_PARENT"
; These are used internally by DT_FOREACH_PROP_ELEM(_SEP)(_VARGS), which
; iterates over each property element.
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM"
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM_SEP"
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM_VARGS"
node-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_PROP_ELEM_SEP_VARGS"
; These are used by DT_CHILD_NUM and DT_CHILD_NUM_STATUS_OKAY macros
node-macro =/ %s"DT_N" path-id %s"_CHILD_NUM"
node-macro =/ %s"DT_N" path-id %s"_CHILD_NUM_STATUS_OKAY"
; These are used by DT_DESCENDANT_NUM_ON_BUS and
; DT_DESCENDANT_NUM_ON_BUS_STATUS_OKAY
; macros to count descendants on a given bus under a controller node.
; Descendants behind child bus-controller nodes are excluded.
; dt-name in this case is something like "spi" or "i2c".
node-macro =/ %s"DT_N" path-id %s"_DESCENDANT_NUM_ON_BUS_" dt-name
node-macro =/ %s"DT_N" path-id %s"_DESCENDANT_NUM_ON_BUS_" dt-name %s"_STATUS_OKAY"
; This is used by DT_CHILD_BY_UNIT_ADDR_INT
node-macro =/ %s"DT_N" path-id %s"_CHILD_UNIT_ADDR_INT_" DIGIT
; These are used internally by DT_FOREACH_CHILD, which iterates over
; each child node.
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_SEP"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_VARGS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_SEP_VARGS"
; These are used internally by DT_FOREACH_CHILD_STATUS_OKAY, which iterates
; over each child node with status "okay".
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_SEP"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_VARGS"
node-macro =/ %s"DT_N" path-id %s"_FOREACH_CHILD_STATUS_OKAY_SEP_VARGS"
; These are used internally by DT_FOREACH_NODELABEL and
; DT_FOREACH_NODELABEL_VARGS, which iterate over a node's node labels.
node-macro =/ %s"DT_N" path-id %s"_FOREACH_NODELABEL" [ %s"_VARGS" ]
; These are used internally by DT_NUM_NODELABELS
node-macro =/ %s"DT_N" path-id %s"_NODELABEL_NUM"
```

(continues on next page)

(continued from previous page)

```
; The node's zero-based index in the list of it's parent's child nodes.
node-macro =/ %s"DT_N" path-id %s"_CHILD_IDX"
; The node's status macro; dt-name in this case is something like "okay"
; or "disabled".
node-macro =/ %s"DT_N" path-id %s"_STATUS_" dt-name
; The node's dependency ordinal. This is a non-negative integer
; value that is used to represent dependency information.
node-macro =/ %s"DT_N" path-id %s"_ORD"
; The node's path, as a string literal
node-macro =/ %s"DT_N" path-id %s"_PATH"
; The node's name@unit-addr, as a string literal
node-macro =/ %s"DT_N" path-id %s"_FULL_NAME"
; The dependency ordinals of a node's requirements (direct dependencies).
node-macro =/ %s"DT_N" path-id %s"_REQUIRES_ORDS"
; The dependency ordinals of a node supports (reverse direct dependencies).
node-macro =/ %s"DT_N" path-id %s"_SUPPORTS_ORDS"

; ---------------------------------------------------------------------
; pinctrl-macro: a macro related to the pinctrl properties in a node
;
; These are a bit of a special case because they kind of form an array,
; but the array indexes correspond to pinctrl-DIGIT properties in a node.
;
; So they're related to a node, but not just one property within the node.
;
; The following examples assume something like this:
;
;      foo {
;              pinctrl-0 = <&bar>;
;              pinctrl-1 = <&baz>;
;              pinctrl-names = "default", "sleep";
;      };

; Total number of pinctrl-DIGIT properties in the node. May be zero.
;
;   #define DT_N_<node path>_PINCTRL_NUM 2
pinctrl-macro = %s"DT_N" path-id %s"_PINCTRL_NUM"
; A given pinctrl-DIGIT property exists.
;
;     #define DT_N_<node path>_PINCTRL_IDX_0_EXISTS 1
;     #define DT_N_<node path>_PINCTRL_IDX_1_EXISTS 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_IDX_" DIGIT %s"_EXISTS"
; A given pinctrl property name exists.
;
;     #define DT_N_<node path>_PINCTRL_NAME_default_EXISTS 1
;     #define DT_N_<node path>_PINCTRL_NAME_sleep_EXISTS 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_EXISTS"
; The corresponding index number of a named pinctrl property.
;
;     #define DT_N_<node path>_PINCTRL_NAME_default_IDX 0
;     #define DT_N_<node path>_PINCTRL_NAME_sleep_IDX 1
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_IDX"
; The node identifier for the phandle in a named pinctrl property.
;
;    #define DT_N_<node path>_PINCTRL_NAME_default_IDX_0_PH <node id for 'bar'>
;
```

(continues on next page)

(continued from previous page)

```
; There's no need for a separate macro for access by index: that's
; covered by property-macro. We only need this because the map from
; names to properties is implicit in the structure of the DT.
pinctrl-macro =/ %s"DT_N" path-id %s"_PINCTRL_NAME_" dt-name %s"_IDX_" DIGIT
↪%s"_PH"

; ---------------------------------------------------------------------
; gpiohogs-macro: a macro related to GPIO hog nodes
;
; The following examples assume something like this:
;
;     gpio1: gpio@... {
;       compatible = "vnd,gpio";
;       #gpio-cells = <2>;
;
;       node-1 {
;               gpio-hog;
;               gpios = <0x0 0x10>, <0x1 0x20>;
;               output-high;
;       };
;
;       node-2 {
;               gpio-hog;
;               gpios = <0x2 0x30>;
;               output-low;
;       };
;     };
;
; Bindings fragment for the vnd,gpio compatible:
;
;     gpio-cells:
;       - pin
;       - flags

; The node contains GPIO hogs.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_EXISTS 1
gpioshogs-macro = %s"DT_N" path-id %s"_GPIO_HOGS_EXISTS"
; Number of hogged GPIOs in a node.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_NUM 2
;   #define DT_N_<node-2 path>_GPIO_HOGS_NUM 1
gpioshogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_NUM"
; A given logical GPIO hog array index exists.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_EXISTS 1
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_EXISTS 1
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_EXISTS"
; The node identifier for the phandle of a logical index in the GPIO hogs array.
; These macros are currently unused by Zephyr.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_PH <node id for 'gpio1'>
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_PH <node id for 'gpio1'>
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_PH <node id for 'gpio1'>
```

(continues on next page)

(continued from previous page)

```
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_PH"
; The pin cell of a logical index in the GPIO hogs array exists.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_pin_EXISTS 1
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_pin_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_pin_EXISTS 1
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_pin_EXISTS"
; The value of the pin cell of a logical index in the GPIO hogs array.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_pin 0
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_pin 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_pin 2
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_pin"
; The flags cell of a logical index in the GPIO hogs array exists.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_flags_EXISTS 1
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_flags_EXISTS 1
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_flags_EXISTS 1
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_flags_EXISTS"
; The value of the flags cell of a logical index in the GPIO hogs array.
;
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_0_VAL_flags 0x10
;   #define DT_N_<node-1 path>_GPIO_HOGS_IDX_1_VAL_flags 0x20
;   #define DT_N_<node-2 path>_GPIO_HOGS_IDX_0_VAL_flags 0x30
gpiohogs-macro =/ %s"DT_N" path-id %s"_GPIO_HOGS_IDX_" DIGIT %s"_VAL_flags"


; ---------------------------------------------------------------------
; map-macro: a macro related to map properties
;
; Macros are generated when there is a property ending with '-map' as shown below.
;
; connector {
;     ...
;     interrupt-map = <0 1 &intc0 2 3>;
; };
;
; The existence of map entry;
;
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_EXISTS 1
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT %s"_EXISTS"
;
; The child address length, value, and existence.
;
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_CHILD_ADDRESS_LEN 1
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_CHILD_ADDRESS_IDX_0 0
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_CHILD_ADDRESS_IDX_0_EXISTS 1
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_CHILD_ADDRESS_LEN"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_CHILD_ADDRESS_IDX_" DIGIT
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_CHILD_ADDRESS_IDX_" DIGIT %s"_EXISTS"
;
; The length of child specifiers length, value, and existence.
;
```

(continues on next page)

(continued from previous page)

```
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_CHILD_SPECIFIER_LEN 1
; #define␣
↪DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_CHILD_SPECIFIER_IDX_0_EXISTS 1
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_CHILD_SPECIFIER_IDX_0 0
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_CHILD_SPECIFIER_LEN"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_CHILD_SPECIFIER_IDX_" DIGIT
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_CHILD_SPECIFIER_IDX_" DIGIT %s"_EXISTS"
;
; The parent node of this map.
;
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT DT_N_S_gpio_emul
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT %s"_PARENT"
;
; The length of parent address, value, and existence.
;
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT_ADDRESS_LEN 1
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT_ADDRESS_IDX_0 2
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT_ADDRESS_IDX_0_EXISTS␣
↪1
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_PARENT_ADDRESS_LEN"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_PARENT_ADDRESS_IDX_" DIGIT
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_PARENT_ADDRESS_IDX_" DIGIT %s"_EXISTS"
;
; The length of parent specifiers length, value, and existence.
;
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT_SPECIFIER_LEN 1
; #define␣
↪DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT_SPECIFIER_IDX_0_EXISTS 1
; #define DT_N_S_connector_P_interrupt_map_MAP_ENTRY_0_PARENT_SPECIFIER_IDX_0 3
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_PARENT_SPECIFIER_LEN"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_PARENT_SPECIFIER_IDX_" DIGIT
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_MAP_ENTRY_" DIGIT
↪%s"_PARENT_SPECIFIER_IDX_" DIGIT %s"_EXISTS"
;
; Iteration helpers for map.
;
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_MAP_ENTRY"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_MAP_ENTRY_SEP"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_MAP_ENTRY_VARGS"
map-macro =/ %s"DT_N" path-id %s"_P_" prop-id %s"_FOREACH_MAP_ENTRY_SEP_VARGS"


; ---------------------------------------------------------------------
; property-macro: a macro related to a node property
;
; These combine a node identifier with a "lowercase-and-underscores form"
; property name. The value expands to something related to the property's
; value.
```

(continues on next page)

(continued from previous page)

```
;
; The optional prop-suf suffix is when there's some specialized
; subvalue that deserves its own macro, like the macros for an array
; property's individual elements
;
; The "plain vanilla" macro for a property's value, with no prop-suf,
; looks like this:
;
;   DT_N_<node path>_P_<property name>
;
; Components:
;
; - path-id: node's devicetree path converted to a C token
; - prop-id: node's property name converted to a C token
; - prop-suf: an optional property-specific suffix
property-macro =  %s"DT_N" path-id %s"_P_" prop-id [prop-suf]

; ---------------------------------------------------------------------
; path-id: a node's path-based macro identifier
;
; This in "lowercase-and-underscores" form. I.e. it is
; the node's devicetree path converted to a C token by changing:
;
; - each slash (/) to _S_
; - all letters to lowercase
; - non-alphanumerics characters to underscores
;
; For example, the leaf node "bar-BAZ" in this devicetree:
;
;   / {
;           foo@123 {
;                   bar-BAZ {};
;           };
;   };
;
; has path-id "_S_foo_123_S_bar_baz".
path-id = 1*( %s"_S_" dt-name )

; ----------------------------------------------------------------------
; prop-id: a property identifier
;
; A property name converted to a C token by changing:
;
; - all letters to lowercase
; - non-alphanumeric characters to underscores
;
; Example node:
;
;   chosen {
;       zephyr,console = &uart1;
;       WHY,AM_I_SHOUTING = "unclear";
;   };
;
; The 'zephyr,console' property has prop-id 'zephyr_console'.
; 'WHY,AM_I_SHOUTING' has prop-id 'why_am_i_shouting'.
prop-id = dt-name
```

(continues on next page)

(continued from previous page)

```
; ---------------------------------------------------------------------
; prop-suf: a property-specific macro suffix
;
; Extra macros are generated for properties:
;
; - that are special to the specification ("reg", "interrupts", etc.)
; - with array types (uint8-array, phandle-array, etc.)
; - with "enum:" in their bindings
; - that have zephyr device API specific macros for phandle-arrays
; - related to phandle specifier names ("foo-names")
;
; Here are some examples:
;
; - _EXISTS: property, index or name existence flag
; - _SIZE: logical property length
; - _IDX_<i>: values of individual array elements
; - _IDX_<DIGIT>_VAL_<dt-name>: values of individual specifier
;   cells within a phandle array
; - _ADDR_<i>: for reg properties, the i-th register block address
; - _LEN_<i>: for reg properties, the i-th register block length
;
; The different cases are not exhaustively documented here to avoid
; this file going stale. Please see devicetree.h if you need to know
; the details.
prop-suf = 1*( "_" gen-name ["_" dt-name] )

; ---------------------------------------------------------------------
; other-macro: grab bag for everything that isn't a node-macro.

; See examples below.
other-macro =  %s"DT_N_" alternate-id
; Total count of enabled instances of a compatible.
other-macro =/ %s"DT_N_INST_" dt-name %s"_NUM_OKAY"
; These are used internally by DT_FOREACH_NODE and
; DT_FOREACH_STATUS_OKAY_NODE respectively.
other-macro =/ %s"DT_FOREACH_HELPER"
other-macro =/ %s"DT_FOREACH_OKAY_HELPER"
; These are used internally by DT_FOREACH_STATUS_OKAY,
; which iterates over each enabled node of a compatible.
other-macro =/ %s"DT_FOREACH_OKAY_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_VARGS_" dt-name
; These are used internally by DT_INST_FOREACH_STATUS_OKAY,
; which iterates over each enabled instance of a compatible.
other-macro =/ %s"DT_FOREACH_OKAY_INST_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_INST_VARGS_" dt-name
; E.g.: #define DT_CHOSEN_zephyr_flash
other-macro =/ %s"DT_CHOSEN_" dt-name
; Declares that a compatible has at least one node on a bus.
; Example:
;
;   #define DT_COMPAT_vnd_dev_BUS_spi 1
other-macro =/ %s"DT_COMPAT_" dt-name %s"_BUS_" dt-name
; Declares that a compatible has at least one status "okay" node.
; Example:
;
```

(continues on next page)

(continued from previous page)

```
;   #define DT_COMPAT_HAS_OKAY_vnd_dev 1
other-macro =/ %s"DT_COMPAT_HAS_OKAY_" dt-name
; Declares that a device class has at least one status "okay" node,
; and the count of such nodes.
; Example:
;
;   #define DT_CLASS_HAS_OKAY_adc 1
;   #define DT_N_CLASS_adc_NUM_OKAY 2
other-macro =/ %s"DT_CLASS_HAS_OKAY_" dt-name
other-macro =/ %s"DT_N_CLASS_" dt-name %s"_NUM_OKAY"
; These are used internally by DT_FOREACH_CLASS_STATUS_OKAY,
; which iterates over each enabled node of a device class.
other-macro =/ %s"DT_FOREACH_OKAY_CLASS_" dt-name
other-macro =/ %s"DT_FOREACH_OKAY_VARGS_CLASS_" dt-name
; Currently used to allow mapping a lowercase-and-underscores "label"
; property to a fixed-partitions node. See the flash map API docs
; for an example.
other-macro =/ %s"DT_COMPAT_" dt-name %s"_LABEL_" dt-name
; Get the node label of the phandle
other-macro =/ %s"DT_N_NODELABEL_" %s"DT_N" path-id "_IDX_" DIGIT "_C_TOKEN"

; ---------------------------------------------------------------------
; alternate-id: another way to specify a node besides a path-id
;
; Example devicetree:
;
;   / {
;           aliases {
;                   dev = &dev_1;
;           };
;
;           soc {
;               dev_1: device@123 {
;                   compatible = "vnd,device";
;               };
;           };
;   };
;
; Node device@123 has these alternate-id values:
;
; - ALIAS_dev
; - NODELABEL_dev_1
; - INST_0_vnd_device
;
; The full alternate-id macros are:
;
;   #define DT_N_INST_0_vnd_device     DT_N_S_soc_S_device_123
;   #define DT_N_ALIAS_dev             DT_N_S_soc_S_device_123
;   #define DT_N_NODELABEL_dev_1       DT_N_S_soc_S_device_123
;
; These mainly exist to allow pasting an alternate-id macro onto a
; "_P_<prop-id>" to access node properties given a node's alias, etc.
;
; Notice that "inst"-type IDs have a leading instance identifier,
; which is generated by the devicetree scripts. The other types of
; alternate-id begin immediately with names taken from the devicetree.
```

(continues on next page)

```
alternate-id =  ( %s"ALIAS" / %s"NODELABEL" ) dt-name
alternate-id =/ %s"INST_" 1*DIGIT "_" dt-name

; ---------------------------------------------------------------------
; miscellaneous helper definitions

; A dt-name is one or more:
; - lowercase ASCII letters (a-z)
; - numbers (0-9)
; - underscores ("_")
;
; They are the result of converting names or combinations of names
; from devicetree to a valid component of a C identifier by
; lowercasing letters (in practice, this is a no-op) and converting
; non-alphanumeric characters to underscores.
;
; You'll see these referred to as "lowercase-and-underscores" forms of
; various devicetree identifiers throughout the documentation.
dt-name = 1*( lower / DIGIT / "_" )

; gen-name is used as a stand-in for a component of a generated macro
; name which does not come from devicetree (dt-name covers that case).
;
; - uppercase ASCII letters (a-z)
; - numbers (0-9)
; - underscores ("_")
gen-name = upper 1*( upper / DIGIT / "_" )

; "lowercase ASCII letter" turns out to be pretty annoying to specify
; in RFC-7405 syntax.
;
; This is just ASCII letters a (0x61) through z (0x7a).
lower = %x61-7A

; "uppercase ASCII letter" in RFC-7405 syntax
upper = %x41-5A
```

Phandles

The devicetree concept of a *phandle* is very similar to pointers in C. You can use phandles to refer to nodes in devicetree similarly to the way you can use pointers to refer to structures in C.

Contents

- *Getting phandles*
- *Using phandles*
  - *One node: phandle type*
  - *Zero or more nodes: phandles type*
  - *Zero or more nodes with metadata: phandle-array type*
- *phandle-array properties*
  - *High level description*

- *Example phandle-arrays: GPIOs*
- *Specifier spaces*
  - *High level description*
  - *Example specifier space: gpio*
- *Associating properties with specifier spaces*
  - *High level description*
  - *Special case: GPIOs and IO channels*
  - *Manually specifying a space*
- *Naming the cells in a specifier*
- *See also*

Getting phandles The usual way to get a phandle for a devicetree node is from one of its node labels. For example, with this devicetree:

```
/ {
        lbl_a: node-1 {};
        lbl_b: lbl_c: node-2 {};
};
```

You can write the phandle for:

- `/node-1` as `&lbl_a`
- `/node-2` as either `&lbl_b` or `&lbl_c`

Notice how the `&nodelabel` devicetree syntax is similar to the "address of" C syntax.

> **Note**
>
> "Type" in this section refers to one of the type names documented in *Properties* in the devicetree bindings documentation.

Using phandles Here are the main ways you will use phandles.

One node: phandle type You can use phandles to refer to `node-b` from `node-a`, where `node-b` is related to `node-a` in some way.

One common example is when `node-a` represents some hardware that generates an interrupt, and `node-b` represents the interrupt controller that receives the asserted interrupt. In this case, you could write:

```
node_b: node-b {
        interrupt-controller;
};

node-a {
        interrupt-parent = <&node_b>;
};
```

This uses the standard `interrupt-parent` property defined in the devicetree specification to capture the relationship between the two nodes.

These properties have type `phandle`.

Zero or more nodes: phandles type You can use phandles to make an array of references to other nodes.

One common example occurs in *pin control*. Pin control properties like `pinctrl-0`, `pinctrl-1` etc. may contain multiple phandles, each of which "points" to a node containing information related to pin configuration for that hardware peripheral. Here's an example of six phandles in a single property:

```
pinctrl-0 = <&quadspi_clk_pe10 &quadspi_ncs_pe11
             &quadspi_bk1_io0_pe12 &quadspi_bk1_io1_pe13
             &quadspi_bk1_io2_pe14 &quadspi_bk1_io3_pe15>;
```

These properties have type `phandles`.

Zero or more nodes with metadata: phandle-array type You can use phandles to refer to and configure one or more resources that are "owned" by some other node.

This is the most complex case. There are examples and more details in the next section.

These properties have type `phandle-array`.

phandle-array properties These properties are commonly used to specify a resource that is owned by another node along with additional metadata about the resource.

High level description Usually, properties with this type are written like `phandle-array-prop` in this example:

```
node {
        phandle-array-prop = <&foo 1 2>, <&bar 3>, <&baz 4 5>;
};
```

That is, the property's value is written as a comma-separated sequence of "groups", where each "group" is written inside of angle brackets (`< ... >`). Each "group" starts with a phandle (`&foo`, `&bar`, `&baz`). The values that follow the phandle in each "group" are called *specifiers*. There are three specifiers in the above example:

1. `1 2`
2. `3`
3. `4 5`

The phandle in each "group" is used to "point" to the hardware that controls the resource you are interested in. The specifier describes the resource itself, along with any additional necessary metadata.

The rest of this section describes a common example. Subsequent sections document more rules about how to use phandle-array properties in practice.

Example phandle-arrays: GPIOs Perhaps the most common use case for phandle-array properties is specifying one or more GPIOs on your SoC that another chip on your board connects to. For that reason, we'll focus on that use case here. However, there are **many other use cases** that are handled in devicetree with phandle-array properties.

For example, consider an external chip with an interrupt pin that is connected to a GPIO on your SoC. You will typically need to provide that GPIO's information (GPIO controller and pin number) to the *device driver* for that chip. You usually also need to provide other metadata about the GPIO, like whether it is active low or high, what kind of internal pull resistor within the SoC should be enabled in order to communicate with the device, etc., to the driver.

In the devicetree, there will be a node that represents the GPIO controller that controls a group of pins. This reflects the way GPIO IP blocks are usually developed in hardware. Therefore, there is no single node in the devicetree that represents a GPIO pin, and you can't use a single phandle to represent it.

Instead, you would use a phandle-array property, like this:

```
my-external-ic {
        irq-gpios = <&gpioX pin flags>;
};
```

In this example, `irq-gpios` is a phandle-array property with just one "group" in its value. `&gpioX` is the phandle for the GPIO controller node that controls the pin. `pin` is the pin number (0, 1, 2, ...). `flags` is a bit mask describing pin metadata (for example `(GPIO_ACTIVE_LOW | GPIO_PULL_UP)`); see include/zephyr/dt-bindings/gpio/gpio.h for more details.

The device driver handling the `my-external-ic` node can then use the `irq-gpios` property's value to set up interrupt handling for the chip as it is used on your board. This lets you configure the device driver in devicetree, without changing the driver's source code.

Such properties can contain multiple values as well:

```
my-other-external-ic {
        handshake-gpios = <&gpioX pinX flagsX>, <&gpioY pinY flagsY>;
};
```

The above example specifies two pins:

- `pinX` on the GPIO controller with phandle `&gpioX`, flags `flagsX`
- `pinY` on `&gpioY`, flags `flagsY`

You may be wondering how the "pin and flags" convention is established and enforced. To answer this question, we'll need to introduce a concept called specifier spaces before moving on to some information about devicetree bindings.

Specifier spaces *Specifier spaces* are a way to allow nodes to describe how you should use them in phandle-array properties.

We'll start with an abstract, high level description of how specifier spaces work in DTS files, before moving on to a concrete example and providing references to further reading for how this all works in practice using DTS files and bindings files.

High level description As described above, a phandle-array property is a sequence of "groups" of phandles followed by some number of cells:

```
node {
        phandle-array-prop = <&foo 1 2>, <&bar 3>;
};
```

The cells that follow each phandle are called a *specifier*. In this example, there are two specifiers:

1. `1 2`: two cells
2. `3`: one cell

Every phandle-array property has an associated *specifier space*. This sounds complex, but it's really just a way to assign a meaning to the cells that follow each phandle in a hardware specific way. Every specifier space has a unique name. There are a few "standard" names for commonly used hardware, but you can create your own as well.

Devicetree nodes encode the number of cells that must appear in a specifier, by name, using the `#SPACE_NAME-cells` property. For example, let's assume that `phandle-array-prop`'s specifier

space is named baz. Then we would need the foo and bar nodes to have the following #baz-cells properties:

```
foo: node@1000 {
        #baz-cells = <2>;
};

bar: node@2000 {
        #baz-cells = <1>;
};
```

Without the #baz-cells property, the devicetree tooling would not be able to validate the number of cells in each specifier in phandle-array-prop.

This flexibility allows you to write down an array of hardware resources in a single devicetree property, even though the amount of metadata you need to describe each resource might be different for different nodes.

A single node can also have different numbers of cells in different specifier spaces. For example, we might have:

```
foo: node@1000 {
        #baz-cells = <2>;
        #bob-cells = <1>;
};
```

With that, if phandle-array-prop-2 has specifier space bob, we could write:

```
node {
        phandle-array-prop = <&foo 1 2>, <&bar 3>;
        phandle-array-prop-2 = <&foo 4>;
};
```

This flexibility allows you to have a node that manages multiple different kinds of resources at the same time. The node describes the amount of metadata needed to describe each kind of resource (how many cells are needed in each case) using different #SPACE_NAME-cells properties.

Example specifier space: gpio From the above example, you're already familiar with how one specifier space works: in the "gpio" space, specifiers almost always have two cells:

1. a pin number
2. a bit mask of flags related to the pin

Therefore, almost all GPIO controller nodes you will see in practice will look like this:

```
gpioX: gpio-controller@deadbeef {
        gpio-controller;
        #gpio-cells = <2>;
};
```

Associating properties with specifier spaces Above, we have described that:

- each phandle-array property has an associated specifier space
- specifier spaces are identified by name
- devicetree nodes use #SPECIFIER_NAME-cells properties to configure the number of cells which must appear in a specifier

In this section, we explain how phandle-array properties get their specifier spaces.

High level description In general, a `phandle-array` property named `foos` implicitly has specifier space `foo`. For example:

```
properties:
  dmas:
    type: phandle-array
  pwms:
    type: phandle-array
```

The `dmas` property's specifier space is "dma". The `pwms` property's specifier space is `pwm`.

Special case: GPIOs and IO channels `*-gpios` and `*-io-channels` properties are special-cased so that e.g. `foo-gpios` and `bar-io-channels` resolve to `#gpio-cells` and `#io-channel-cells` respectively, rather than `#foo-gpio-cells` and `#bar-io-channel-cells`.

Manually specifying a space You can manually specify the specifier space for any `phandle-array` property. See *specifier-space*.

Naming the cells in a specifier You should name the cells in each specifier space your hardware supports when writing bindings. For details on how to do this, see *Specifier cell names (*-cells)*.

This allows C code to query information about and retrieve the values of cells in a specifier by name using devicetree APIs like these:

- `DT_PHA_BY_IDX`
- `DT_PHA_BY_NAME`

This feature and these macros are used internally by numerous hardware-specific APIs. Here are a few examples:

- `DT_GPIO_PIN_BY_IDX`
- `DT_PWMS_CHANNEL_BY_IDX`
- `DT_DMAS_CELL_BY_NAME`
- `DT_IO_CHANNELS_INPUT_BY_IDX`
- `DT_CLOCKS_CELL_BY_NAME`

See also

- *Writing property values*: how to write phandles in devicetree properties
- *Properties*: how to write bindings for properties with phandle types (`phandle`, `phandles`, `phandle-array`)
- *specifier-space*: how to manually specify a phandle-array property's specifier space
- *dependency-mode*: how to control node dependencies using phandle properties

The `/zephyr,user` node

Zephyr's devicetree scripts handle the `/zephyr,user` node as a special case: you can put essentially arbitrary properties inside it and retrieve their values without having to write a binding. It is meant as a convenient container when only a few simple properties are needed. The type is inferred from the values assigned to the property.

> **Note**
>
> This node is meant for sample code and user applications. It should not be used in the upstream Zephyr source code for device drivers, subsystems, etc.

Simple values You can store numeric or array values in `/zephyr,user` if you want them to be configurable at build time via devicetree.

For example, with this devicetree overlay:

```
/ {
     zephyr,user {
             boolean;
             bytes = [81 82 83];
             number = <23>;
             numbers = <1>, <2>, <3>;
             string = "text";
             strings = "a", "b", "c";
     };
};
```

You can get the above property values in C/C++ code like this:

```
#define ZEPHYR_USER_NODE DT_PATH(zephyr_user)

DT_PROP(ZEPHYR_USER_NODE, boolean) // 1
DT_PROP(ZEPHYR_USER_NODE, bytes)   // {0x81, 0x82, 0x83}
DT_PROP(ZEPHYR_USER_NODE, number)  // 23
DT_PROP(ZEPHYR_USER_NODE, numbers) // {1, 2, 3}
DT_PROP(ZEPHYR_USER_NODE, string)  // "text"
DT_PROP(ZEPHYR_USER_NODE, strings) // {"a", "b", "c"}
```

Devices You can store *phandles* in `/zephyr,user` if you want to be able to reconfigure which devices your application uses in simple cases using devicetree overlays.

For example, with this devicetree overlay:

```
/ {
     zephyr,user {
             handle = <&gpio0>;
             handles = <&gpio0>, <&gpio1>;
     };
};
```

You can convert the phandles in the `handle` and `handles` properties to device pointers like this:

```
/*
 * Same thing as:
 *
 * ... my_dev = DEVICE_DT_GET(DT_NODELABEL(gpio0));
 */
const struct device *my_device =
     DEVICE_DT_GET(DT_PROP(ZEPHYR_USER_NODE, handle));

/*
 * Same thing as:
 *
```

(continues on next page)

(continued from previous page)

```
 * ... *my_devices[] = {
 *         DEVICE_DT_GET(DT_NODELABEL(gpio0)),
 *         DEVICE_DT_GET(DT_NODELABEL(gpio1))
 * };
 */
const struct device *my_devices[] = {
     DT_FOREACH_PROP_ELEM_SEP(ZEPHYR_USER_NODE, handles, DEVICE_DT_GET_BY_IDX, (,))
};
```

GPIOs The `/zephyr,user` node is a convenient place to store application-specific GPIOs that you want to be able to reconfigure with a devicetree overlay.

> **Note**
>
> All properties with a value containing at least one phandle and a number will be inferred as a phandle-array. example `<&adc0 1>`.

For example, with this devicetree overlay:

```
#include <zephyr/dt-bindings/gpio/gpio.h>

/ {
     zephyr,user {
             signal-gpios = <&gpio0 1 GPIO_ACTIVE_HIGH>;
     };
};
```

You can convert the pin defined in `signal-gpios` to a `struct gpio_dt_spec` in your source code, then use it like this:

```
#include <zephyr/drivers/gpio.h>

#define ZEPHYR_USER_NODE DT_PATH(zephyr_user)

const struct gpio_dt_spec signal =
        GPIO_DT_SPEC_GET(ZEPHYR_USER_NODE, signal_gpios);

/* Configure the pin */
gpio_pin_configure_dt(&signal, GPIO_OUTPUT_INACTIVE);

/* Set the pin to its active level */
gpio_pin_set_dt(&signal, 1);
```

(See `gpio_dt_spec`, `GPIO_DT_SPEC_GET`, and `gpio_pin_configure_dt()` for details on these APIs.)

Devicetree HOWTOs

This page has step-by-step advice for getting things done with devicetree.

> **Tip**
>
> See *Troubleshooting devicetree* for troubleshooting advice.

Get your devicetree and generated header A board's devicetree (*BOARD.dts*) pulls in common node definitions via `#include` preprocessor directives. This at least includes the SoC's `.dtsi`. One way to figure out the devicetree's contents is by opening these files, e.g. by looking in `dts/<ARCH>/<vendor>/<soc>.dtsi`, but this can be time consuming.

If you just want to see the "final" devicetree for your board, build an application and open the `zephyr.dts` file in the build directory.

> **Tip**
>
> You can build hello_world to see the "base" devicetree for your board without any additional changes from *overlay files*.

For example, using the qemu_cortex_m3 board to build hello_world:

```
# --cmake-only here just forces CMake to run, skipping the
# build process to save time.
west build -b qemu_cortex_m3 samples/hello_world --cmake-only
```

You can change `qemu_cortex_m3` to match your board.

CMake prints the input and output file locations like this:

```
-- Found BOARD.dts: .../zephyr/boards/arm/qemu_cortex_m3/qemu_cortex_m3.dts
-- Generated zephyr.dts: .../zephyr/build/zephyr/zephyr.dts
-- Generated devicetree_generated.h: .../
↪zephyr/build/zephyr/include/generated/zephyr/devicetree_generated.h
```

The `zephyr.dts` file is the final devicetree in DTS format.

The `devicetree_generated.h` file is the corresponding generated header.

See *Input and output files* for details about these files.

Get a struct device from a devicetree node When writing Zephyr applications, you'll often want to get a driver-level *struct device* corresponding to a devicetree node.

For example, with this devicetree fragment, you might want the struct device for `serial@40002000`:

```
/ {
        soc {
                serial0: serial@40002000 {
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };

        aliases {
                my-serial = &serial0;
        };

        chosen {
                zephyr,console = &serial0;
        };
};
```

Start by making a *node identifier* for the device you are interested in. There are different ways to do this; pick whichever one works best for your requirements. Here are some examples:

```
/* Option 1: by node label */
#define MY_SERIAL DT_NODELABEL(serial0)

/* Option 2: by alias */
#define MY_SERIAL DT_ALIAS(my_serial)

/* Option 3: by chosen node */
#define MY_SERIAL DT_CHOSEN(zephyr_console)

/* Option 4: by path */
#define MY_SERIAL DT_PATH(soc, serial_40002000)
```

Once you have a node identifier there are two ways to proceed. One way to get a device is to use `DEVICE_DT_GET`:

```
const struct device *const uart_dev = DEVICE_DT_GET(MY_SERIAL);

if (!device_is_ready(uart_dev)) {
        /* Not ready, do not use */
        return -ENODEV;
}
```

There are variants of `DEVICE_DT_GET` such as `DEVICE_DT_GET_OR_NULL`, `DEVICE_DT_GET_ONE` or `DEVICE_DT_GET_ANY`. This idiom fetches the device pointer at build-time, which means there is no runtime penalty. This method is useful if you want to store the device pointer as configuration data. But because the device may not be initialized, or may have failed to initialize, you must verify that the device is ready to be used before passing it to any API functions. (This check is done for you by `device_get_binding()`.)

In some situations the device cannot be known at build-time, e.g., if it depends on user input like in a shell application. In this case you can get the `struct device` by combining `device_get_binding()` with the device name:

```
const char *dev_name = /* TODO: insert device name from user */;
const struct device *uart_dev = device_get_binding(dev_name);
```

You can then use `uart_dev` with *Universal Asynchronous Receiver-Transmitter (UART)* API functions like `uart_configure()`. Similar code will work for other device types; just make sure you use the correct API for the device.

If you're having trouble, see *Troubleshooting devicetree*. The first thing to check is that the node has `status = "okay"`, like this:

```
#define MY_SERIAL DT_NODELABEL(my_serial)

#if DT_NODE_HAS_STATUS(MY_SERIAL, okay)
const struct device *const uart_dev = DEVICE_DT_GET(MY_SERIAL);
#else
#error "Node is disabled"
#endif
```

If you see the `#error` output, make sure to enable the node in your devicetree. In some situations your code will compile but it will fail to link with a message similar to:

```
...undefined reference to `__device_dts_ord_N'
collect2: error: ld returned 1 exit status
```

This likely means there's a Kconfig issue preventing the device driver from being built, resulting in a reference that does not exist. If your code compiles successfully, the last thing to check is if the device is ready, like this:

```
if (!device_is_ready(uart_dev)) {
    printk("Device not ready\n");
}
```

If you find that the device is not ready, it likely means that the device's initialization function failed. Enabling logging or debugging driver code may help in such situations. Note that you can also use `device_get_binding()` to obtain a reference at runtime. If it returns `NULL` it can either mean that device's driver failed to initialize or that it does not exist.

Find a devicetree binding *Devicetree bindings* are YAML files which declare what you can do with the nodes they describe, so it's critical to be able to find them for the nodes you are using.

If you don't have them already, *Get your devicetree and generated header*. To find a node's binding, open the generated header file, which starts with a list of nodes in a block comment:

```
/*
 * [...]
 * Nodes in dependency order (ordinal and path):
 *   0   /
 *   1   /aliases
 *   2   /chosen
 *   3   /flash@0
 *   4   /memory@20000000
 *          (etc.)
 * [...]
 */
```

Make note of the path to the node you want to find, like `/flash@0`. Search for the node's output in the file, which starts with something like this if the node has a matching binding:

```
/*
 * Devicetree node:
 *   /flash@0
 *
 * Binding (compatible = soc-nv-flash):
 *   $ZEPHYR_BASE/dts/bindings/mtd/soc-nv-flash.yaml
 * [...]
 */
```

See *Check for missing bindings* for troubleshooting.

Set devicetree overlays Devicetree overlays are explained in *Introduction to devicetree*. The CMake variable **DTC_OVERLAY_FILE** contains a space- or semicolon-separated list of overlay files to use. If **DTC_OVERLAY_FILE** specifies multiple files, they are included in that order by the C preprocessor. A file in a Zephyr module can be referred to by escaping the Zephyr module dir variable like `\${ZEPHYR_<module>_MODULE_DIR}/<path-to>/dts.overlay` when setting the DTC_OVERLAY_FILE variable.

You can set **DTC_OVERLAY_FILE** to contain exactly the files you want to use. Here is an *example* using `west build`.

If you don't set **DTC_OVERLAY_FILE**, the build system will follow these steps, looking for files in your application configuration directory to use as devicetree overlays:

1. If the file `socs/<SOC>_<BOARD_QUALIFIERS>.overlay` exists, it will be used.
2. If the file `boards/<BOARD>.overlay` exists, it will be used in addition to the above.
3. If the current board has *multiple revisions* and `boards/<BOARD>_<revision>.overlay` exists, it will be used in addition to the above.

4. If one or more files have been found in the previous steps, the build system stops looking and just uses those files.
5. Otherwise, if `<BOARD>.overlay` exists, it will be used, and the build system will stop looking for more files.
6. Otherwise, if `app.overlay` exists, it will be used.

Extra devicetree overlays may be provided using `EXTRA_DTC_OVERLAY_FILE` which will still allow the build system to automatically use devicetree overlays described in the above steps.

The build system appends overlays specified in `EXTRA_DTC_OVERLAY_FILE` to the overlays in `DTC_OVERLAY_FILE` when processing devicetree overlays. This means that changes made via `EXTRA_DTC_OVERLAY_FILE` have higher precedence than those made via `DTC_OVERLAY_FILE`.

All configuration files will be taken from the application's configuration directory except for files with an absolute path that are given with the `DTC_OVERLAY_FILE` or `EXTRA_DTC_OVERLAY_FILE` argument.

See *Application Configuration Directory* on how the application configuration directory is defined.

Using *Shields* will also add devicetree overlay files.

The **`DTC_OVERLAY_FILE`** value is stored in the CMake cache and used in successive builds.

The *build system* prints all the devicetree overlays it finds in the configuration phase, like this:

```
-- Found devicetree overlay: .../some/file.overlay
```

Use devicetree overlays See *Set devicetree overlays* for how to add an overlay to the build.

Overlays can override node property values in multiple ways. For example, if your BOARD.dts contains this node:

```
/ {
        soc {
                serial0: serial@40002000 {
                        status = "okay";
                        current-speed = <115200>;
                        /* ... */
                };
        };
};
```

These are equivalent ways to override the `current-speed` value in an overlay:

```
/* Option 1 */
&serial0 {
     current-speed = <9600>;
};

/* Option 2 */
&{/soc/serial@40002000} {
     current-speed = <9600>;
};
```

We'll use the `&serial0` style for the rest of these examples.

You can add aliases to your devicetree using overlays: an alias is just a property of the `/aliases` node. For example:

```
/ {
     aliases {
             my-serial = &serial0;
     };
};
```

Chosen nodes work the same way. For example:

```
/ {
     chosen {
             zephyr,console = &serial0;
     };
};
```

To delete a property (in addition to deleting properties in general, this is how to set a boolean property to false if it's true in BOARD.dts):

```
&serial0 {
     /delete-property/ some-unwanted-property;
};
```

You can add subnodes using overlays. For example, to configure a SPI or I2C child device on an existing bus node, do something like this:

```
/* SPI device example */
&spi1 {
     my_spi_device: temp-sensor@0 {
             compatible = "...";
             label = "TEMP_SENSOR_0";
             /* reg is the chip select number, if needed;
              * If present, it must match the node's unit address. */
             reg = <0>;

             /* Configure other SPI device properties as needed.
              * Find your device's DT binding for details. */
             spi-max-frequency = <DT_FREQ_M(4)>;
     };
};

/* I2C device example */
&i2c2 {
     my_i2c_device: touchscreen@76 {
             compatible = "...";
             label = "TOUCHSCREEN";
             /* reg is the I2C device address.
              * It must match the node's unit address. */
             reg = <76>;

             /* Configure other I2C device properties as needed.
              * Find your device's DT binding for details. */
     };
};
```

Other bus devices can be configured similarly:

- create the device as a subnode of the parent bus
- set its properties according to its binding

Assuming you have a suitable device driver associated with the `my_spi_device` and

my_i2c_device compatibles, you should now be able to enable the driver via Kconfig and *get the struct device* for your newly added bus node, then use it with that driver API.

Write device drivers using devicetree APIs "Devicetree-aware" *device drivers* should create a struct device for each status = "okay" devicetree node with a particular *compatible* (or related set of compatibles) supported by the driver.

Writing a devicetree-aware driver begins by defining a *devicetree binding* for the devices supported by the driver. Use existing bindings from similar drivers as a starting point. A skeletal binding to get started needs nothing more than this:

```
description: <Human-readable description of your binding>
compatible: "foo-company,bar-device"
include: base.yaml
```

See *Find a devicetree binding* for more advice on locating existing bindings.

After writing your binding, your driver C file can then use the devicetree API to find status = "okay" nodes with the desired compatible, and instantiate a struct device for each one. There are two options for instantiating each struct device: using instance numbers, and using node labels.

In either case:

- Each struct device's name should be set to its devicetree node's label property. This allows the driver's users to *Get a struct device from a devicetree node* in the usual way.
- Each device's initial configuration should use values from devicetree properties whenever practical. This allows users to configure the driver using *devicetree overlays*.

Examples for how to do this follow. They assume you've already implemented the device-specific configuration and data structures and API functions, like this:

```
/* my_driver.c */
#include <zephyr/drivers/some_api.h>

/* Define data (RAM) and configuration (ROM) structures: */
struct my_dev_data {
     /* per-device values to store in RAM */
};
struct my_dev_cfg {
     uint32_t freq; /* Just an example: initial clock frequency in Hz */
     /* other configuration to store in ROM */
};

/* Implement driver API functions (drivers/some_api.h callbacks): */
static int my_driver_api_func1(const struct device *dev, uint32_t *foo) { /* ...␣
↪*/ }
static int my_driver_api_func2(const struct device *dev, uint64_t bar) { /* ... */␣
↪}
static struct some_api my_api_funcs = {
     .func1 = my_driver_api_func1,
     .func2 = my_driver_api_func2,
};
```

Option 1: create devices using instance numbers Use this option, which uses *Instance-based APIs*, if possible. However, they only work when devicetree nodes for your driver's compatible are all equivalent, and you do not need to be able to distinguish between them.

To use instance-based APIs, begin by defining DT_DRV_COMPAT to the lowercase-and-underscores version of the compatible that the device driver supports. For example, if your driver's compati-

ble is "vnd,my-device" in devicetree, you would define DT_DRV_COMPAT to vnd_my_device in your driver C file:

```
/*
 * Put this near the top of the file. After the includes is a good place.
 * (Note that you can therefore run "git grep DT_DRV_COMPAT drivers" in
 * the zephyr Git repository to look for example drivers using this style).
 */
#define DT_DRV_COMPAT vnd_my_device
```

Important

As shown, the DT_DRV_COMPAT macro should have neither quotes nor special characters. Remove quotes and convert special characters to underscores when creating DT_DRV_COMPAT from the compatible property.

Finally, define an instantiation macro, which creates each struct device using instance numbers. Do this after defining my_api_funcs.

```
/*
 * This instantiation macro is named "CREATE_MY_DEVICE".
 * Its "inst" argument is an arbitrary instance number.
 *
 * Put this near the end of the file, e.g. after defining "my_api_funcs".
 */
#define CREATE_MY_DEVICE(inst)                                          \
     static struct my_dev_data my_data_##inst = {                       \
             /* initialize RAM values as needed, e.g.: */               \
             .freq = DT_INST_PROP(inst, clock_frequency),               \
     };                                                                 \
     static const struct my_dev_cfg my_cfg_##inst = {                   \
             /* initialize ROM values as needed. */                     \
     };                                                                 \
     DEVICE_DT_INST_DEFINE(inst,                                        \
                           my_dev_init_function,                        \
                           NULL,                                        \
                           &my_data_##inst,                             \
                           &my_cfg_##inst,                              \
                           MY_DEV_INIT_LEVEL, MY_DEV_INIT_PRIORITY,     \
                           &my_api_funcs);
```

Notice the use of APIs like DT_INST_PROP and DEVICE_DT_INST_DEFINE to access devicetree node data. These APIs retrieve data from the devicetree for instance number inst of the node with compatible determined by DT_DRV_COMPAT.

Finally, pass the instantiation macro to DT_INST_FOREACH_STATUS_OKAY:

```
/* Call the device creation macro for each instance: */
DT_INST_FOREACH_STATUS_OKAY(CREATE_MY_DEVICE)
```

DT_INST_FOREACH_STATUS_OKAY expands to code which calls CREATE_MY_DEVICE once for each enabled node with the compatible determined by DT_DRV_COMPAT. It does not append a semicolon to the end of the expansion of CREATE_MY_DEVICE, so the macro's expansion must end in a semicolon or function definition to support multiple devices.

Option 2: create devices using node labels Some device drivers cannot use instance numbers. One example is an SoC peripheral driver which relies on vendor HAL APIs specialized for individual IP blocks to implement Zephyr driver callbacks. Cases like this should use

DT_NODELABEL to refer to individual nodes in the devicetree representing the supported peripherals on the SoC. The devicetree.h *Generic APIs* can then be used to access node data.

For this to work, your *SoC's dtsi file* must define node labels like mydevice0, mydevice1, etc. appropriately for the IP blocks your driver supports. The resulting devicetree usually looks something like this:

```
/ {
        soc {
                mydevice0: dev@0 {
                        compatible = "vnd,my-device";
                };
                mydevice1: dev@1 {
                        compatible = "vnd,my-device";
                };
        };
};
```

The driver can use the mydevice0 and mydevice1 node labels in the devicetree to operate on specific device nodes:

```
/*
 * This is a convenience macro for creating a node identifier for
 * the relevant devices. An example use is MYDEV(0) to refer to
 * the node with label "mydevice0".
 */
#define MYDEV(idx) DT_NODELABEL(mydevice ## idx)

/*
 * Define your instantiation macro; "idx" is a number like 0 for mydevice0
 * or 1 for mydevice1. It uses MYDEV() to create the node label from the
 * index.
 */
#define CREATE_MY_DEVICE(idx)                                          \
     static struct my_dev_data my_data_##idx = {                       \
             /* initialize RAM values as needed, e.g.: */              \
             .freq = DT_PROP(MYDEV(idx), clock_frequency),             \
     };                                                                \
     static const struct my_dev_cfg my_cfg_##idx = { /* ... */ };      \
     DEVICE_DT_DEFINE(MYDEV(idx),                                      \
                      my_dev_init_function,                            \
                      NULL,                                            \
                      &my_data_##idx,                                  \
                      &my_cfg_##idx,                                   \
                      MY_DEV_INIT_LEVEL, MY_DEV_INIT_PRIORITY,         \
                      &my_api_funcs)
```

Notice the use of APIs like DT_PROP and DEVICE_DT_DEFINE to access devicetree node data.

Finally, manually detect each enabled devicetree node and use CREATE_MY_DEVICE to instantiate each struct device:

```
#if DT_NODE_HAS_STATUS(DT_NODELABEL(mydevice0), okay)
CREATE_MY_DEVICE(0)
#endif

#if DT_NODE_HAS_STATUS(DT_NODELABEL(mydevice1), okay)
CREATE_MY_DEVICE(1)
#endif
```

Since this style does not use `DT_INST_FOREACH_STATUS_OKAY()`, the driver author is responsible for calling `CREATE_MY_DEVICE()` for every possible node, e.g. using knowledge about the peripherals available on supported SoCs.

Device drivers that depend on other devices At times, one `struct device` depends on another `struct device` and requires a pointer to it. For example, a sensor device might need a pointer to its SPI bus controller device. Some advice:

- Write your devicetree binding in a way that permits use of *Hardware specific APIs* from devicetree.h if possible.
- In particular, for bus devices, your driver's binding should include a file like dts/bindings/spi/spi-device.yaml which provides common definitions for devices addressable via a specific bus. This enables use of APIs like `DT_BUS` to obtain a node identifier for the bus node. You can then *Get a struct device from a devicetree node* for the bus in the usual way.

Search existing bindings and device drivers for examples.

Applications that depend on board-specific devices One way to allow application code to run unmodified on multiple boards is by supporting a devicetree alias to specify the hardware specific portions, as is done in the blinky sample. The application can then be configured in *BOARD.dts* files or via *devicetree overlays*.

Troubleshooting devicetree

Here are some tips for fixing misbehaving devicetree related code.

See *Devicetree HOWTOs* for other "HOWTO" style information.

> **Important**
>
> Try this first, before doing anything else.

Try again with a pristine build directory See *Pristine Builds* for examples, or just delete the build directory completely and retry.

This is general advice which is especially applicable to debugging devicetree issues, because the outputs are created during the CMake configuration phase, and are not always regenerated when one of their inputs changes.

Use Devicetree diagnostics tool (DT Doctor) An optional Devicetree diagnostics tool is available to help diagnose Devicetree issues. It can be enabled by passing the `-DZEPHYR_SCA_VARIANT=dtdoctor` argument to *west build*.

See *Devicetree diagnostics (dtdoctor)* for more information.

Make sure <devicetree.h> is included Unlike Kconfig symbols, the `devicetree.h` header must be included explicitly.

Many Zephyr header files rely on information from devicetree, so including some other API may transitively include `devicetree.h`, but that's not guaranteed.

undefined reference to `__device_dts_ord_<N>` This usually happens on a line like this:

```
const struct device *dev = DEVICE_DT_GET(NODE_ID);
```

where NODE_ID is a valid *node identifier*, but no device driver has allocated a struct device for this devicetree node. You thus get a linker error, because you're asking for a pointer to a device that isn't defined.

To fix it, you need to make sure that:

1. The node is enabled: the node must have status = "okay";.

   (Recall that a missing status property means the same thing as status = "okay";; see *Important properties* for more information about status).

2. A device driver responsible for allocating the struct device is enabled. That is, the Kconfig option which makes the build system compile the driver sources into your application needs to be set to y.

   (See *Setting Kconfig configuration values* for more information on setting Kconfig options.)

Below, <build> means your build directory.

Making sure the node is enabled:

To find the devicetree node you need to check, use the number <N> from the linker error. Look for this number in the list of nodes at the top of <build>/zephyr/include/generated/zephyr/devicetree_generated.h. For example, if <N> is 15, and your devicetree_generated.h file looks like this, the node you are interested in is /soc/i2c@deadbeef:

```
/*
 * Generated by gen_defines.py
 *
 * DTS input file:
 *   zephyr.dts.pre
 *
 * Directories with bindings:
 *   $ZEPHYR_BASE/dts/bindings
 *
 * Node dependency ordering (ordinal and path):
 *   0   /
 *   1   /aliases
[...]
 *   15  /soc/i2c@deadbeef
[...]
```

Now look for this node in <build>/zephyr/zephyr.dts, which is the final devicetree for your application build. (See *Get your devicetree and generated header* for information and examples.)

If the node has status = "disabled"; in zephyr.dts, then you need to enable it by setting status = "okay";, probably by using a devicetree *overlay*. For example, if zephyr.dts looks like this:

```
i2c0: i2c@deadbeef {
        status = "disabled";
};
```

Then you should put this into your devicetree overlay and *Try again with a pristine build directory*:

```
&i2c0 {
        status = "okay";
};
```

Make sure that you see status = "okay"; in zephyr.dts after you rebuild.

Making sure the device driver is enabled:

The first step is to figure out which device driver is responsible for handling your devicetree node and allocating devices for it. To do this, you need to start with the `compatible` property in your devicetree node, and find the driver that allocates `struct device` instances for that compatible.

If you're not familiar with how devices are allocated from devicetree nodes based on compatible properties, the ZDS 2021 talk A deep dive into the Zephyr 2.5 device model may be a useful place to start, along with the *Device Driver Model* pages. See *Important properties* and the Devicetree specification for more information about `compatible`.

There is currently no documentation for what device drivers exist and which devicetree compatibles they are associated with. You will have to figure this out by reading the source code:

- Look in drivers for the appropriate subdirectory that corresponds to the API your device implements
- Look inside that directory for relevant files until you figure out what the driver is, or realize there is no such driver.

Often, but not always, you can find the driver by looking for a file that sets the `DT_DRV_COMPAT` macro to match your node's `compatible` property, except lowercased and with special characters converted to underscores. For example, if your node's compatible is `vnd,foo-device`, look for a file with this line:

```
#define DT_DRV_COMPAT vnd_foo_device
```

Important

This **does not always work** since not all drivers use `DT_DRV_COMPAT`.

If you find a driver, you next need to make sure the Kconfig option that compiles it is enabled. (If you don't find a driver, and you are sure the compatible property is correct, then you need to write a driver. Writing drivers is outside the scope of this documentation page.)

Continuing the above example, if your devicetree node looks like this now:

```
i2c0: i2c@deadbeef {
        compatible = "nordic,nrf-twim";
        status = "okay";
};
```

Then you would look inside of drivers/i2c for the driver file that handles the compatible `nordic,nrf-twim`. In this case, that is drivers/i2c/i2c_nrfx_twim.c. Notice how even in cases where `DT_DRV_COMPAT` is not set, you can use information like driver file names as clues.

Once you know the driver you want to enable, you need to make sure its Kconfig option is set to `y`. You can figure out which Kconfig option is needed by looking for a line similar to this one in the `CMakeLists.txt` file in the drivers subdirectory. Continuing the above example, drivers/i2c/CMakeLists.txt has a line that looks like this:

```
zephyr_library_sources_ifdef(CONFIG_NRFX_TWIM       i2c_nrfx_twim.c)
```

This means that `CONFIG_NRFX_TWIM` must be set to `y` in `<build>/zephyr/.config` file.

If your driver's Kconfig is not set to `y`, you need to figure out what you need to do to make that happen. Often, this will happen automatically as soon as you enable the devicetree node. Otherwise, it is sometimes as simple as adding a line like this to your application's `prj.conf` file and then making sure to *Try again with a pristine build directory*:

```
CONFIG_FOO=y
```

where `CONFIG_FOO` is the option that `CMakeLists.txt` uses to decide whether or not to compile the driver.

However, there may be other problems in your way, such as unmet Kconfig dependencies that you also have to enable before you can enable your driver.

Consult the Kconfig file that defines `CONFIG_FOO` (for your value of `FOO`) for more information.

Make sure you're using the right names Remember that:

- In C/C++, devicetree names must be lowercased and special characters must be converted to underscores. Zephyr's generated devicetree header has DTS names converted in this way into the C tokens used by the preprocessor-based `<devicetree.h>` API.
- In overlays, use devicetree node and property names the same way they would appear in any DTS file. Zephyr overlays are just DTS fragments.

For example, if you're trying to **get** the `clock-frequency` property of a node with path `/soc/i2c@12340000` in a C/C++ file:

```
/*
 * foo.c: lowercase-and-underscores names
 */

/* Don't do this: */
#define MY_CLOCK_FREQ DT_PROP(DT_PATH(soc, i2c@1234000), clock-frequency)
/*                                        ^               ^
 *                                    @ should be _     - should be _  */

/* Do this instead: */
#define MY_CLOCK_FREQ DT_PROP(DT_PATH(soc, i2c_1234000), clock_frequency)
/*                                        ^               ^           */
```

And if you're trying to **set** that property in a devicetree overlay:

```
/*
 * foo.overlay: DTS names with special characters, etc.
 */

/* Don't do this; you'll get devicetree errors. */
&{/soc/i2c_12340000/} {
     clock_frequency = <115200>;
};

/* Do this instead. Overlays are just DTS fragments. */
&{/soc/i2c@12340000/} {
     clock-frequency = <115200>;
};
```

Look at the preprocessor output To save preprocessor output files, enable the `CONFIG_COMPILER_SAVE_TEMPS` option. For example, to build hello_world with west with this option set, use:

```
west build -b BOARD samples/hello_world -- -DCONFIG_COMPILER_SAVE_TEMPS=y
```

This will create a preprocessor output file named `foo.c.i` in the build directory for each source file `foo.c`.

You can then search for the file in the build directory to see what your devicetree macros expanded to. For example, on macOS and Linux, using `find` to find `main.c.i`:

```
$ find build -name main.c.i
build/CMakeFiles/app.dir/src/main.c.i
```

It's usually easiest to run a style formatter on the results before opening them. For example, to use `clang-format` to reformat the file in place:

```
clang-format -i build/CMakeFiles/app.dir/src/main.c.i
```

You can then open the file in your favorite editor to view the final C results after preprocessing.

Do not track macro expansion Compiler messages for devicetree errors can sometimes be very long. This typically happens when the compiler prints a message for every step of a complex macro expansion that has several intermediate expansion steps.

To prevent the compiler from doing this, you can disable the `CONFIG_COMPILER_TRACK_MACRO_EXPANSION` option. This typically reduces the output to one message per error.

For example, to build hello_world with west and this option disabled, use:

```
west build -b BOARD samples/hello_world -- -
↪DCONFIG_COMPILER_TRACK_MACRO_EXPANSION=n
```

Validate properties If you're getting a compile error reading a node property, check your node identifier and property. For example, if you get a build error on a line that looks like this:

```
int baud_rate = DT_PROP(DT_NODELABEL(my_serial), current_speed);
```

Try checking the node by adding this to the file and recompiling:

```
#if !DT_NODE_EXISTS(DT_NODELABEL(my_serial))
#error "whoops"
#endif
```

If you see the "whoops" error message when you rebuild, the node identifier isn't referring to a valid node. *Get your devicetree and generated header* and debug from there.

Some hints for what to check next if you don't see the "whoops" error message:

- did you *Make sure you're using the right names*?
- does the *property exist*?
- does the node have a *matching binding*?
- does the binding define the property?

Check for missing bindings See *Devicetree bindings* for information about bindings, and *Bindings index* for information on bindings built into Zephyr.

If the build fails to *Find a devicetree binding* for a node, then either the node's `compatible` property is not defined, or its value has no matching binding. If the property is set, check for typos in its name. In a devicetree source file, `compatible` should look like `"vnd,some-device"` – *Make sure you're using the right names*.

If your binding file is not under `zephyr/dts`, you may need to set *DTS_ROOT*; see *Where bindings are located*.

Errors with DT_INST_() APIs If you're using an API like `DT_INST_PROP()`, you must define `DT_DRV_COMPAT` to the lowercase-and-underscores version of the compatible you are interested in. See *Option 1: create devices using instance numbers*.

Devicetree versus Kconfig

Along with devicetree, Zephyr also uses the Kconfig language to configure the source code. Whether to use devicetree or Kconfig for a particular purpose can sometimes be confusing. This section should help you decide which one to use.

In short:

- Use devicetree to describe **hardware** and its **boot-time configuration**. Examples include peripherals on a board, boot-time clock frequencies, interrupt lines, etc.
- Use Kconfig to configure **software support** to build into the final image. Examples include whether to add networking support, which drivers are needed by the application, etc.

In other words, devicetree mainly deals with hardware, and Kconfig with software.

For example, consider a board containing a SoC with 2 UART, or serial port, instances.

- The fact that the board has this UART **hardware** is described with two UART nodes in the devicetree. These provide the UART type (via the `compatible` property) and certain settings such as the address range of the hardware peripheral registers in memory (via the `reg` property).
- Additionally, the UART **boot-time configuration** is also described with devicetree. This could include configuration such as the RX IRQ line's priority and the UART baud rate. These may be modifiable at runtime, but their boot-time configuration is described in devicetree.
- From the application code, both UART are interchangeable and using one or the other has no impact, it is only a matter of board design. It is pure **hardware configuration** and should be done with devicetree.
- Whether or not to include **software support** for UART in the build is controlled via Kconfig. Applications which do not need to use the UARTs can remove the driver source code from the build using Kconfig, even though the board's devicetree still includes UART nodes.

As another example, consider a device with a 2.4GHz, multi-protocol radio supporting both the Bluetooth Low Energy and 802.15.4 wireless technologies.

- Devicetree should be used to describe the presence of the radio **hardware**, what driver or drivers it's compatible with, etc.
- **Boot-time configuration** for the radio, such as TX power in dBm, should also be specified using devicetree.
- On the application, using one or the other protocol will not get the same code into play, but the same hardware will be used. This is relevant to **software configuration**.
- Kconfig should determine which **software features** should be built for the radio, such as selecting a BLE or 802.15.4 protocol stack.

As another example, Kconfig options that formerly enabled a particular instance of a driver (that is itself enabled by Kconfig) have been removed. The devices are selected individually using devicetree's *status* keyword on the corresponding hardware instance.

There are **exceptions** to these rules:

- Because Kconfig is unable to flexibly control some instance-specific driver configuration parameters, such as the size of an internal buffer, these options may be defined in devicetree. However, to make clear that they are specific to Zephyr drivers and not hardware description or configuration these properties should be prefixed with `zephyr,`, e.g. `zephyr,random-mac-address` in the common Ethernet devicetree properties.
- Devicetree's `chosen` keyword, which allows the user to select a specific instance of a hardware device to be used for a particular purpose. An example of this is selecting a particular UART for use as the system's console.

Automatic Kconfig symbols from devicetree During the devicetree processing step, CMake runs scripts/dts/gen_driver_kconfig_dts.py, which scans all *DTS root* directories, including dts/bindings and writes `Kconfig.dts` into the build's `KCONFIG_BINARY_DIR` (for example `<build>/zephyr` or `<build>/<image>/zephyr` in case of *Sysbuild (System build)*). For each `compatible = "vendor,chip"` that appears in a binding, the generated file contains:

```
DT_COMPAT_VENDOR_CHIP := vendor,chip

config DT_HAS_VENDOR_CHIP_ENABLED
        def_bool $(dt_compat_enabled,$(DT_COMPAT_VENDOR_CHIP))
```

The assignment keeps the literal `compatible` string available to the preprocessor functions described in *Custom Kconfig Preprocessor Functions*, so that Kconfig files can call helpers such as `$(dt_compat_on_bus,$(DT_COMPAT_<compatible>),i2c)`. Characters like `-`, `,` and `@` are converted to underscores when the symbol names are created.

The hidden boolean symbol becomes `CONFIG_DT_HAS_<compatible>_ENABLED` after Kconfig runs. Its value tracks whether the current devicetree contains at least one node with that `compatible` whose *status* is `okay`. Symbols for enabling drivers should almost certainly be `default y` and have `depends on CONFIG_DT_HAS_<compatible>_ENABLED`, for example:

```
config SENSOR_VENDOR_CHIP
        bool "Vendor Chip sensor"
        default y
        depends on DT_HAS_VENDOR_CHIP_ENABLED
```

Because these symbols are generated automatically, adding a new binding with a `compatible` property is all that is required to make the corresponding `DT_HAS_<compatible>_ENABLED` and `DT_COMPAT_<compatible>` constructs available to Kconfig.

5.3.2 Devicetree Reference

These pages contain reference material for Zephyr's devicetree APIs and built-in bindings.

For the platform-independent details, see the Devicetree specification.

Devicetree API

This is a reference page for the `<zephyr/devicetree.h>` API. The API is macro based. Use of these macros has no impact on scheduling. They can be used from any calling context and at file scope.

Some of these – the ones beginning with `DT_INST_` – require a special macro named `DT_DRV_COMPAT` to be defined before they can be used; these are discussed individually below. These macros are generally meant for use within *device drivers*, though they can be used outside of drivers with appropriate care.

Contents

- *Generic APIs*
  - *Node identifiers and helpers*
  - *Property access*
  - *`ranges` property*
  - *`reg` property*
  - *`interrupts` property*
  - *For-each macros*

  - *Existence checks*
  - *Inter-node dependencies*
  - *Bus helpers*
- *Instance-based APIs*
- *Hardware specific APIs*
  - *CAN*
  - *Clocks*
  - *DMA*
  - *Display*
  - *Fixed and mapped flash partitions*
  - *GPIO*
  - *HWSpinlock*
  - *IO channels*
  - *MBOX*
  - *MUX*
  - *NVMEM*
  - *Ordinals*
  - *Pinctrl (pin control)*
  - *PWM*
  - *Reset Controller*
  - *SPI*
- *Chosen nodes*
- *Zephyr-specific chosen nodes*

Generic APIs The APIs in this section can be used anywhere and do not require `DT_DRV_COMPAT` to be defined.

Node identifiers and helpers A *node identifier* is a way to refer to a devicetree node at C preprocessor time. While node identifiers are not C values, you can use them to access devicetree data in C rvalue form using, for example, the *Property access* API.

The root node / has node identifier `DT_ROOT`. You can create node identifiers for other devicetree nodes using `DT_PATH`, `DT_NODELABEL`, `DT_ALIAS`, and `DT_INST`.

There are also `DT_PARENT` and `DT_CHILD` macros which can be used to create node identifiers for a given node's parent node or a particular child node, respectively.

The following macros create or operate on node identifiers.

Node identifiers and helpers

Related code samples

- GPIO with custom Devicetree bindingUse custom Devicetree binding to control a GPIO.

Property access The following general-purpose macros can be used to access node properties. There are special-purpose APIs for accessing the *ranges property*, *reg property* and *interrupts property*.

Property values can be read using these macros even if the node is disabled, as long as it has a matching binding.

Property accessors

`ranges` property Use these APIs instead of *Property access* to access the `ranges` property. Because this property's semantics are defined by the devicetree specification, these macros can be used even for nodes without matching bindings. However, they take on special semantics when the node's binding indicates it is a PCIe bus node, as defined in the PCI Bus Binding to: IEEE Std 1275-1994 Standard for Boot (Initialization Configuration) Firmware

ranges property

`reg` property Use these APIs instead of *Property access* to access the `reg` property. Because this property's semantics are defined by the devicetree specification, these macros can be used even for nodes without matching bindings.

reg property

`interrupts` property Use these APIs instead of *Property access* to access the `interrupts` property.

Because this property's semantics are defined by the devicetree specification, some of these macros can be used even for nodes without matching bindings. This does not apply to macros which take cell names as arguments.

interrupts property

For-each macros The `DT_FOREACH_ANCESTOR` macro allows iterating over the ancestor node of a devicetree node. Additionally, the `DT_FOREACH_CHILD` macro allows iterating over the children of a devicetree node.

There are special-purpose for-each macros, like `DT_INST_FOREACH_STATUS_OKAY`, but these require `DT_DRV_COMPAT` to be defined before use.

"For-each" macros

Existence checks This section documents miscellaneous macros that can be used to test if a node exists, how many nodes of a certain type exist, whether a node has certain properties, etc. Some macros used for special purposes (such as `DT_IRQ_HAS_IDX` and all macros which require `DT_DRV_COMPAT`) are documented elsewhere on this page.

Existence checks

Related code samples

- GPIO with custom Devicetree bindingUse custom Devicetree binding to control a GPIO.

Inter-node dependencies The `devicetree.h` API has some support for tracking dependencies between nodes. Dependency tracking relies on a binary "depends on" relation between devicetree nodes, which is defined as the transitive closure of the following "directly depends on" relation:

- every non-root node directly depends on its parent node

- a node directly depends on any nodes its properties refer to by phandle, this can be changed with *dependency-mode* in the node's binding
- a node directly depends on its `interrupt-parent` if it has an `interrupts` property
- a parent node inherits all dependencies from its child nodes

A *dependency ordering* of a devicetree is a list of its nodes, where each node `n` appears earlier in the list than any nodes that depend on `n`. A node's *dependency ordinal* is then its zero-based index in that list. Thus, for two distinct devicetree nodes `n1` and `n2` with dependency ordinals `d1` and `d2`, we have:

- `d1 != d2`
- if `n1` depends on `n2`, then `d1 > d2`
- `d1 > d2` does **not** necessarily imply that `n1` depends on `n2`

The Zephyr build system chooses a dependency ordering of the final devicetree and assigns a dependency ordinal to each node. Dependency related information can be accessed using the following macros. The exact dependency ordering chosen is an implementation detail, but cyclic dependencies are detected and cause errors, so it's safe to assume there are none when using these macros.

There are instance number-based conveniences as well; see `DT_INST_DEP_ORD` and subsequent documentation.

Dependency tracking

Bus helpers Zephyr's devicetree bindings language supports a `bus:` key which allows bindings to declare that nodes with a given compatible describe system buses. In this case, child nodes are considered to be on a bus of the given type, and the following APIs may be used.

Bus helpers

Instance-based APIs These are recommended for use within device drivers. To use them, define `DT_DRV_COMPAT` to the lowercase-and-underscores compatible the device driver implements support for. Here is an example devicetree fragment:

```
serial@40001000 {
        compatible = "vnd,serial";
        status = "okay";
        current-speed = <115200>;
};
```

Example usage, assuming `serial@40001000` is the only enabled node with compatible `vnd,serial`:

```
#define DT_DRV_COMPAT vnd_serial
DT_DRV_INST(0)                  // node identifier for serial@40001000
DT_INST_PROP(0, current_speed)  // 115200
```

Warning

Be careful making assumptions about instance numbers. See `DT_INST` for the API guarantees.

As shown above, the `DT_INST_*` APIs are conveniences for addressing nodes by instance number. They are almost all defined in terms of one of the *Generic APIs*. The equivalent generic API can be found by removing `INST_` from the macro name. For example, `DT_INST_PROP(inst, prop)` is equivalent to `DT_PROP(DT_DRV_INST(inst), prop)`. Similarly, `DT_INST_REG_ADDR(inst)` is equivalent to `DT_REG_ADDR(DT_DRV_INST(inst))`, and so on. There

are some exceptions: `DT_ANY_INST_ON_BUS_STATUS_OKAY` and `DT_INST_FOREACH_STATUS_OKAY` are special-purpose helpers without straightforward generic equivalents.

Since `DT_DRV_INST()` requires `DT_DRV_COMPAT` to be defined, it's an error to use any of these without that macro defined.

Note that there are also helpers available for specific hardware; these are documented in *Hardware specific APIs*.

Instance-based devicetree APIs

Hardware specific APIs The following APIs can also be used by including `<devicetree.h>`; no additional include is needed.

CAN These conveniences may be used for nodes which describe CAN controllers/transceivers, and properties related to them.

Devicetree CAN API

Clocks These conveniences may be used for nodes which describe clock sources, and properties related to them.

Devicetree Clocks API

DMA These conveniences may be used for nodes which describe direct memory access controllers or channels, and properties related to them.

Devicetree DMA API

Display These conveniences may be used for nodes which describe display controllers, and properties related to them.

Devicetree Display API

Fixed and mapped flash partitions These conveniences may be used for the special-purpose `fixed-partitions` and `zephyr,mapped-partition` compatibles used to encode information about flash memory partitions in the device tree. See `fixed-partitions` and `zephyr,mapped-partition` for more details.

Devicetree Fixed Partition API

GPIO These conveniences may be used for nodes which describe GPIO controllers/pins, and properties related to them.

Devicetree GPIO API

HWSpinlock These conveniences may be used for nodes which describe hardware spinlock, and properties related to them.

Devicetree HW spinlock API

IO channels These are commonly used by device drivers which need to use IO channels (e.g. ADC or DAC channels) for conversion.

Devicetree IO Channels API

MBOX These conveniences may be used for nodes which describe MBOX controllers/users, and properties related to them.

Devicetree MBOX API

MUX These conveniences may be used for nodes which describe MUX controllers/consumers, and properties related to them.

Devicetree MUX API

NVMEM These conveniences may be used for nodes which describe Non-Volatile Memory, and properties related to them.

NVMEM Devicetree Helpers

Ordinals These conveniences may be used for nodes which describe Dependency tracking, and properties related to them.

Dependency tracking

Pinctrl (pin control) These are used to access pin control properties by name or index.

Devicetree nodes may have properties which specify pin control (sometimes known as pin mux) settings. These are expressed using `pinctrl-<index>` properties within the node, where the `<index>` values are contiguous integers starting from 0. These may also be named using the `pinctrl-names` property.

Here is an example:

```
node {
    ...
    pinctrl-0 = <&foo &bar ...>;
    pinctrl-1 = <&baz ...>;
    pinctrl-names = "default", "sleep";
};
```

Above, `pinctrl-0` has name `"default"`, and `pinctrl-1` has name `"sleep"`. The `pinctrl-<index>` property values contain phandles. The `&foo`, `&bar`, etc. phandles within the properties point to nodes whose contents vary by platform, and which describe a pin configuration for the node.

Pin control

PWM These conveniences may be used for nodes which describe PWM controllers and properties related to them.

Devicetree PWMs API

Reset Controller These conveniences may be used for nodes which describe reset controllers and properties related to them.

Devicetree Reset Controller API

SPI These conveniences may be used for nodes which describe either SPI controllers or devices, depending on the case.

Devicetree SPI API

Chosen nodes The special `/chosen` node contains properties whose values describe system-wide settings. The `DT_CHOSEN()` macro can be used to get a node identifier for a chosen node.

Chosen nodes

Zephyr-specific chosen nodes The following table documents some commonly used Zephyr-specific chosen nodes.

Sometimes, a chosen node's label property will be used to set the default value of a Kconfig option which in turn configures a hardware-specific device. This is usually for backwards compatibility in cases when the Kconfig option predates devicetree support in Zephyr. In other cases, there is no Kconfig option, and the devicetree node is used directly in the source code to select a device.

Table 1: Zephyr-specific chosen properties

Property	Purpose
mcuboot,ram-load-dev	When a Zephyr application is built to be loaded to RAM by MCUboot, with `CONFIG_MCUBOOT_BOOTLOADER_MODE_SINGLE_APP_RAM_LOAD`, this property is used to tell MCUboot the load address of the image, which will be the `reg` of the chosen node.
zephyr,boot-mode	Used for *Boot mode* selection, part of *Retention System*, which specifies what image on a device should be booted.
zephyr,bootloader-info	Selects the *Retention System* area used to share information with the bootloader.
zephyr,bt-c2h-uart	Selects the UART used for host communication in the bluetooth_hci_uart
zephyr,bt-hci	Selects the HCI device used by the Bluetooth host stack
zephyr,bt-hci-ipc	Selects the IPC device used to expose a Bluetooth controller to another device or CPU in the bluetooth_hci_ipc sample.
zephyr,bt-mon-uart	Sets UART device used for the Bluetooth monitor logging
zephyr,camera	Video input device, typically a camera.
zephyr,canbus	Sets the default CAN controller
zephyr,code-partition	Flash partition that the Zephyr image's text section should be linked into
zephyr,console	Sets UART device used by console driver
zephyr,cpu-load-counter	Selects the *Counter* device used to track CPU idle time.
zephyr,crc	Selects the CRC device used as an accelerator by the CRC subsystem
zephyr,display	Sets the default display controller
zephyr,dtcm	Data Tightly Coupled Memory node on some Arm SoCs
zephyr,edac	Selects the *Error Detection And Correction (EDAC)* device used by the EDAC subsystem.
zephyr,entropy	A device which can be used as a system-wide entropy source
zephyr,flash	A node whose `reg` is sometimes used to set the defaults for `CONFIG_FLASH_BASE_ADDRESS` and `CONFIG_FLASH_SIZE`
zephyr,flash-controller	The node corresponding to the flash controller device for the `zephyr,flash` node
zephyr,gdbstub-uart	Sets UART device used by the *GDB stub* subsystem
zephyr,hdlc-rcp-if	Selects the radio device used by the OpenThread HDLC RCP interface. This device must have `hdlc_api` as its `api`.
zephyr,host-cmd-espi-backend	Refer to *EC Host Command* API documentation.
zephyr,host-cmd-shi-backend	Refer to *EC Host Command* API documentation.
zephyr,host-cmd-spi-backend	Refer to *EC Host Command* API documentation.

continues on next page

Table 1 – continued from previous page

Property	Purpose
zephyr,host-cmd-uart-backend	Refer to *EC Host Command* API documentation.
zephyr,ieee802154	Used by the networking subsystem to set the IEEE 802.15.4 device
zephyr,ipc	Used by the OpenAMP subsystem to specify the inter-process communication (IPC) device
zephyr,ipc_rsc_table	Specifies a memory region that will be used for the OpenAMP resource table. Only needed if `CONFIG_OPENAMP_COPY_RSC_TABLE` is enabled.
zephyr,ipc_rx	Alternative to `zephyr,ipc`. Selects the IPC device to use for reception when separate devices are used for TX and RX. `zephyr,ipc_tx` must also be provided alongside this chosen, and selects the IPC device to use for transmission. When this chosen and `zephyr,ipc_tx` are provided, the single-device `zephyr,ipc` compatible **must not** be provided.
zephyr,ipc_shm	A node whose `reg` is used by the OpenAMP subsystem to determine the base address and size of the shared memory (SHM) usable for interprocess-communication (IPC)
zephyr,ipc_tx	See description of `zephyr,ipc_rx`.
zephyr,itcm	Instruction Tightly Coupled Memory node on some Arm SoCs
zephyr,led-strip	A LED-strip node which is used to determine the timings of the WS2812 GPIO driver
zephyr,log-ipc	Selects the IPC device used by the logging subsystem's IPC service backend.
zephyr,log-uart	Sets the UART device(s) used by the logging subsystem's UART backend. If defined, the UART log backend would output to the devices listed in this node.
zephyr,modem-uart	Selects the modem UART used by the at_client sample.
zephyr,ocm	On-chip memory node on Xilinx Zynq-7000 and ZynqMP SoCs
zephyr,openthread-counter	Selects the counter device used by the OpenThread platform for microsecond alarm timers when `CONFIG_OPENTHREAD_ALARM_COUNTER` is enabled.
zephyr,osdp-uart	Sets UART device used by OSDP subsystem
zephyr,ot-uart	Used by the OpenThread to specify UART device for Spinel protocol
zephyr,pcie-controller	The node corresponding to the PCIe Controller
zephyr,ppp-uart	Sets UART device used by PPP
zephyr,ram-console	Selects the `RAM section` in which the console subsystem's RAM backend should place its buffer.
zephyr,rtc	Sets the default RTC device (used for example by the *SNTP library* to set the system time when `CONFIG_NET_CONFIG_CLOCK_SNTP_SET_RTC` is enabled)
zephyr,rtk-serial	Selects the *Universal Asynchronous Receiver-Transmitter (UART)* device used by the Serial GNSS RTK client.
zephyr,secure-storage-its-partition	Fixed partition node. This selects the partition used by the Secure Storage ZMS backend.
zephyr,sensor-clock	Selects the *Counter* device used as sensor time source.
zephyr,settings-partition	Fixed partition node. If defined this selects the partition used by the NVS and FCB settings backends.
zephyr,shell-uart	Sets UART device used by serial shell backend
zephyr,sram	A node whose `reg` sets the base address and size of SRAM memory available to the Zephyr image, used during linking

continues on next page

Table 1 – continued from previous page

Property	Purpose
zephyr,system-timer	Selects the hardware timer instance used as the Zephyr system timer, which is a singleton system-wide function. This chosen is needed when the selected system timer driver corresponds to timer hardware for which multiple instances may exist: it selects one instance as the system timer and allows using the other instances with other APIs. This chosen is ignored when the selected system timer driver corresponds to timer hardware for which only one instance may ever exist, such as the Cortex-M SysTick timer; otherwise, it must corresponds to a node that the selected system timer driver can operate. (Note: the Zephyr system timer *driver* is selected using Kconfig options such as `CONFIG_CORTEX_M_SYSTICK` - not Devicetree!)
zephyr,system-timer-companion	Selects the device used to keep time while the primary system timer is inactive in low-power states. It must implement the *Counter* API.
zephyr,tflm-storage	Selects the `memory region` used by the tflite-ethosu sample.
zephyr,touch	Touchscreen controller device node. When LVGL is used, if `CONFIG_LV_Z_POINTER_FROM_CHOSEN_TOUCH` is enabled, an LVGL pointer input device is created using the touchscreen controller as its input source.
zephyr,tracing-uart	Sets UART device used by tracing subsystem
zephyr,uart-mcumgr	UART used for *Device Management*
zephyr,uart-pipe	Sets UART device used by serial pipe driver
zephyr,videoenc	Video encoder device, typically an H264 or MJPEG video encoder.

Bindings index

This page documents the available devicetree bindings. See *Devicetree bindings* for an introduction to the Zephyr bindings file format.

The bindings are grouped both by **type** and by **vendor**. Within each group, bindings are listed by the "compatible" property they apply to, like this:

- <compatible-A>
- <compatible-B> (on <bus-name> bus)
- <compatible-C>
- ...

The text "(on <bus-name> bus)" appears when bindings may behave differently depending on the bus the node appears on. For example, this applies to some sensor device nodes, which may appear as children of either I2C or SPI bus nodes.

Type index This section contains an index of bindings by type. Click on a type's name to go to the list of bindings for that type.

- *1-Wire*
- *ADC*
- *ALH*
- *ARC architecture*
- *ARM architecture*
- *Audio*
- *Auxiliary Display*
- *Base*

- *Biometrics*
- *Bluetooth*
- *Buzzer*
- *Cache*
- *CAN*
- *Charger*
- *Clock control*
- *Clock monitor*
- *Comparator*
- *Core dump*
- *Counter*
- *CPU*
- *CPU frequency scaling*
- *CRC*
- *Cryptographic accelerator*
- *DAC*
- *DAI*
- *DALI*
- *Debug*
- *DFPMCCH*
- *DFPMCCU*
- *Disk*
- *Display*
- *DMA*
- *DSA*
- *DSP*
- *EDAC*
- *ESPI*
- *Ethernet*
- *File system*
- *Firmware*
- *Flash controller*
- *FPGA*
- *Fuel gauge*
- *GNSS*
- *GPIO & Headers*
- *Haptics*
- *Hardware information*
- *Hardware spinlock*

- *HDA*
- *I2C*
- *I2S*
- *I3C*
- *IEEE 802.15.4*
- *IEEE 802.15.4 HDLC RCP interface*
- *IIO*
- *Input*
- *Interrupt controller*
- *IPC*
- *IPM*
- *LED*
- *LED strip*
- *LIN*
- *LoRa*
- *Mailbox*
- *Mailbox Handling Unit*
- *MCUmgr*
- *Memory controller*
- *Memory management*
- *Memory window*
- *MIPI-DBI*
- *MIPI-DSI*
- *Miscellaneous*
- *MMU / MPU*
- *Modem*
- *MTD*
- *Multi-bit SPI*
- *Multi-Function Device*
- *MUX*
- *Networking*
- *NVMEM*
- *OCTOSPI*
- *OPAMP*
- *Options*
- *OTP memory*
- *PCIe*
- *PECI*
- *Performance state*

- *PHY*
- *Pin control*
- *PMCI*
- *Power domain*
- *Power management*
- *Power management CPU operations*
- *PPC architecture*
- *PS/2*
- *PSI5*
- *Pulse IO*
- *PWM*
- *QSPI*
- *Regulator*
- *Renesas RX*
- *Reserved memory*
- *Reset controller*
- *Retained memory*
- *Retention*
- *RISC-V architecture*
- *RNG*
- *RTC*
- *SD*
- *SDHC*
- *Sensors*
- *SENT*
- *Serial controller*
- *Service in Platform*
- *SHI*
- *SMbus*
- *Sound*
- *SPI*
- *SRAM*
- *Stepper*
- *System controller*
- *Tachometer*
- *Test*
- *Timer*
- *Timestamp*
- *USB*

- *USB Audio Offload Link*
- *USB Type-C*
- *USB Type-C Port Controller*
- *Video*
- *Virtio*
- *Virtualization*
- *Wakeup Controller*
- *Watchdog*
- *Wi-Fi*
- *Xen*
- *xSPI*

Vendor index This section contains an index of hardware vendors. Click on a vendor's name to go to the list of bindings for that vendor.

- *Generic or vendor-independent*
- *AD Semiconductor Co., Ltd. (adsemi)*
- *Adafruit Industries, LLC (adafruit)*
- *ADH Technology Co. Ltd. (adh-tech)*
- *Advanced Micro Devices (AMD), Inc. (amd)*
- *Aesc Silicon (aesc)*
- *Alibaba Xuantie (xuantie)*
- *Alif Semiconductor (alif)*
- *Allegro DVT (allegro)*
- *Allwinner Technology Co., Ltd. (allwinner)*
- *Altera Corp. (altr)*
- *Ambiq Micro, Inc. (ambiq)*
- *AMS AG (ams)*
- *Analog Devices, Inc. (adi)*
- *Andes Technology Corporation (andestech)*
- *Angst+Pfister (ap)*
- *Apa Electronic Co., Ltd (apa)*
- *Aptina Imaging (aptina)*
- *Arducam Technology Co., Limited (arducam)*
- *Arduino (arduino)*
- *ARM Ltd. (arm)*
- *Asahi Kasei Corp. (asahi-kasei)*
- *ASMedia Technology Inc. (asmedia)*
- *ASPEED Technology Inc. (aspeed)*
- *Atmel Corporation (atmel)*
- *Avago Technologies (avago)*

- *avia semiconductor (avia)*
- *Beijing Phosense Electronic Technology Co., Ltd. (phosense)*
- *Bosch Sensortec GmbH (bosch)*
- *Bouffalo Lab (Nanjing) Co., Ltd. (bflb)*
- *Broadcom Corporation (brcm)*
- *Cadence Design Systems Inc. (cdns)*
- *Centre for Development of Advanced Computing (C-DAC) (cdac)*
- *CFSensor Co., Ltd (cfsensor)*
- *Chip Wealth Technology Ltd. (chipwealth)*
- *ChipOne (chipone)*
- *Chipsemi Corp. (chipsemi)*
- *Cirque Corporation (cirque)*
- *Cirrus Logic, Inc. (cirrus)*
- *Cirrus Logic, Inc. (formerly Wolfson Microelectronics plc) (wolfson)*
- *Cypress Semiconductor Corporation (cypress)*
- *DAVICOM Semiconductor, Inc. (davicom)*
- *DFRobot (dfrobot)*
- *Dialog Semiconductor (dlg)*
- *Digilent, Inc. (digilent)*
- *Diodes Incorporated (diodes)*
- *E Ink Corporation (eink)*
- *Efinix Inc (efinix)*
- *Egis Technology Inc (egis)*
- *Elan Microelectronic Corp. (elan)*
- *ENE Technology, Inc. (ene)*
- *EPCOS AG (epcos)*
- *Espressif Systems (espressif)*
- *Everlight Electronics Co. Ltd. (everlight)*
- *Fairchild Semiconductor (fcs)*
- *Feature Integration Technology Inc. (fintek)*
- *Festo SE & Co. KG (festo)*
- *Fibocom Wireless Inc. (fibocom)*
- *Fitipower Integrated Technology Inc. (fitipower)*
- *FoBE Studio (fobe)*
- *FocalTech Systems Co.,Ltd (focaltech)*
- *Freescale Semiconductor (fsl)*
- *Fujitsu Ltd. (fujitsu)*
- *Futaba Corporation (futaba)*
- *Future Technology Devices International Ltd. (ftdi)*

- *Fuzhou Rockchip Electronics Co., Ltd (rockchip)*
- *Gaisler (gaisler)*
- *Galaxycore, Inc. (galaxycore)*
- *Gas Sensing Solutions Ltd. (gss)*
- *GigaDevice Semiconductor (gd)*
- *GlobalTop Technology, Inc. (globaltop)*
- *GreeLed Electronic Ltd. (greeled)*
- *Guangzhou Aosong Electronic Co., Ltd. (aosong)*
- *Hamamatsu Photonics K.K. (hamamatsu)*
- *Hangzhou Grow Technology Co., Ltd. (hzgrow)*
- *Hangzhou Zhian Technology Co., Ltd. (zhiantec)*
- *Hazard3 by Luke Wren (hazard3)*
- *Himax Technologies, Inc. (himax)*
- *Hitachi Ltd. (hit)*
- *Holtek Semiconductor, Inc. (holtek)*
- *Honeywell (honeywell)*
- *HOPERF Microelectronics Co. Ltd (hoperf)*
- *Hycon Technology Corp. (hycon)*
- *Hynitron (hynitron)*
- *Iclegend Microelectronics Co., Ltd. (iclegend)*
- *ILI Technology Corporation (ILITEK) (ilitek)*
- *Imagination Technologies Ltd. (formerly MIPS Technologies Inc.) (mti)*
- *Infineon Technologies (infineon)*
- *Innovative Sensor Technology IST AG (ist)*
- *Integrated Silicon Solutions Inc. (issi)*
- *Integrated Solutions Technology Inc. (istech)*
- *Intel Corporation (intel)*
- *Intersil (isil)*
- *InvenSense Inc. (invensense)*
- *Inventek Systems (inventek)*
- *Isentek Inc. (isentek)*
- *ITE Tech. Inc. (ite)*
- *Japan Display Inc. (jdi)*
- *JEDEC Solid State Technology Association (jedec)*
- *Koninklijke Philips N.V. (philips)*
- *Kvaser (kvaser)*
- *Lattice Semiconductor (lattice)*
- *LeveTop Semiconductor (levetop)*
- *Linaro Limited (linaro)*

- *Linear Technology Corporation (lltc)*
- *LiteOn OptoElectronics (liteon)*
- *LiteX SoC builder (litex)*
- *lowRISC Community Interest Company (lowrisc)*
- *LuatOS Team (luatos)*
- *M5Stack (m5stack)*
- *Macronix International Co., Ltd. (mxicy)*
- *MaxBotix Inc. (maxbotix)*
- *Maxim Integrated Products (maxim)*
- *Maxlinear, Inc. (maxlinear)*
- *Measurement Specialties (meas)*
- *MediaTek Inc. (mediatek)*
- *Melexis N.V. (melexis)*
- *MEMSIC Inc. (memsic)*
- *Micro Crystal AG (microcrystal)*
- *Micro:bit Educational Foundation (microbit)*
- *Microchip Technology Inc. (microchip)*
- *Micron Technology Inc. (micron)*
- *Monolithic Power Systems Inc. (mps)*
- *Motorcomm Electronic Technology Co., Ltd. (motorcomm)*
- *Motorola, Inc. (motorola)*
- *Murata Manufacturing Co., Ltd. (murata)*
- *National Semiconductor (national)*
- *NEORV32 RISC-V Processor (neorv32)*
- *Netsol, CO., LTD (netsol)*
- *Ningbo Peacefair Electronic Technology Co., Ltd (peacefair)*
- *Nintendo (nintendo)*
- *Nordic Semiconductor (nordic)*
- *Noritake Co., Inc. Electronics Division (noritake)*
- *Nuclei System Technology (nuclei)*
- *Nuvoton Technology Corporation (nuvoton)*
- *NXP Semiconductors (nxp)*
- *OmniVision Technologies Inc. (ovti)*
- *OMRON Corporation (omron)*
- *ON Semiconductor Corp. (onnn)*
- *open-isa.org (openisa)*
- *OpenCores.org (opencores)*
- *OpenHW Group (openhwgroup)*
- *OpenThread.io (openthread)*

- *Orise Technology (orisetech)*
- *Panasonic Corporation (panasonic)*
- *Parade Technologies Inc. (parade)*
- *PixArt Imaging Inc. (pixart)*
- *Plantower Co., Ltd (plantower)*
- *PNI Sensor Corporation (pni)*
- *Princeton Technology Corp. (ptc)*
- *QEMU, a generic and open source machine emulator and virtualizer (qemu)*
- *Qorvo, Inc (formerly Decawave) (decawave)*
- *QST Corporation (qst)*
- *Qualcomm Atheros, Inc. (qca)*
- *Qualcomm Technologies, Inc (qcom)*
- *Quectel Wireless Solutions Co., Ltd. (quectel)*
- *QuickLogic Corp. (quicklogic)*
- *Raspberry Pi Foundation (raspberrypi)*
- *Raydium Semiconductor Corp. (raydium)*
- *Realtek Semiconductor Corp. (realtek)*
- *Renesas Electronics Corporation (renesas)*
- *Reyax Technology Co., Ltd. (reyax)*
- *Richtek Technology Corporation (richtek)*
- *RISC-V Foundation (riscv)*
- *ROCKTECH DISPLAYS LIMITED (rocktech)*
- *ROHM Semiconductor Co., Ltd (rohm)*
- *Sciosense B.V. (sciosense)*
- *Seeed Technology Co., Ltd (seeed)*
- *SEGGER Microcontroller GmbH (segger)*
- *Seiko Epson Corp. (epson)*
- *Semtech Corporation (semtech)*
- *Sensirion AG (sensirion)*
- *sensry.io (sensry)*
- *Sensylink Inc. (sensylink)*
- *Sequans Communications (sqn)*
- *SHAKTI Processor Program (shakti)*
- *Shanghai Awinic Technology Co., Ltd. (awinic)*
- *Sharp Corporation (sharp)*
- *Shenzhen Frida LCD Co., Ltd. (frida)*
- *Shenzhen Huiding Technology Co., Ltd. (goodix)*
- *Shenzhen Jinghua Displays Electronics Co., Ltd. (jhd)*
- *Shenzhen Titan Micro Electronics Co., Ltd. (titanmec)*

- *Shenzhen Xptek Technology Co., Ltd (xptek)*
- *Siemens AG (siemens)*
- *Sierra Wireless (swir)*
- *SiFive, Inc. (sifive)*
- *SiFli Technologies (sifli)*
- *Silergy Corp. (silergy)*
- *Silicon Laboratories (silabs)*
- *SIMCom Wireless Solutions Co., LTD (simcom)*
- *Sino Wealth Electronic Ltd (sinowealth)*
- *Sitronix Technology Corporation (sitronix)*
- *Skyworks Solutions, Inc. (skyworks)*
- *Smart Battery System (sbs)*
- *Socionext Inc. (socionext)*
- *Soldered Electronics (solderedelectronics)*
- *Solomon Systech Limited (solomon)*
- *Sony Corporation (sony)*
- *SparkFun Electronics (sparkfun)*
- *SpinalHDL (spinalhdl)*
- *Standard Microsystems Corporation (smsc)*
- *StarChips Technology Inc. (sct)*
- *STMicroelectronics (st)*
- *Synaptics Inc. (syna)*
- *Synopsys, Inc. (snps)*
- *Synopsys, Inc. (formerly ARC International PLC) (arc)*
- *TBS Technologies (tbs)*
- *TDK Corporation. (tdk)*
- *Telink Semiconductor (telink)*
- *Telit Cinterion (telit)*
- *Texas Instruments (ti)*
- *Trasna (trasna)*
- *u-blox (u-blox)*
- *UltraChip Inc. (ultrachip)*
- *Virtual I/O Device Specification, developed by the OASIS consortium (virtio)*
- *Vishay Intertechnology, Inc (vishay)*
- *Waveshare Electronics (waveshare)*
- *WeAct Studio (weact)*
- *Winbond Electronics corp. (winbond)*
- *WinChipHead (wch)*
- *Wistron NeWeb Corporation (wnc)*

- *WIZnet Co., Ltd. (wiznet)*
- *Worldsemi Co., Limited (worldsemi)*
- *Würth Elektronik GmbH. (we)*
- *X-Powers (x-powers)*
- *Xen Hypervisor (xen)*
- *Xilinx (xlnx)*
- *Zephyr Project (zephyr)*
- *Zhengzhou Winsen Electronics Technology Co., Ltd. (winsen)*
- *Unknown vendor*

Bindings by type

1-Wire

- dtbinding_adi_max32_w1
- dtbinding_maxim_ds2482_800
- dtbinding_maxim_ds2482_800_channel
- dtbinding_maxim_ds2484
- dtbinding_maxim_ds2485
- dtbinding_zephyr_w1_gpio
- dtbinding_zephyr_w1_serial

ADC (Analog to Digital Converter)

- dtbinding_adi_ad4050_adc
- dtbinding_adi_ad4052_adc
- dtbinding_adi_ad4114_adc
- dtbinding_adi_ad4130_adc
- dtbinding_adi_ad4170_adc
- dtbinding_adi_ad4190_8_adc
- dtbinding_adi_ad4190_adc
- dtbinding_adi_ad4195_8_adc
- dtbinding_adi_ad4195_adc
- dtbinding_adi_ad559x_adc
- dtbinding_adi_ad7124_adc
- dtbinding_adi_ade7913
- dtbinding_adi_max32_adc
- dtbinding_adi_max32_adc_10b
- dtbinding_adi_max32_adc_b_me18
- dtbinding_adi_max32_adc_sar
- dtbinding_ambiq_adc
- dtbinding_arduino_uno_adc

- dtbinding_atmel_sam_adc
- dtbinding_atmel_sam_afec
- dtbinding_atmel_sam0_adc
- dtbinding_bflb_adc
- dtbinding_ene_kb106x_adc
- dtbinding_ene_kb1200_adc
- dtbinding_espressif_esp32_adc
- dtbinding_gd_gd32_adc
- dtbinding_infineon_adc
- dtbinding_infineon_autanalog_sar_adc
- dtbinding_infineon_autanalog_sar_fifo
- dtbinding_infineon_autanalog_sar_fir
- dtbinding_infineon_hppass_sar_adc
- dtbinding_infineon_sar_adc
- dtbinding_infineon_xmc4xxx_adc
- dtbinding_ite_it51xxx_adc
- dtbinding_ite_it8xxx2_adc
- dtbinding_lltc_ltc2451
- dtbinding_m5stack_m5pm1_adc
- dtbinding_maxim_max11102
- dtbinding_maxim_max11103
- dtbinding_maxim_max11105
- dtbinding_maxim_max11106
- dtbinding_maxim_max11110
- dtbinding_maxim_max11111
- dtbinding_maxim_max11115
- dtbinding_maxim_max11116
- dtbinding_maxim_max11117
- dtbinding_maxim_max11253
- dtbinding_maxim_max11254
- dtbinding_maxim_max2253x
- dtbinding_microchip_adc_g1
- dtbinding_microchip_mcp3204
- dtbinding_microchip_mcp3208
- dtbinding_microchip_mcp3221
- dtbinding_microchip_mcp356xr
- dtbinding_microchip_xec_adc
- dtbinding_nordic_nrf_adc
- dtbinding_nordic_nrf_saadc

- dtbinding_nuvoton_npcx_adc
- dtbinding_nuvoton_numaker_adc
- dtbinding_nxp_adc12
- dtbinding_nxp_aon_lpadc
- dtbinding_nxp_gau_adc
- dtbinding_nxp_kinetis_adc16
- dtbinding_nxp_lpc_lpadc
- dtbinding_nxp_mcux_12b1msps_sar
- dtbinding_nxp_s32_adc_sar
- dtbinding_nxp_sar_adc
- dtbinding_nxp_vf610_adc
- dtbinding_raspberrypi_pico_adc
- dtbinding_realtek_ameba_adc
- dtbinding_realtek_bee_adc
- dtbinding_realtek_rts5912_adc
- dtbinding_renesas_ra_adc_b
- dtbinding_renesas_ra_adc_b_controller
- dtbinding_renesas_ra_adc_b_scan_group
- dtbinding_renesas_ra_adc12
- dtbinding_renesas_ra_adc16
- dtbinding_renesas_rx_adc
- dtbinding_renesas_rz_adc
- dtbinding_renesas_rz_adc_c
- dtbinding_renesas_rz_adc_e
- dtbinding_renesas_rza2m_adc
- dtbinding_renesas_smartbond_adc
- dtbinding_renesas_smartbond_sdadc
- dtbinding_sifli_sf32lb_gpadc
- dtbinding_silabs_gecko_adc
- dtbinding_silabs_iadc
- dtbinding_silabs_series3_adc
- dtbinding_silabs_siwx91x_adc
- dtbinding_st_stm32_adc
- dtbinding_st_stm32f1_adc
- dtbinding_st_stm32f4_adc
- dtbinding_st_stm32n6_adc
- dtbinding_st_stm32wb0_adc
- dtbinding_telink_b91_adc
- dtbinding_ti_adc081c021

- dtbinding_ti_adc081c027
- dtbinding_ti_adc101c021
- dtbinding_ti_adc101c027
- dtbinding_ti_adc121c021
- dtbinding_ti_adc121c027
- dtbinding_ti_adc124s021
- dtbinding_ti_ads1013
- dtbinding_ti_ads1014
- dtbinding_ti_ads1015
- dtbinding_ti_ads1112
- dtbinding_ti_ads1113
- dtbinding_ti_ads1114
- dtbinding_ti_ads1115
- dtbinding_ti_ads1118
- dtbinding_ti_ads1119
- dtbinding_ti_ads114s06
- dtbinding_ti_ads114s08
- dtbinding_ti_ads1220
- dtbinding_ti_ads124s06
- dtbinding_ti_ads124s08
- dtbinding_ti_ads1262
- dtbinding_ti_ads1263
- dtbinding_ti_ads131m02
- dtbinding_ti_ads7052
- dtbinding_ti_ads7828
- dtbinding_ti_ads7830
- dtbinding_ti_ads7950
- dtbinding_ti_ads7951
- dtbinding_ti_ads7952
- dtbinding_ti_ads7953
- dtbinding_ti_ads7954
- dtbinding_ti_ads7955
- dtbinding_ti_ads7956
- dtbinding_ti_ads7957
- dtbinding_ti_ads7958
- dtbinding_ti_ads7959
- dtbinding_ti_ads7960
- dtbinding_ti_ads7961
- dtbinding_ti_am335x_adc

- dtbinding_ti_cc13xx_cc26xx_adc
- dtbinding_ti_cc23x0_adc
- dtbinding_ti_cc32xx_adc
- dtbinding_ti_lmp90077
- dtbinding_ti_lmp90078
- dtbinding_ti_lmp90079
- dtbinding_ti_lmp90080
- dtbinding_ti_lmp90097
- dtbinding_ti_lmp90098
- dtbinding_ti_lmp90099
- dtbinding_ti_lmp90100
- dtbinding_ti_mspm0_adc12
- dtbinding_ti_tla2021
- dtbinding_ti_tla2022
- dtbinding_ti_tla2024
- dtbinding_ti_tla2528_adc
- dtbinding_wch_adc
- dtbinding_wisblock_adc
- dtbinding_zephyr_adc_emul

ALH (Audio Link Hub)

- dtbinding_intel_alh_dai

ARC architecture

- dtbinding_arc_dccm
- dtbinding_arc_iccm
- dtbinding_arc_xccm
- dtbinding_arc_yccm

ARM architecture

- dtbinding_arm_axi_timing_adapter
- dtbinding_arm_beetle_syscon
- dtbinding_arm_dtcm
- dtbinding_arm_ethos_u
- dtbinding_arm_itcm
- dtbinding_arm_scc
- dtbinding_atmel_sam_ssc
- dtbinding_atmel_sam0_id
- dtbinding_atmel_sam0_sercom
- dtbinding_infineon_edge_npu

- dtbinding_infineon_scb
- dtbinding_linaro_optee_tz
- dtbinding_microchip_sercom_g1
- dtbinding_nordic_nrf_acl
- dtbinding_nordic_nrf_bprot
- dtbinding_nordic_nrf_ctrlapperi
- dtbinding_nordic_nrf_dcnf
- dtbinding_nordic_nrf_egu
- dtbinding_nordic_nrf_kmu
- dtbinding_nordic_nrf_mpu
- dtbinding_nordic_nrf_mutex
- dtbinding_nordic_nrf_mwu
- dtbinding_nordic_nrf_pwr_antswc
- dtbinding_nordic_nrf_reset
- dtbinding_nordic_nrf_resetinfo
- dtbinding_nordic_nrf_spu
- dtbinding_nordic_nrf_swi
- dtbinding_nordic_nrf_tampc
- dtbinding_nordic_nrf_uicr
- dtbinding_nordic_nrf_wicr
- dtbinding_nordic_nrf71_uicr
- dtbinding_nordic_nrf71_wifi_antsw
- dtbinding_nuvoton_numaker_npu
- dtbinding_nxp_imx_dtcm
- dtbinding_nxp_imx_epit
- dtbinding_nxp_imx_itcm
- dtbinding_nxp_lpc_flexcomm
- dtbinding_nxp_mcux_xbar
- dtbinding_nxp_nbu
- dtbinding_nxp_rw_soc_ctrl
- dtbinding_renesas_ra_npu
- dtbinding_st_stm32_ccm
- dtbinding_xlnx_zynq_ocm

Audio

- dtbinding_ambiq_pdm
- dtbinding_awinic_aw88298
- dtbinding_cirrus_cs35l56_i2c
- dtbinding_cirrus_cs35l56_spi

- dtbinding_cirrus_cs35l57_i2c
- dtbinding_cirrus_cs35l57_spi
- dtbinding_cirrus_cs43l22
- dtbinding_dlg_da7212
- dtbinding_infineon_pdm
- dtbinding_intel_adsp_mic_privacy
- dtbinding_maxim_max98091
- dtbinding_nordic_nrf_pdm
- dtbinding_nxp_dmic
- dtbinding_nxp_micfil
- dtbinding_renesas_ra_pdm
- dtbinding_sifli_sf32lb_audcodec
- dtbinding_st_mpxxdtyy_i2s
- dtbinding_st_stm32_dfsdm
- dtbinding_st_stm32_dfsdm_dmic
- dtbinding_st_stm32_mdf
- dtbinding_ti_pcm1681_spi
- dtbinding_ti_pcm1681_i2c
- dtbinding_ti_tas2563
- dtbinding_ti_tas6422dac
- dtbinding_ti_tlv320aic26
- dtbinding_ti_tlv320aic3110
- dtbinding_ti_tlv320dac
- dtbinding_wolfson_wm8904
- dtbinding_wolfson_wm8960
- dtbinding_wolfson_wm8962
- dtbinding_zephyr_dummy_codec
- dtbinding_zephyr_native_sim_dmic
- dtbinding_zephyr_pdm_dmic

Auxiliary Display

- dtbinding_gpio_7_segment
- dtbinding_hit_hd44780
- dtbinding_jhd_jhd1313
- dtbinding_noritake_itron
- dtbinding_nxp_slcd
- dtbinding_ptc_pt6314
- dtbinding_slcd_panel
- dtbinding_sparkfun_serlcd

- dtbinding_titanmec_tm1637

Base

- dtbinding_zephyr_memory_region

Biometrics

- dtbinding_adh_tech_gt5x
- dtbinding_zephyr_biometrics_emul
- dtbinding_zhiantec_zfm_x0

Bluetooth

- dtbinding_ambiq_bt_hci_spi
- dtbinding_bflb_bt_hci
- dtbinding_espressif_esp_hosted_mcu_bt_hci
- dtbinding_espressif_esp32_bt_hci
- dtbinding_infineon_bless_hci
- dtbinding_infineon_bt_hci_uart
- dtbinding_infineon_cyw208xx_hci
- dtbinding_nxp_bt_hci_uart
- dtbinding_nxp_hci_ble
- dtbinding_realtek_ameba_bt_hci
- dtbinding_realtek_bee_bt_hci
- dtbinding_renesas_bt_hci_da1453x
- dtbinding_renesas_bt_hci_da1469x
- dtbinding_sifli_sf32lb_mailbox
- dtbinding_silabs_bt_hci_efr32
- dtbinding_silabs_siwx91x_bt_hci
- dtbinding_st_hci_spi_v1
- dtbinding_st_hci_spi_v2
- dtbinding_st_hci_stm32wb0
- dtbinding_st_hci_stm32wba
- dtbinding_st_stm32wb_rf
- dtbinding_zephyr_bt_hci_3wire_uart
- dtbinding_zephyr_bt_hci_entropy
- dtbinding_zephyr_bt_hci_ipc
- dtbinding_zephyr_bt_hci_ll_sw_split
- dtbinding_zephyr_bt_hci_spi
- dtbinding_zephyr_bt_hci_spi_peripheral
- dtbinding_zephyr_bt_hci_spi_slave
- dtbinding_zephyr_bt_hci_uart

- dtbinding_zephyr_bt_hci_userchan

Buzzer

- dtbinding_gpio_buzzer
- dtbinding_pwm_buzzer

Cache

- dtbinding_andestech_l2c
- dtbinding_bflb_l1c

CAN (Controller Area Network)

- dtbinding_adi_max32_can
- dtbinding_atmel_sam_can
- dtbinding_atmel_sam0_can
- dtbinding_bflb_bl61x_can
- dtbinding_espressif_esp32_twai
- dtbinding_espressif_esp32_twaifd
- dtbinding_infineon_can
- dtbinding_infineon_canfd_controller
- dtbinding_infineon_xmc4xxx_can
- dtbinding_infineon_xmc4xxx_can_node
- dtbinding_kvaser_pcican
- dtbinding_microchip_mcp2515
- dtbinding_microchip_mcp251xfd
- dtbinding_nordic_nrf_can
- dtbinding_nuvoton_numaker_canfd
- dtbinding_nxp_flexcan
- dtbinding_nxp_flexcan_fd
- dtbinding_nxp_lpc_mcan
- dtbinding_nxp_s32_canxl
- dtbinding_realtek_bee_can
- dtbinding_renesas_ra_canfd
- dtbinding_renesas_ra_canfd_global
- dtbinding_renesas_rcar_can
- dtbinding_renesas_rcar_rscanfd
- dtbinding_renesas_rcar_rscanfd_global
- dtbinding_renesas_rz_canfd
- dtbinding_renesas_rz_canfd_global
- dtbinding_st_stm32_bxcan
- dtbinding_st_stm32_fdcan

- dtbinding_st_stm32h7_fdcan
- dtbinding_ti_mspm0_canfd
- dtbinding_ti_tcan4x5x
- dtbinding_zephyr_can_loopback
- dtbinding_zephyr_fake_can
- dtbinding_zephyr_native_linux_can

Charger

- dtbinding_adi_adp5360_charger
- dtbinding_maxim_max20335_charger
- dtbinding_nordic_npm10xx_charger
- dtbinding_nxp_pca9422_charger
- dtbinding_nxp_pf1550_charger
- dtbinding_sbs_sbs_charger
- dtbinding_silergy_sy6974b
- dtbinding_ti_bq24190
- dtbinding_ti_bq24295
- dtbinding_ti_bq24296
- dtbinding_ti_bq24296m
- dtbinding_ti_bq24297
- dtbinding_ti_bq24298
- dtbinding_ti_bq25180
- dtbinding_ti_bq25186
- dtbinding_ti_bq25188
- dtbinding_ti_bq25713
- dtbinding_x_powers_axp2101_charger
- dtbinding_zephyr_charger_gpio

Clock control

- dtbinding_adi_axi_clkgen
- dtbinding_adi_max32_erfo
- dtbinding_adi_max32_extclk
- dtbinding_adi_max32_gcr
- dtbinding_aesc_clock_controller
- dtbinding_alif_clock_mux
- dtbinding_alif_clockctrl
- dtbinding_ambiq_clkctrl
- dtbinding_aspeed_ast10x0_clock
- dtbinding_atmel_sam_pmc
- dtbinding_atmel_sam0_gclk

- dtbinding_atmel_sam0_mclk
- dtbinding_atmel_sam0_osc32kctrl
- dtbinding_bflb_bclk
- dtbinding_bflb_bl60x_clock_controller
- dtbinding_bflb_bl616cl_clock_controller
- dtbinding_bflb_bl61x_clock_controller
- dtbinding_bflb_bl70x_clock_controller
- dtbinding_bflb_bl70x_l_clock_controller
- dtbinding_bflb_bl70x_l_dll
- dtbinding_bflb_bl808_clock_controller
- dtbinding_bflb_clock_controller
- dtbinding_bflb_f32k
- dtbinding_bflb_flash_clk
- dtbinding_bflb_mm_clk
- dtbinding_bflb_pll
- dtbinding_bflb_root_clk
- dtbinding_elan_em32_ahb
- dtbinding_elan_em32_apb
- dtbinding_espressif_esp32_clock
- dtbinding_fixed_clock
- dtbinding_fixed_factor_clock
- dtbinding_focaltech_ft9001_cpm
- dtbinding_gd_gd32_cctl
- dtbinding_infineon_cat1c_clk_hf
- dtbinding_infineon_clk_osc
- dtbinding_infineon_clk_pll
- dtbinding_infineon_fixed_clock
- dtbinding_infineon_fixed_factor_clock
- dtbinding_infineon_peri_clk
- dtbinding_infineon_peri_div
- dtbinding_intel_adsp_shim_clkctl
- dtbinding_intel_agilex_clock
- dtbinding_intel_agilex5_clock
- dtbinding_ite_it51xxx_ecpm
- dtbinding_litex_clk
- dtbinding_litex_clkout
- dtbinding_mediatek_mt8188_infracfg_ao
- dtbinding_mediatek_mt818x_cpuclk
- dtbinding_mediatek_mt8195_cpuclk

- dtbinding_microchip_pic32ck_sg_gc_clock
- dtbinding_microchip_pic32ck_sg_gc_dfll48m
- dtbinding_microchip_pic32ck_sg_gc_gclkgen
- dtbinding_microchip_pic32ck_sg_gc_gclkperiph
- dtbinding_microchip_pic32ck_sg_gc_mclkcpu
- dtbinding_microchip_pic32ck_sg_gc_mclkperiph
- dtbinding_microchip_pic32ck_sg_gc_pll
- dtbinding_microchip_pic32ck_sg_gc_rtc
- dtbinding_microchip_pic32ck_sg_gc_xosc
- dtbinding_microchip_pic32ck_sg_gc_xosc32k
- dtbinding_microchip_pic32cm_jh_clock
- dtbinding_microchip_pic32cm_jh_fdpll
- dtbinding_microchip_pic32cm_jh_gclkgen
- dtbinding_microchip_pic32cm_jh_gclkperiph
- dtbinding_microchip_pic32cm_jh_mclkcpu
- dtbinding_microchip_pic32cm_jh_mclkperiph
- dtbinding_microchip_pic32cm_jh_osc32k
- dtbinding_microchip_pic32cm_jh_osc48m
- dtbinding_microchip_pic32cm_jh_rtc
- dtbinding_microchip_pic32cm_jh_xosc
- dtbinding_microchip_pic32cm_jh_xosc32k
- dtbinding_microchip_pic32cm_pl_clock
- dtbinding_microchip_pic32cm_pl_gclkgen
- dtbinding_microchip_pic32cm_pl_gclkperiph
- dtbinding_microchip_pic32cm_pl_mclkcpu
- dtbinding_microchip_pic32cm_pl_mclkperiph
- dtbinding_microchip_pic32cm_pl_osc32k
- dtbinding_microchip_pic32cm_pl_oschf
- dtbinding_microchip_pic32cm_pl_rtc
- dtbinding_microchip_pic32cm_pl_xosc32k
- dtbinding_microchip_pic32cm_sg_gc_clock
- dtbinding_microchip_pic32cm_sg_gc_dfll48m
- dtbinding_microchip_pic32cm_sg_gc_dpll
- dtbinding_microchip_pic32cm_sg_gc_gclkgen
- dtbinding_microchip_pic32cm_sg_gc_gclkperiph
- dtbinding_microchip_pic32cm_sg_gc_mclkdomain
- dtbinding_microchip_pic32cm_sg_gc_mclkperiph
- dtbinding_microchip_pic32cm_sg_gc_rtc
- dtbinding_microchip_pic32cm_sg_gc_xosc

- dtbinding_microchip_pic32cm_sg_gc_xosc32k
- dtbinding_microchip_pic32cz_ca_clock
- dtbinding_microchip_pic32cz_ca_dfll48m
- dtbinding_microchip_pic32cz_ca_dpll
- dtbinding_microchip_pic32cz_ca_gclkgen
- dtbinding_microchip_pic32cz_ca_gclkperiph
- dtbinding_microchip_pic32cz_ca_mclkdomain
- dtbinding_microchip_pic32cz_ca_mclkperiph
- dtbinding_microchip_pic32cz_ca_rtc
- dtbinding_microchip_pic32cz_ca_xosc
- dtbinding_microchip_pic32cz_ca_xosc32k
- dtbinding_microchip_sam_d5x_e5x_clock
- dtbinding_microchip_sam_d5x_e5x_dfll
- dtbinding_microchip_sam_d5x_e5x_fdpll
- dtbinding_microchip_sam_d5x_e5x_gclkgen
- dtbinding_microchip_sam_d5x_e5x_gclkperiph
- dtbinding_microchip_sam_d5x_e5x_mclkcpu
- dtbinding_microchip_sam_d5x_e5x_mclkperiph
- dtbinding_microchip_sam_d5x_e5x_rtc
- dtbinding_microchip_sam_d5x_e5x_xosc
- dtbinding_microchip_sam_d5x_e5x_xosc32k
- dtbinding_microchip_sam_pmc
- dtbinding_microchip_sama7g5_sckc
- dtbinding_microchip_smartfusion2_clock
- dtbinding_microchip_xec_pcr
- dtbinding_nordic_nrf_auxpll
- dtbinding_nordic_nrf_clock
- dtbinding_nordic_nrf_clock_hfclk
- dtbinding_nordic_nrf_clock_hfclk192m
- dtbinding_nordic_nrf_clock_hfclkaudio
- dtbinding_nordic_nrf_clock_lfclk
- dtbinding_nordic_nrf_clock_xo
- dtbinding_nordic_nrf_clock_xo24m
- dtbinding_nordic_nrf_fll16m
- dtbinding_nordic_nrf_hsfll_global
- dtbinding_nordic_nrf_hsfll_local
- dtbinding_nordic_nrf_iron_hsfll_local
- dtbinding_nordic_nrf_lfrc
- dtbinding_nordic_nrf51_hfxo

- dtbinding_nordic_nrf52_hfxo
- dtbinding_nordic_nrf53_hfxo
- dtbinding_nordic_nrf53_lfxo
- dtbinding_nordic_nrf53_oscillators
- dtbinding_nordic_nrf54h_hfxo
- dtbinding_nordic_nrf54h_lfxo
- dtbinding_nordic_nrf54l_hfxo
- dtbinding_nordic_nrf54l_lfxo
- dtbinding_nordic_nrf71_hfxo
- dtbinding_nordic_nrf71_lfxo
- dtbinding_nordic_nrfs_audiopll
- dtbinding_nordic_nrfs_lfclk
- dtbinding_nuvoton_m4_hclk_clock
- dtbinding_nuvoton_m4_hxt_clock
- dtbinding_nuvoton_m4_lxt_clock
- dtbinding_nuvoton_m4_pll_clock
- dtbinding_nuvoton_npcm_pcc
- dtbinding_nuvoton_npcx_pcc
- dtbinding_nuvoton_numaker_pcc
- dtbinding_nuvoton_numaker_scc
- dtbinding_nuvoton_numicro_m4_pcc
- dtbinding_nuvoton_numicro_m4_scc
- dtbinding_nxp_firc
- dtbinding_nxp_fxosc
- dtbinding_nxp_imx_anatop
- dtbinding_nxp_imx_ccm
- dtbinding_nxp_imx_ccm_fnpll
- dtbinding_nxp_imx_ccm_rev2
- dtbinding_nxp_imx_ccm_rev3
- dtbinding_nxp_imx_ccm_rev3_root
- dtbinding_nxp_imx_cguana
- dtbinding_nxp_imx_cguana_avpll
- dtbinding_nxp_imx_cguana_core_pll
- dtbinding_nxp_imx_cguana_frac_pll
- dtbinding_nxp_imx_cguana_fro12m
- dtbinding_nxp_imx_cguana_fro192m
- dtbinding_nxp_imx_cguana_sxosc
- dtbinding_nxp_imx9_pll
- dtbinding_nxp_imxrt118x_arm_pll

- dtbinding_nxp_imxrt11xx_arm_pll
- dtbinding_nxp_kinetis_ke1xf_sim
- dtbinding_nxp_kinetis_mcg
- dtbinding_nxp_kinetis_pcc
- dtbinding_nxp_kinetis_scg
- dtbinding_nxp_kinetis_sim
- dtbinding_nxp_lpc_syscon
- dtbinding_nxp_lpc11u6x_syscon
- dtbinding_nxp_lpc84x_clock
- dtbinding_nxp_mc_cgm
- dtbinding_nxp_mcxc_osc
- dtbinding_nxp_mcxw7x_clock
- dtbinding_nxp_plldig
- dtbinding_nxp_s32_clock
- dtbinding_nxp_scg_k4
- dtbinding_pwm_clock
- dtbinding_raspberrypi_pico_clock
- dtbinding_raspberrypi_pico_clock_controller
- dtbinding_raspberrypi_pico_pll
- dtbinding_raspberrypi_pico_rosc
- dtbinding_raspberrypi_pico_xosc
- dtbinding_realtek_ameba_rcc
- dtbinding_realtek_bee_cctl
- dtbinding_realtek_rts5817_clock
- dtbinding_realtek_rts5912_sccon
- dtbinding_renesas_r8a7795_cpg_mssr
- dtbinding_renesas_r8a779f0_cpg_mssr
- dtbinding_renesas_r8a779g0_cpg_mssr
- dtbinding_renesas_ra_cgc_busclk
- dtbinding_renesas_ra_cgc_external_clock
- dtbinding_renesas_ra_cgc_pclk
- dtbinding_renesas_ra_cgc_pclk_block
- dtbinding_renesas_ra_cgc_pll
- dtbinding_renesas_ra_cgc_pll_out
- dtbinding_renesas_ra_cgc_subclk
- dtbinding_renesas_ra_cgc_utasel
- dtbinding_renesas_rx_cgc_pclk
- dtbinding_renesas_rx_cgc_pclk_block
- dtbinding_renesas_rx_cgc_pll

- dtbinding_renesas_rx_cgc_pllccr_type
- dtbinding_renesas_rx_cgc_root_clock
- dtbinding_renesas_rz_cgc
- dtbinding_renesas_rz_cgc_pll
- dtbinding_renesas_rz_cgc_subclk
- dtbinding_renesas_rz_cgc_sys_clock
- dtbinding_renesas_rz_cpg
- dtbinding_renesas_rz_cpg_clock
- dtbinding_renesas_rz_cpg_pll
- dtbinding_renesas_rza2m_cpg
- dtbinding_renesas_smartbond_lp_clk
- dtbinding_renesas_smartbond_lp_osc
- dtbinding_renesas_smartbond_sys_clk
- dtbinding_rockchip_rk3588_cru
- dtbinding_rockchip_rk3588_cru_clk
- dtbinding_sifli_sf32lb_dll
- dtbinding_sifli_sf32lb_hxt48
- dtbinding_sifli_sf32lb_rcc_clk
- dtbinding_silabs_hfxo
- dtbinding_silabs_series_clock
- dtbinding_silabs_series_clock_output
- dtbinding_silabs_series0_cmu
- dtbinding_silabs_series0_hfxo
- dtbinding_silabs_series0_lfrco
- dtbinding_silabs_series0_lfxo
- dtbinding_silabs_series2_hfrcodpll
- dtbinding_silabs_series2_hfrcoem23
- dtbinding_silabs_series2_lfrco
- dtbinding_silabs_series2_lfxo
- dtbinding_silabs_si32_ahb
- dtbinding_silabs_si32_apb
- dtbinding_silabs_si32_pll
- dtbinding_silabs_siwx91x_clock_route
- dtbinding_silabs_siwx91x_cmu_aon
- dtbinding_silabs_siwx91x_cmu_hp
- dtbinding_silabs_siwx91x_cmu_ulp
- dtbinding_silabs_siwx91x_physical_clock
- dtbinding_silabs_siwx91x_pll
- dtbinding_silabs_socpll

- dtbinding_st_stm32_clock_mco
- dtbinding_st_stm32_clock_mux
- dtbinding_st_stm32_hse_clock
- dtbinding_st_stm32_hsi48_clock
- dtbinding_st_stm32_lse_clock
- dtbinding_st_stm32_msi_clock
- dtbinding_st_stm32_rcc
- dtbinding_st_stm32c0_hsi_clock
- dtbinding_st_stm32c5_rcc
- dtbinding_st_stm32c5_xsik_clock
- dtbinding_st_stm32f0_pll_clock
- dtbinding_st_stm32f0_rcc
- dtbinding_st_stm32f1_clock_mco
- dtbinding_st_stm32f1_pll_clock
- dtbinding_st_stm32f100_pll_clock
- dtbinding_st_stm32f105_pll_clock
- dtbinding_st_stm32f105_pll2_clock
- dtbinding_st_stm32f4_rcc
- dtbinding_st_stm32fx_pll_clock
- dtbinding_st_stm32g0_hsi_clock
- dtbinding_st_stm32g0_pll_clock
- dtbinding_st_stm32g4_pll_clock
- dtbinding_st_stm32h5_pll_clock
- dtbinding_st_stm32h5_rcc
- dtbinding_st_stm32h7_hsi_clock
- dtbinding_st_stm32h7_pll_clock
- dtbinding_st_stm32h7_rcc
- dtbinding_st_stm32h7rs_pll_clock
- dtbinding_st_stm32h7rs_rcc
- dtbinding_st_stm32l0_hsi_clock
- dtbinding_st_stm32l0_msi_clock
- dtbinding_st_stm32l0_pll_clock
- dtbinding_st_stm32l4_pll_clock
- dtbinding_st_stm32mp1_rcc
- dtbinding_st_stm32mp13_cpu_clock_mux
- dtbinding_st_stm32mp13_pll_clock
- dtbinding_st_stm32mp2_rcc
- dtbinding_st_stm32n6_cpu_clock_mux
- dtbinding_st_stm32n6_hse_clock

- dtbinding_st_stm32n6_ic_clock_mux
- dtbinding_st_stm32n6_msi_clock
- dtbinding_st_stm32n6_pll_clock
- dtbinding_st_stm32n6_rcc
- dtbinding_st_stm32u0_pll_clock
- dtbinding_st_stm32u3_msi_clock
- dtbinding_st_stm32u5_msi_clock
- dtbinding_st_stm32u5_pll_clock
- dtbinding_st_stm32u5_rcc
- dtbinding_st_stm32wb_pll_clock
- dtbinding_st_stm32wb_rcc
- dtbinding_st_stm32wb0_lsi_clock
- dtbinding_st_stm32wb0_rcc
- dtbinding_st_stm32wba_hse_clock
- dtbinding_st_stm32wba_pll_clock
- dtbinding_st_stm32wba_rcc
- dtbinding_st_stm32wl_hse_clock
- dtbinding_st_stm32wl_rcc
- dtbinding_st_stm32wl3_rcc
- dtbinding_syna_sr100_clock
- dtbinding_ti_cc23x0_lf_xosc
- dtbinding_ti_cdce913
- dtbinding_ti_cdce925
- dtbinding_ti_cdce937
- dtbinding_ti_cdce949
- dtbinding_ti_k2g_sci_clk
- dtbinding_ti_mspm0_clk
- dtbinding_ti_mspm0_osc
- dtbinding_ti_mspm0_pll
- dtbinding_wch_ch32h41x_pll_clock
- dtbinding_wch_ch32v00x_hse_clock
- dtbinding_wch_ch32v00x_hsi_clock
- dtbinding_wch_ch32v00x_pll_clock
- dtbinding_wch_ch32v20x_30x_pll_clock
- dtbinding_wch_rcc
- dtbinding_xlnx_clkx5_wiz_1.0

Clock monitor

- dtbinding_nxp_cmu_fc
- dtbinding_nxp_cmu_fm
- dtbinding_nxp_fmeas
- dtbinding_nxp_freqme

Comparator

- dtbinding_espressif_esp32_ana_cmpr
- dtbinding_infineon_autanalog_ptcomp_comp
- dtbinding_infineon_hppass_csg_comp
- dtbinding_infineon_lp_comp
- dtbinding_infineon_lp_comp_channel
- dtbinding_ite_it51xxx_vcmp
- dtbinding_microchip_ac_g1_comparator
- dtbinding_nordic_nrf_comp
- dtbinding_nordic_nrf_lpcomp
- dtbinding_nxp_acomp
- dtbinding_nxp_cmp
- dtbinding_nxp_hscmp
- dtbinding_nxp_kinetis_acmp
- dtbinding_nxp_lpacmp
- dtbinding_nxp_lpcmp
- dtbinding_renesas_ra_acmphs
- dtbinding_renesas_ra_acmphs_global
- dtbinding_renesas_ra_lvd
- dtbinding_renesas_rx_lvd
- dtbinding_silabs_acmp
- dtbinding_st_stm32_comp
- dtbinding_st_stm32g4_comp
- dtbinding_st_stm32h7_comp
- dtbinding_ti_mspm0_comparator
- dtbinding_wch_ch5xx_comparator
- dtbinding_zephyr_fake_comp

Core dump

- dtbinding_zephyr_coredump

Counter

- dtbinding_adi_max32_counter
- dtbinding_adi_max32_rtc_counter
- dtbinding_adi_max32_wut
- dtbinding_ambiq_counter
- dtbinding_andestech_atcpit100
- dtbinding_arm_cmsdk_dtimer
- dtbinding_arm_cmsdk_timer
- dtbinding_arm_crsas_ma2_counter
- dtbinding_arm_crsas_ma2_timer
- dtbinding_atmel_sam_tc
- dtbinding_atmel_sam0_tc32
- dtbinding_bflb_rtc
- dtbinding_bflb_timer
- dtbinding_bflb_timer_channel
- dtbinding_espressif_esp32_counter
- dtbinding_espressif_esp32_rtc_timer
- dtbinding_espressif_esp32_timer
- dtbinding_gd_gd32_timer
- dtbinding_infineon_counter
- dtbinding_infineon_tcpwm_counter
- dtbinding_intel_ace_art_counter
- dtbinding_intel_ace_rtc_counter
- dtbinding_ite_it51xxx_counter
- dtbinding_ite_it8xxx2_counter
- dtbinding_microchip_rtc_g1_counter
- dtbinding_microchip_sam_pit64b_counter
- dtbinding_microchip_tc_g1
- dtbinding_microchip_tc_g1_counter
- dtbinding_microchip_tc_g2_counter
- dtbinding_microchip_tcc_g1
- dtbinding_microchip_tcc_g1_counter
- dtbinding_microcrystal_rv3028_counter
- dtbinding_microcrystal_rv3032_counter
- dtbinding_neorv32_gptmr
- dtbinding_nordic_nrf_timer
- dtbinding_nuvoton_npck_lct
- dtbinding_nuvoton_npcx_lct_base
- dtbinding_nuvoton_npcx_lct_v1

- dtbinding_nuvoton_npcx_lct_v2
- dtbinding_nxp_imx_gpt
- dtbinding_nxp_imx_qtmr
- dtbinding_nxp_imx_snvs_rtc
- dtbinding_nxp_imx_tmr
- dtbinding_nxp_irtc_wake_timer
- dtbinding_nxp_lpc_ctimer
- dtbinding_nxp_lpc_rtc_highres
- dtbinding_nxp_lpit
- dtbinding_nxp_lpit_channel
- dtbinding_nxp_lptmr
- dtbinding_nxp_mrt
- dtbinding_nxp_mrt_channel
- dtbinding_nxp_pit
- dtbinding_nxp_pit_channel
- dtbinding_nxp_s32_sys_timer
- dtbinding_nxp_stm
- dtbinding_nxp_sysctr
- dtbinding_nxp_tpm_timer
- dtbinding_nxp_tstmr
- dtbinding_nxp_wake_timer
- dtbinding_raspberrypi_pico_pit
- dtbinding_raspberrypi_pico_pit_channel
- dtbinding_raspberrypi_pico_timer
- dtbinding_realtek_ameba_counter
- dtbinding_realtek_bee_counter_rtc
- dtbinding_realtek_bee_counter_timer
- dtbinding_realtek_rts5912_slwtimer
- dtbinding_realtek_rts5912_timer
- dtbinding_renesas_ra_agt_counter
- dtbinding_renesas_ra_ulpt_counter
- dtbinding_renesas_rz_cmtw_counter
- dtbinding_renesas_rz_gtm_counter
- dtbinding_renesas_rza2m_ostm_counter
- dtbinding_renesas_smartbond_timer
- dtbinding_silabs_burtc_counter
- dtbinding_silabs_protimer
- dtbinding_silabs_timer_counter
- dtbinding_snps_dw_timers

- dtbinding_st_stm32_counter
- dtbinding_ti_cc23x0_lgpt
- dtbinding_ti_cc23x0_rtc
- dtbinding_ti_k3_rtc_counter
- dtbinding_ti_mspm0_timer_counter
- dtbinding_wch_adtm
- dtbinding_wch_gptm
- dtbinding_xlnx_ttc
- dtbinding_xlnx_ttc_counter
- dtbinding_xlnx_zynqmp_rtc
- dtbinding_zephyr_native_sim_counter

CPU (Central Processing Unit)

- dtbinding_adi_max32_m4f_cpu1
- dtbinding_adi_max32_rv32
- dtbinding_andestech_andescore_v5
- dtbinding_arm_armv8
- dtbinding_arm_cortex_a32
- dtbinding_arm_cortex_a320
- dtbinding_arm_cortex_a510
- dtbinding_arm_cortex_a53
- dtbinding_arm_cortex_a55
- dtbinding_arm_cortex_a57
- dtbinding_arm_cortex_a7
- dtbinding_arm_cortex_a72
- dtbinding_arm_cortex_a720
- dtbinding_arm_cortex_a76
- dtbinding_arm_cortex_a78
- dtbinding_arm_cortex_a9
- dtbinding_arm_cortex_m0
- dtbinding_arm_cortex_m0+
- dtbinding_arm_cortex_m1
- dtbinding_arm_cortex_m23
- dtbinding_arm_cortex_m3
- dtbinding_arm_cortex_m33
- dtbinding_arm_cortex_m33f
- dtbinding_arm_cortex_m4
- dtbinding_arm_cortex_m4f
- dtbinding_arm_cortex_m52

- dtbinding_arm_cortex_m52f
- dtbinding_arm_cortex_m55
- dtbinding_arm_cortex_m55f
- dtbinding_arm_cortex_m7
- dtbinding_arm_cortex_m85
- dtbinding_arm_cortex_m85f
- dtbinding_arm_cortex_r4
- dtbinding_arm_cortex_r4f
- dtbinding_arm_cortex_r5
- dtbinding_arm_cortex_r52
- dtbinding_arm_cortex_r5f
- dtbinding_arm_cortex_r7
- dtbinding_arm_cortex_r8
- dtbinding_arm_cortex_r82
- dtbinding_arm_cortex_r8f
- dtbinding_cdns_swerv_s400
- dtbinding_cdns_swerv_s420
- dtbinding_cdns_tensilica_xtensa_lx3
- dtbinding_cdns_tensilica_xtensa_lx4
- dtbinding_cdns_tensilica_xtensa_lx6
- dtbinding_cdns_tensilica_xtensa_lx7
- dtbinding_efinix_vexriscv_sapphire
- dtbinding_espressif_riscv
- dtbinding_espressif_xtensa_lx6
- dtbinding_espressif_xtensa_lx7
- dtbinding_gaisler_leon3
- dtbinding_infineon_tricore
- dtbinding_intel_alder_lake
- dtbinding_intel_apollo_lake
- dtbinding_intel_bartlett_lake
- dtbinding_intel_elkhart_lake
- dtbinding_intel_ish
- dtbinding_intel_lakemont
- dtbinding_intel_niosv
- dtbinding_intel_nova_lake
- dtbinding_intel_panther_lake
- dtbinding_intel_raptor_lake
- dtbinding_intel_wildcat_lake
- dtbinding_intel_x86

- dtbinding_intel_x86_64
- dtbinding_ite_riscv_ite
- dtbinding_litex_vexiiriscv_standard
- dtbinding_litex_vexriscv_standard
- dtbinding_lowrisc_ibex
- dtbinding_neorv32_cpu
- dtbinding_nordic_vpr
- dtbinding_nuclei_bumblebee
- dtbinding_openhwgroup_cva6
- dtbinding_openisa_ri5cy
- dtbinding_openisa_zero_ri5cy
- dtbinding_qcom_hexagon_v67
- dtbinding_qemu_riscv_virt
- dtbinding_renesas_rx
- dtbinding_renesas_rxv1
- dtbinding_renesas_rxv2
- dtbinding_renesas_rxv3
- dtbinding_renesas_rxv3e
- dtbinding_riscv
- dtbinding_sample_controller
- dtbinding_sensry_sy1xx
- dtbinding_sifive_e24
- dtbinding_sifive_e31
- dtbinding_sifive_e51
- dtbinding_sifive_s7
- dtbinding_sifive_u54
- dtbinding_snps_arcem
- dtbinding_snps_av5rhx
- dtbinding_snps_av5rmx
- dtbinding_spinalhdl_vexiiriscv
- dtbinding_spinalhdl_vexriscv
- dtbinding_telink_b91
- dtbinding_wch_qingke_v2
- dtbinding_wch_qingke_v3c
- dtbinding_wch_qingke_v3f
- dtbinding_wch_qingke_v4b
- dtbinding_wch_qingke_v4c
- dtbinding_wch_qingke_v4f
- dtbinding_wch_qingke_v5f

- dtbinding_xuantie_e907
- dtbinding_zephyr_native_sim_cpu

CPU frequency scaling

- dtbinding_zephyr_cpu_freq_thermal_cap

CRC (Cyclic Redundancy Check)

- dtbinding_ambiq_hw_crc32
- dtbinding_nxp_crc
- dtbinding_nxp_lpc_crc
- dtbinding_renesas_ra_crc
- dtbinding_sifli_sf32lb_crc
- dtbinding_silabs_gpcrc
- dtbinding_st_stm32_crc

Cryptographic accelerator

- dtbinding_arm_cryptocell_310
- dtbinding_arm_cryptocell_312
- dtbinding_atmel_ataes132a
- dtbinding_bflb_sec_eng_aes
- dtbinding_bflb_sec_eng_sha
- dtbinding_bflb_sec_eng_trng
- dtbinding_espressif_esp32_aes
- dtbinding_espressif_esp32_sha
- dtbinding_infineon_mxcrypto_crypto
- dtbinding_infineon_mxcrypto_trng
- dtbinding_infineon_mxcryptolite_crypto
- dtbinding_infineon_mxcryptolite_trng
- dtbinding_intel_adsp_sha
- dtbinding_ite_it51xxx_sha
- dtbinding_ite_it8xxx2_sha
- dtbinding_ite_it8xxx2_sha_v2
- dtbinding_microchip_aes_g1
- dtbinding_microchip_sha_g1_crypto
- dtbinding_microchip_xec_symcr
- dtbinding_nordic_nrf_ccm
- dtbinding_nordic_nrf_ecb
- dtbinding_nuvoton_npcx_sha
- dtbinding_nxp_els
- dtbinding_nxp_mcux_dcp

- dtbinding_nxp_s32_crypto_hse_mu
- dtbinding_raspberrypi_pico_sha256
- dtbinding_realtek_bee_aes
- dtbinding_realtek_bee_sha256
- dtbinding_realtek_rts5912_sha
- dtbinding_renesas_smartbond_crypto
- dtbinding_sifli_sf32lb_crypto
- dtbinding_silabs_gecko_semailbox
- dtbinding_silabs_series3_crypto
- dtbinding_silabs_si32_aes
- dtbinding_st_stm32_aes
- dtbinding_st_stm32_cryp
- dtbinding_st_stm32_hash
- dtbinding_st_stm32l4_aes
- dtbinding_st_stm32wba_pka
- dtbinding_ti_cc23x0_aes
- dtbinding_ti_mspm0_aes

DAC (Digital to Analog Converter)

- dtbinding_adi_ad5529r
- dtbinding_adi_ad559x_dac
- dtbinding_adi_ad5601
- dtbinding_adi_ad5611
- dtbinding_adi_ad5621
- dtbinding_adi_ad5628
- dtbinding_adi_ad5648
- dtbinding_adi_ad5668
- dtbinding_adi_ad5672
- dtbinding_adi_ad5674
- dtbinding_adi_ad5676
- dtbinding_adi_ad5679
- dtbinding_adi_ad5684
- dtbinding_adi_ad5686
- dtbinding_adi_ad5687
- dtbinding_adi_ad5689
- dtbinding_adi_ad5691
- dtbinding_adi_ad5692
- dtbinding_adi_ad5693
- dtbinding_adi_max22017_dac

- dtbinding_atmel_sam_dac
- dtbinding_atmel_sam0_dac
- dtbinding_atmel_samd5x_dac
- dtbinding_espressif_esp32_dac
- dtbinding_gd_gd32_dac
- dtbinding_infineon_autanalog_ctdac
- dtbinding_infineon_hppass_csg_dac
- dtbinding_lltc_ltc1660
- dtbinding_lltc_ltc1665
- dtbinding_microchip_dac_g1
- dtbinding_microchip_dac_g2
- dtbinding_microchip_mcp4725
- dtbinding_microchip_mcp4728
- dtbinding_nxp_dac12
- dtbinding_nxp_gau_dac
- dtbinding_nxp_hpdac
- dtbinding_nxp_kinetis_dac
- dtbinding_nxp_kinetis_dac32
- dtbinding_nxp_lpdac
- dtbinding_renesas_ra_dac
- dtbinding_renesas_ra_dac_global
- dtbinding_silabs_vdac
- dtbinding_st_stm32_dac
- dtbinding_ti_dac161s997
- dtbinding_ti_dac43508
- dtbinding_ti_dac43608
- dtbinding_ti_dac5311
- dtbinding_ti_dac53508
- dtbinding_ti_dac53608
- dtbinding_ti_dac60508
- dtbinding_ti_dac6311
- dtbinding_ti_dac63508
- dtbinding_ti_dac70508
- dtbinding_ti_dac7311
- dtbinding_ti_dac80508
- dtbinding_ti_dac8311
- dtbinding_ti_dac8411
- dtbinding_ti_dacx0501
- dtbinding_ti_mspm0_dac

- dtbinding_zephyr_dac_emul

DAI (Digital Audio Interface)

- dtbinding_amd_acp_sdw_dai
- dtbinding_amd_tdm_dai
- dtbinding_intel_adsp_dmic_vss
- dtbinding_intel_dai_dmic
- dtbinding_mediatek_afe
- dtbinding_nxp_dai_esai
- dtbinding_nxp_dai_micfil
- dtbinding_nxp_dai_sai

DALI (Digital Addressable Lighting Interface)

- dtbinding_zephyr_dali_pwm

Debug

- dtbinding_arm_armv7m_itm
- dtbinding_arm_armv8m_itm
- dtbinding_arm_stmesp
- dtbinding_nordic_coresight_nrf
- dtbinding_nordic_nrf_tbm
- dtbinding_silabs_pti

DFPMCCH

- dtbinding_intel_adsp_dfpmcch

DFPMCCU

- dtbinding_intel_adsp_dfpmccu

Disk

- dtbinding_nvme_controller
- dtbinding_virtio_blk
- dtbinding_zephyr_ftl_dhara
- dtbinding_zephyr_memc_ram_disk
- dtbinding_zephyr_ram_disk

Display

- dtbinding_adi_adv7535
- dtbinding_adi_adv7535_dsi
- dtbinding_charlieplex_led_matrix
- dtbinding_chipone_co5300_mipi_dsi
- dtbinding_chipone_co5300_mspi

- dtbinding_chipone_co5300_spi
- dtbinding_chipwealth_ch1115_spi
- dtbinding_chipwealth_ch1115_i2c
- dtbinding_chipwealth_ch1116_spi
- dtbinding_chipwealth_ch1116_i2c
- dtbinding_eink_ac057tc1
- dtbinding_eink_ed2208_doa
- dtbinding_eink_ed2208_gca
- dtbinding_espressif_esp_dsi_display
- dtbinding_fitipower_ek79007
- dtbinding_frida_nt35510
- dtbinding_ftdi_ft800
- dtbinding_galaxycore_gc9x01x
- dtbinding_himax_hx8353e
- dtbinding_himax_hx8379c
- dtbinding_himax_hx8394
- dtbinding_ilitek_ili9163c
- dtbinding_ilitek_ili9340
- dtbinding_ilitek_ili9341
- dtbinding_ilitek_ili9342c
- dtbinding_ilitek_ili9488
- dtbinding_ilitek_ili9806e
- dtbinding_ilitek_ili9881c
- dtbinding_intel_multiboot_framebuffer
- dtbinding_istech_ist3931
- dtbinding_ite_it8951
- dtbinding_jdi_lpm013m126
- dtbinding_led_strip_matrix
- dtbinding_levetop_lt7680
- dtbinding_maxim_max7219
- dtbinding_nordic_nrf_led_matrix
- dtbinding_nxp_dcnano_lcdif
- dtbinding_nxp_imx_elcdif
- dtbinding_nxp_imx_lcdifv2
- dtbinding_nxp_imx_lcdifv3
- dtbinding_orisetech_otm8009a
- dtbinding_philips_pcd8544
- dtbinding_qemu_ramfb
- dtbinding_raspberrypi_bcm2711_framebuffer

- dtbinding_raydium_rm67162
- dtbinding_raydium_rm67199
- dtbinding_raydium_rm68200
- dtbinding_raydium_rm692c9
- dtbinding_renesas_ra_glcdc
- dtbinding_renesas_rz_lcdc
- dtbinding_renesas_smartbond_display
- dtbinding_rocktech_rk043fn02h_ct
- dtbinding_sharp_ls0xx
- dtbinding_sifli_sf32lb_lcdc
- dtbinding_sinowealth_sh1106_i2c
- dtbinding_sinowealth_sh1106_spi
- dtbinding_sinowealth_sh1107_spi
- dtbinding_sinowealth_sh1107_i2c
- dtbinding_sinowealth_sh1122_i2c
- dtbinding_sinowealth_sh1122_mipi_dbi
- dtbinding_sitronix_st7305
- dtbinding_sitronix_st7306
- dtbinding_sitronix_st75256
- dtbinding_sitronix_st7567_mipi_dbi
- dtbinding_sitronix_st7567_i2c
- dtbinding_sitronix_st7586s
- dtbinding_sitronix_st7701
- dtbinding_sitronix_st7735r
- dtbinding_sitronix_st7789v
- dtbinding_sitronix_st7796s
- dtbinding_socionext_dpu
- dtbinding_solomon_ssd1305_spi
- dtbinding_solomon_ssd1305_i2c
- dtbinding_solomon_ssd1306_spi
- dtbinding_solomon_ssd1306_i2c
- dtbinding_solomon_ssd1306b_i2c
- dtbinding_solomon_ssd1306b_spi
- dtbinding_solomon_ssd1309_i2c
- dtbinding_solomon_ssd1309_spi
- dtbinding_solomon_ssd1315_i2c
- dtbinding_solomon_ssd1315_spi
- dtbinding_solomon_ssd1320_i2c
- dtbinding_solomon_ssd1320_mipi_dbi

- dtbinding_solomon_ssd1322
- dtbinding_solomon_ssd1325_mipi_dbi
- dtbinding_solomon_ssd1325_i2c
- dtbinding_solomon_ssd1327_mipi_dbi
- dtbinding_solomon_ssd1327_i2c
- dtbinding_solomon_ssd1331
- dtbinding_solomon_ssd1351
- dtbinding_solomon_ssd1357
- dtbinding_solomon_ssd1363_mipi_dbi
- dtbinding_solomon_ssd1363_i2c
- dtbinding_solomon_ssd1608
- dtbinding_solomon_ssd1673
- dtbinding_solomon_ssd1675a
- dtbinding_solomon_ssd1680
- dtbinding_solomon_ssd1681
- dtbinding_solomon_ssd1683
- dtbinding_st_neochrom_gpu2d
- dtbinding_st_stm32_dma2d
- dtbinding_st_stm32_ltdc
- dtbinding_ultrachip_uc8151d
- dtbinding_ultrachip_uc8175
- dtbinding_ultrachip_uc8176
- dtbinding_ultrachip_uc8179
- dtbinding_ultrachip_uc8253
- dtbinding_waveshare_dsi2dpi
- dtbinding_zephyr_displays
- dtbinding_zephyr_dummy_dc
- dtbinding_zephyr_hub12
- dtbinding_zephyr_panel_color_palette
- dtbinding_zephyr_panel_timing
- dtbinding_zephyr_sdl_dc

DMA (Direct Memory Access)

- dtbinding_adi_axi_dmac
- dtbinding_adi_max32_dma
- dtbinding_altr_msgdma
- dtbinding_amd_acp_host_dma
- dtbinding_amd_acp_sdw_dma
- dtbinding_amd_acp_tdm_dma

- dtbinding_amd_versal2_dma_1.0
- dtbinding_andestech_atcdmacx00
- dtbinding_arm_dma_pl330
- dtbinding_atmel_sam_xdmac
- dtbinding_atmel_sam0_dmac
- dtbinding_bflb_dma
- dtbinding_brcm_iproc_pax_dma_v1
- dtbinding_brcm_iproc_pax_dma_v2
- dtbinding_espressif_esp32_gdma
- dtbinding_gd_gd32_dma
- dtbinding_gd_gd32_dma_v1
- dtbinding_infineon_dma
- dtbinding_infineon_dmac
- dtbinding_infineon_mdma
- dtbinding_infineon_xmc4xxx_dma
- dtbinding_intel_adsp_gpdma
- dtbinding_intel_adsp_hda_dmic_cap
- dtbinding_intel_adsp_hda_host_in
- dtbinding_intel_adsp_hda_host_out
- dtbinding_intel_adsp_hda_link_in
- dtbinding_intel_adsp_hda_link_out
- dtbinding_intel_adsp_hda_ssp_cap
- dtbinding_intel_lpss
- dtbinding_intel_sedi_dma
- dtbinding_microchip_dmac_g1_dma
- dtbinding_microchip_dmac_g2_dma
- dtbinding_microchip_dmac_g3_dma
- dtbinding_microchip_xec_dmac
- dtbinding_nuvoton_npcx_gdma
- dtbinding_nxp_4ch_dma
- dtbinding_nxp_edma
- dtbinding_nxp_gdma
- dtbinding_nxp_lpc_dma
- dtbinding_nxp_mcux_edma
- dtbinding_nxp_pxp
- dtbinding_nxp_sdma
- dtbinding_nxp_smartdma
- dtbinding_nxp_sof_host_dma
- dtbinding_raspberrypi_pico_dma

- dtbinding_realtek_ameba_gdma
- dtbinding_realtek_bee_dma
- dtbinding_renesas_ra_dma
- dtbinding_renesas_rz_dmac
- dtbinding_renesas_rz_dmac_b
- dtbinding_renesas_rza2m_dma
- dtbinding_renesas_smartbond_dma
- dtbinding_sifli_sf32lb_dmac
- dtbinding_silabs_gpdma
- dtbinding_silabs_ldma
- dtbinding_silabs_si32_dma
- dtbinding_silabs_udma
- dtbinding_snps_designware_dma
- dtbinding_snps_designware_dma_axi
- dtbinding_st_stm32_bdma
- dtbinding_st_stm32_dma
- dtbinding_st_stm32_dma_v1
- dtbinding_st_stm32_dma_v2
- dtbinding_st_stm32_dma_v2bis
- dtbinding_st_stm32_dmamux
- dtbinding_st_stm32u5_dma
- dtbinding_ti_cc23x0_dma
- dtbinding_ti_mspm0_dma
- dtbinding_wch_wch_dma
- dtbinding_xlnx_axi_dma_1.00.a
- dtbinding_xlnx_eth_dma
- dtbinding_xlnx_zynqmp_dma_1.0
- dtbinding_zephyr_dma_emul

DSA (Distributed Switch Architecture)

- dtbinding_nxp_netc_switch
- dtbinding_zephyr_dsa_port

DSP (Digital Signal Processor)

- dtbinding_nxp_powerquad

EDAC (Error Detection and Correction)

- dtbinding_intel_ibecc
- dtbinding_nxp_eim
- dtbinding_nxp_erm
- dtbinding_nxp_mecc
- dtbinding_xlnx_zynqmp_ddrc_2.40a

ESPI (Enhanced Serial Peripheral Interface)

- dtbinding_intel_espi_peci
- dtbinding_ite_it8xxx2_espi
- dtbinding_microchip_xec_espi
- dtbinding_microchip_xec_espi_host_dev
- dtbinding_microchip_xec_espi_saf
- dtbinding_microchip_xec_espi_saf_v2
- dtbinding_microchip_xec_espi_v2
- dtbinding_microchip_xec_espi_vw_routing
- dtbinding_nuvoton_npcx_espi
- dtbinding_nuvoton_npcx_espi_taf
- dtbinding_nuvoton_npcx_espi_vw_conf
- dtbinding_nuvoton_npcx_host_sub
- dtbinding_nuvoton_npcx_host_uart
- dtbinding_nxp_espi
- dtbinding_nxp_espi_taf
- dtbinding_realtek_rts5912_espi
- dtbinding_zephyr_espi_emul_controller

Ethernet

- dtbinding_adi_adin1100_phy
- dtbinding_adi_adin1110
- dtbinding_adi_adin1140
- dtbinding_adi_adin1140_mdio
- dtbinding_adi_adin1140_phy
- dtbinding_adi_adin2111
- dtbinding_adi_adin2111_mdio
- dtbinding_adi_adin2111_phy
- dtbinding_atmel_sam_gmac
- dtbinding_atmel_sam_mdio
- dtbinding_atmel_sam0_gmac
- dtbinding_brcm_genet
- dtbinding_brcm_genet_mdio

- dtbinding_davicom_dm8806_phy
- dtbinding_davicom_dm9051
- dtbinding_espressif_esp32_eth
- dtbinding_ethernet_phy
- dtbinding_ethernet_phy_fixed_link
- dtbinding_infineon_xmc4xxx_ethernet
- dtbinding_intel_e1000
- dtbinding_intel_eth_plat
- dtbinding_intel_igc_mac
- dtbinding_intel_igc_mdio
- dtbinding_litex_liteeth
- dtbinding_litex_liteeth_mdio
- dtbinding_maxlinear_gpy111
- dtbinding_microchip_enc28j60
- dtbinding_microchip_enc424j600
- dtbinding_microchip_gmac_g1_eth
- dtbinding_microchip_gmac_g1_mdio
- dtbinding_microchip_ksz8081
- dtbinding_microchip_ksz9131
- dtbinding_microchip_lan865x
- dtbinding_microchip_lan865x_mdio
- dtbinding_microchip_lan8742
- dtbinding_microchip_lan8840
- dtbinding_microchip_lan9250
- dtbinding_microchip_sam_ethernet_controller
- dtbinding_microchip_t1s_phy
- dtbinding_microchip_vsc8541
- dtbinding_motorcomm_yt8521
- dtbinding_motorcomm_yt8531
- dtbinding_nuvoton_numaker_ethernet
- dtbinding_nxp_enet
- dtbinding_nxp_enet_mac
- dtbinding_nxp_enet_mdio
- dtbinding_nxp_enet_ptp_clock
- dtbinding_nxp_enet_qos
- dtbinding_nxp_imx_netc
- dtbinding_nxp_imx_netc_blk_ctrl
- dtbinding_nxp_imx_netc_emdio
- dtbinding_nxp_imx_netc_psi

- dtbinding_nxp_imx_netc_vsi
- dtbinding_nxp_netc_ptp_clock
- dtbinding_nxp_s32_gmac
- dtbinding_nxp_s32_netc_emdio
- dtbinding_nxp_s32_netc_psi
- dtbinding_nxp_s32_netc_vsi
- dtbinding_nxp_t1s_phy
- dtbinding_nxp_tja1103
- dtbinding_nxp_tja11xx
- dtbinding_qca_ar8031
- dtbinding_realtek_rtl8211f
- dtbinding_renesas_ra_eswm
- dtbinding_renesas_ra_ethernet
- dtbinding_renesas_ra_ethernet_rmac
- dtbinding_renesas_ra_mdio
- dtbinding_renesas_ra_mdio_rmac
- dtbinding_rockchip_rk3588_gmac
- dtbinding_sensry_sy1xx_mac
- dtbinding_sensry_sy1xx_mdio
- dtbinding_siemens_ivshmem_eth
- dtbinding_silabs_gecko_ethernet
- dtbinding_smsc_lan91c111
- dtbinding_smsc_lan91c111_mdio
- dtbinding_smsc_lan9220
- dtbinding_snps_designware_ethernet
- dtbinding_snps_dwcxgmac
- dtbinding_snps_dwcxgmac_mdio
- dtbinding_snps_dwmac
- dtbinding_snps_dwmac_mdio
- dtbinding_snps_dwmac_ptp_clock
- dtbinding_snps_ethernet_cyclonev
- dtbinding_st_stm32_ethernet
- dtbinding_st_stm32h5_ethernet
- dtbinding_st_stm32h7_ethernet
- dtbinding_st_stm32mp13_ethernet
- dtbinding_st_stm32n6_ethernet
- dtbinding_ti_dp83825
- dtbinding_ti_dp83867
- dtbinding_ti_stellaris_ethernet

- dtbinding_virtio_net
- dtbinding_wch_ch9120
- dtbinding_wch_ethernet
- dtbinding_wiznet_w5100s
- dtbinding_wiznet_w5500
- dtbinding_wiznet_w6100
- dtbinding_wiznet_w6300
- dtbinding_xlnx_axi_ethernet_1.00.a
- dtbinding_xlnx_axi_ethernet_1.00.a_mdio
- dtbinding_xlnx_gem
- dtbinding_xlnx_gem_mdio
- dtbinding_zephyr_cdc_ecm_ethernet
- dtbinding_zephyr_cdc_ncm_ethernet
- dtbinding_zephyr_mdio_gpio
- dtbinding_zephyr_native_ptp_clock
- dtbinding_zephyr_native_tap

File system

- dtbinding_zephyr_fstab
- dtbinding_zephyr_fstab_ext2
- dtbinding_zephyr_fstab_fatfs
- dtbinding_zephyr_fstab_littlefs

Firmware

- dtbinding_arm_scmi
- dtbinding_arm_scmi_clock
- dtbinding_arm_scmi_pinctrl
- dtbinding_arm_scmi_power
- dtbinding_arm_scmi_reset
- dtbinding_arm_scmi_shmem
- dtbinding_arm_scmi_smc
- dtbinding_arm_scmi_system
- dtbinding_nxp_scmi_cpu
- dtbinding_qemu_fw_cfg_ioport
- dtbinding_qemu_fw_cfg_mmio
- dtbinding_raspberrypi_bcm283x_firmware
- dtbinding_ti_k2g_sci

Flash controller

- dtbinding_adi_max32_flash_controller
- dtbinding_adi_max32_spixf_nor
- dtbinding_aesc_spi_flash_controller
- dtbinding_altr_nios2_qspi_nor
- dtbinding_ambiq_flash_controller
- dtbinding_atmel_sam_flash_controller
- dtbinding_atmel_sam0_nvmctrl
- dtbinding_atmel_sam4l_flashcalw_controller
- dtbinding_cdns_nand
- dtbinding_cdns_qspi_nor
- dtbinding_cypress_psoc6_flash_controller
- dtbinding_espressif_esp32_flash_controller
- dtbinding_gd_gd32_flash_controller
- dtbinding_infineon_flash_controller
- dtbinding_infineon_qspi_flash
- dtbinding_infineon_rram_controller
- dtbinding_infineon_xmc4xxx_flash_controller
- dtbinding_intel_pflash_cfi01
- dtbinding_ite_it51xxx_manual_flash_1k
- dtbinding_ite_it8xxx2_flash_controller
- dtbinding_microchip_igloo2_envm_controller
- dtbinding_microchip_nvmctrl_g1_flash
- dtbinding_microchip_nvmctrl_g2
- dtbinding_microchip_nvmctrl_g3
- dtbinding_microchip_smartfusion2_flash_controller
- dtbinding_micron_mt25qu02g
- dtbinding_nordic_nrf_mramc
- dtbinding_nordic_nrf_qspi
- dtbinding_nordic_nrf51_flash_controller
- dtbinding_nordic_nrf52_flash_controller
- dtbinding_nordic_nrf53_flash_controller
- dtbinding_nordic_nrf91_flash_controller
- dtbinding_nordic_rram_controller
- dtbinding_nuvoton_npcx_fiu_nor
- dtbinding_nuvoton_npcx_fiu_qspi
- dtbinding_nuvoton_numaker_fmc
- dtbinding_nuvoton_numaker_rmc
- dtbinding_nxp_c40_flash_controller

- dtbinding_nxp_iap_fmc11
- dtbinding_nxp_iap_fmc54
- dtbinding_nxp_iap_fmc55
- dtbinding_nxp_iap_fmc553
- dtbinding_nxp_iap_fmc84x
- dtbinding_nxp_kinetis_ftfa
- dtbinding_nxp_kinetis_ftfc
- dtbinding_nxp_kinetis_ftfe
- dtbinding_nxp_kinetis_ftfl
- dtbinding_nxp_msf1
- dtbinding_nxp_xspi_nor
- dtbinding_openisa_rv32m1_ftfe
- dtbinding_raspberrypi_pico_flash_controller
- dtbinding_realtek_ameba_flash_controller
- dtbinding_realtek_bee_nor_flash_controller
- dtbinding_realtek_rts5912_flash_controller
- dtbinding_renesas_ra_faci
- dtbinding_renesas_ra_fcb
- dtbinding_renesas_ra_flash_hp_controller
- dtbinding_renesas_ra_flash_lp_controller
- dtbinding_renesas_ra_mram_controller
- dtbinding_renesas_ra_ospi_b_nor
- dtbinding_renesas_rx_flash
- dtbinding_renesas_rz_qspi_spibsc
- dtbinding_renesas_rz_qspi_xspi
- dtbinding_renesas_rza2m_qspi_spibsc
- dtbinding_renesas_smartbond_flash_controller
- dtbinding_silabs_gecko_flash_controller
- dtbinding_silabs_series2_flash_controller
- dtbinding_silabs_series3_extmem
- dtbinding_silabs_si32_flash_controller
- dtbinding_silabs_siwx91x_flash_controller
- dtbinding_st_stm32_flash_controller
- dtbinding_st_stm32_ospi_nor
- dtbinding_st_stm32_qspi_nor
- dtbinding_st_stm32_xspi_nor
- dtbinding_st_stm32c5_flash_controller
- dtbinding_st_stm32f1_flash_controller
- dtbinding_st_stm32f2_flash_controller

- dtbinding_st_stm32f4_flash_controller
- dtbinding_st_stm32f7_flash_controller
- dtbinding_st_stm32g0_flash_controller
- dtbinding_st_stm32g4_flash_controller
- dtbinding_st_stm32h7_flash_controller
- dtbinding_st_stm32l4_flash_controller
- dtbinding_st_stm32l5_flash_controller
- dtbinding_st_stm32u3_flash_controller
- dtbinding_st_stm32wb_flash_controller
- dtbinding_st_stm32wb0_flash_controller
- dtbinding_st_stm32wba_flash_controller
- dtbinding_telink_b91_flash_controller
- dtbinding_ti_cc13xx_cc26xx_flash_controller
- dtbinding_ti_cc23x0_flash_controller
- dtbinding_ti_mspm0_flash_controller
- dtbinding_ti_stellaris_flash_controller
- dtbinding_wch_ch32v_flash_controller
- dtbinding_zephyr_semihost_flash
- dtbinding_zephyr_sim_flash

FPGA (Field Programmable Gate Array)

- dtbinding_lattice_ice40_fpga
- dtbinding_lattice_ice40_fpga_base
- dtbinding_lattice_ice40_fpga_bitbang
- dtbinding_renesas_slg47105
- dtbinding_renesas_slg47115
- dtbinding_renesas_slg471x5
- dtbinding_renesas_slg47910
- dtbinding_xlnx_fpga

Fuel gauge

- dtbinding_adi_adp5360_fuel_gauge
- dtbinding_adi_ltc2959
- dtbinding_hycon_hy4245
- dtbinding_maxim_max17048
- dtbinding_onnn_lc709203f
- dtbinding_sbs_default_sbs_gauge
- dtbinding_sbs_sbs_gauge_new_api
- dtbinding_silergy_sy24561
- dtbinding_ti_bq27z746

- dtbinding_ti_bq40z50
- dtbinding_x_powers_axp2101_fuel_gauge
- dtbinding_zephyr_fuel_gauge_composite

GNSS (Global Navigation Satellite System)

- dtbinding_ericsson_f5521gw
- dtbinding_globaltop_pa6h
- dtbinding_gnss_nmea_generic
- dtbinding_luatos_air530z
- dtbinding_quectel_lc26g
- dtbinding_quectel_lc76g
- dtbinding_quectel_lc86g
- dtbinding_u_blox_f9p
- dtbinding_u_blox_m10
- dtbinding_u_blox_m8
- dtbinding_zephyr_gnss_emul

GPIO (General Purpose Input/Output) & Headers

- dtbinding_adafruit_feather_header
- dtbinding_adi_ad559x_gpio
- dtbinding_adi_adp5585_gpio
- dtbinding_adi_max14906_gpio
- dtbinding_adi_max14915_gpio
- dtbinding_adi_max14916_gpio
- dtbinding_adi_max14917_gpio
- dtbinding_adi_max22017_gpio
- dtbinding_adi_max22190_gpio
- dtbinding_adi_max22199_gpio
- dtbinding_adi_max32_gpio
- dtbinding_adi_sdp_120
- dtbinding_aesc_gpio
- dtbinding_allwinner_sun50i_h618_gpio
- dtbinding_allwinner_sunxi_gpio
- dtbinding_altr_pio_1.0
- dtbinding_ambiq_gpio
- dtbinding_ambiq_gpio_bank
- dtbinding_ambiq_header
- dtbinding_andestech_atcgpio100
- dtbinding_arducam_dvp_20pin_connector
- dtbinding_arducam_ffc_40pin_connector

- dtbinding_arduino_header_r3
- dtbinding_arduino_mega_header
- dtbinding_arduino_mkr_header
- dtbinding_arduino_nano_header
- dtbinding_arm_cmsdk_gpio
- dtbinding_arm_mmio32_gpio
- dtbinding_atmel_sam_gpio
- dtbinding_atmel_sam0_gpio
- dtbinding_atmel_sam4l_gpio
- dtbinding_atmel_xplained_header
- dtbinding_atmel_xplained_pro_header
- dtbinding_awinic_aw9523b_gpio
- dtbinding_bflb_bl60x_70x_gpio
- dtbinding_bflb_bl61x_808_gpio
- dtbinding_brcm_bcm2711_gpio
- dtbinding_brcm_brcmstb_gpio
- dtbinding_brcm_iproc_gpio
- dtbinding_cdac_thejas32_gpio
- dtbinding_cypress_cy8c95xx_gpio
- dtbinding_cypress_cy8c95xx_gpio_port
- dtbinding_cypress_psoc6_gpio
- dtbinding_cypress_psoc6_hsiom
- dtbinding_digilent_pmod
- dtbinding_diodes_pi4ioe5v6408
- dtbinding_efinix_sapphire_gpio
- dtbinding_elan_em32_gpio
- dtbinding_ene_kb106x_gpio
- dtbinding_ene_kb1200_gpio
- dtbinding_esp_01_header
- dtbinding_espressif_esp32_gpio
- dtbinding_espressif_esp32_lpgpio
- dtbinding_fcs_fxl6408
- dtbinding_fobe_quill_header
- dtbinding_gaisler_grgpio
- dtbinding_gd_gd32_gpio
- dtbinding_gpio_mmio_latch
- dtbinding_grove_header
- dtbinding_infineon_cyw43_gpio
- dtbinding_infineon_gpio

- dtbinding_infineon_shared_gpio
- dtbinding_infineon_tle9104_gpio
- dtbinding_infineon_xmc4xxx_gpio
- dtbinding_intel_gpio
- dtbinding_intel_sedi_gpio
- dtbinding_ite_it51xxx_gpio
- dtbinding_ite_it8801_gpio
- dtbinding_ite_it8xxx2_gpio
- dtbinding_ite_it8xxx2_gpio_v2
- dtbinding_ite_it8xxx2_gpiokscan
- dtbinding_linaro_96b_lscon_1v8
- dtbinding_linaro_96b_lscon_3v3
- dtbinding_litex_gpio
- dtbinding_m5stack_atom_header
- dtbinding_m5stack_m5pm1_gpio
- dtbinding_m5stack_mbus_header
- dtbinding_m5stack_stamps3_header
- dtbinding_microbit_edge_connector
- dtbinding_microchip_mcp23008
- dtbinding_microchip_mcp23009
- dtbinding_microchip_mcp23016
- dtbinding_microchip_mcp23017
- dtbinding_microchip_mcp23018
- dtbinding_microchip_mcp23s08
- dtbinding_microchip_mcp23s09
- dtbinding_microchip_mcp23s17
- dtbinding_microchip_mcp23s18
- dtbinding_microchip_mec5_gpio
- dtbinding_microchip_mpfs_gpio
- dtbinding_microchip_port_g1_gpio
- dtbinding_microchip_sam_pio4
- dtbinding_microchip_xec_gpio
- dtbinding_microchip_xec_gpio_v2
- dtbinding_microchip_xpro_header
- dtbinding_mikro_bus
- dtbinding_mr_mcxn_t1_header
- dtbinding_neorv32_gpio
- dtbinding_nordic_expansion_board_header
- dtbinding_nordic_npm10xx_gpio

- dtbinding_nordic_npm1300_gpio
- dtbinding_nordic_npm1304_gpio
- dtbinding_nordic_npm2100_gpio
- dtbinding_nordic_npm6001_gpio
- dtbinding_nordic_nrf_gpio
- dtbinding_nordic_nrf_gpio_forwarder
- dtbinding_nordic_nrf_gpiote
- dtbinding_nordic_nrf71_p4_gpio
- dtbinding_nordic_thingy53_edge_connector
- dtbinding_nuvoton_nct38xx_gpio
- dtbinding_nuvoton_nct38xx_gpio_alert
- dtbinding_nuvoton_nct38xx_gpio_port
- dtbinding_nuvoton_npcx_gpio
- dtbinding_nuvoton_numaker_gpio
- dtbinding_nuvoton_numicro_gpio
- dtbinding_nxp_cam_44pins_connector
- dtbinding_nxp_gpio_cluster
- dtbinding_nxp_i2c_tsc_fpc
- dtbinding_nxp_imx_gpio
- dtbinding_nxp_imx_gpio_v2
- dtbinding_nxp_imx_rgpio
- dtbinding_nxp_kinetis_gpio
- dtbinding_nxp_lcd_8080
- dtbinding_nxp_lcd_pmod
- dtbinding_nxp_lpc_gpio
- dtbinding_nxp_lpc_gpio_port
- dtbinding_nxp_lpc11u6x_gpio
- dtbinding_nxp_parallel_lcd_connector
- dtbinding_nxp_pca6408
- dtbinding_nxp_pca6416
- dtbinding_nxp_pca9538
- dtbinding_nxp_pca9539
- dtbinding_nxp_pca9554
- dtbinding_nxp_pca9555
- dtbinding_nxp_pca95xx
- dtbinding_nxp_pcal6408
- dtbinding_nxp_pcal6408a
- dtbinding_nxp_pcal6416
- dtbinding_nxp_pcal6416a

- dtbinding_nxp_pcal6524
- dtbinding_nxp_pcal6534
- dtbinding_nxp_pcal9538
- dtbinding_nxp_pcal9539
- dtbinding_nxp_pcal9722
- dtbinding_nxp_pcf857x
- dtbinding_nxp_sc18im704_gpio
- dtbinding_nxp_sc18is606_gpio
- dtbinding_nxp_siul2_gpio
- dtbinding_openisa_rv32m1_gpio
- dtbinding_panasonic_reduced_arduino_header
- dtbinding_particle_gen3_header
- dtbinding_quicklogic_eos_s3_gpio
- dtbinding_raspberrypi_bcm283x_gpio
- dtbinding_raspberrypi_csi_connector
- dtbinding_raspberrypi_pico_gpio
- dtbinding_raspberrypi_pico_gpio_port
- dtbinding_raspberrypi_pico_header
- dtbinding_raspberrypi_rp1_gpio
- dtbinding_raspberrypi_40pins_header
- dtbinding_realtek_ameba_gpio
- dtbinding_realtek_bee_gpio
- dtbinding_realtek_rts5817_gpio
- dtbinding_realtek_rts5912_gpio
- dtbinding_renesas_ra_gpio_ioport
- dtbinding_renesas_ra_gpio_mipi_header
- dtbinding_renesas_ra_parallel_graphics_header
- dtbinding_renesas_rcar_gpio
- dtbinding_renesas_rx_gpio
- dtbinding_renesas_rx_gpio_ioport
- dtbinding_renesas_rz_gpio
- dtbinding_renesas_rz_gpio_common
- dtbinding_renesas_rz_gpio_common_v2
- dtbinding_renesas_rz_gpio_common_v3
- dtbinding_renesas_rz_gpio_int
- dtbinding_renesas_rz_gpio_mipi_header
- dtbinding_renesas_rza2m_gpio
- dtbinding_renesas_rza2m_gpio_int
- dtbinding_renesas_rzt2m_gpio

- dtbinding_renesas_rzt2m_gpio_common
- dtbinding_renesas_smartbond_gpio
- dtbinding_richtek_rt1718s
- dtbinding_richtek_rt1718s_gpio_port
- dtbinding_rohm_bd8lb600fs_gpio
- dtbinding_seeed_xiao_expand
- dtbinding_seeed_xiao_gpio
- dtbinding_semtech_sx1509b
- dtbinding_sensry_sy1xx_gpio
- dtbinding_sifive_gpio0
- dtbinding_sifli_sf32lb_gpio
- dtbinding_sifli_sf32lb_gpio_parent
- dtbinding_silabs_exp_header
- dtbinding_silabs_gecko_gpio
- dtbinding_silabs_gecko_gpio_port
- dtbinding_silabs_gpio
- dtbinding_silabs_gpio_port
- dtbinding_silabs_si32_gpio
- dtbinding_silabs_siwx91x_gpio
- dtbinding_silabs_siwx91x_gpio_port
- dtbinding_silabs_siwx91x_gpio_uulp
- dtbinding_snps_creg_gpio
- dtbinding_snps_designware_gpio
- dtbinding_solderedelectronics_easyc_connector
- dtbinding_sparkfun_micromod_gpio
- dtbinding_sparkfun_pro_micro_gpio
- dtbinding_st_dsi_lcd_qsh_030_connector
- dtbinding_st_dvp_cam_zif_30_connector
- dtbinding_st_m2_memory_connector
- dtbinding_st_mfxstm32l152
- dtbinding_st_stm32_gpio
- dtbinding_st_stm32mp2_gpio
- dtbinding_st_stmod_plus_connector
- dtbinding_st_stmpe1600
- dtbinding_st_morpho_header
- dtbinding_st_zio_header
- dtbinding_stemma_qt_connector
- dtbinding_swj_connector
- dtbinding_telink_b91_gpio

- dtbinding_ti_ads1x4s0x_gpio
- dtbinding_ti_boosterpack_header
- dtbinding_ti_cc13xx_cc26xx_gpio
- dtbinding_ti_cc23x0_gpio
- dtbinding_ti_cc32xx_gpio
- dtbinding_ti_davinci_gpio
- dtbinding_ti_davinci_gpio_nexus
- dtbinding_ti_lmp90xxx_gpio
- dtbinding_ti_mspm0_gpio
- dtbinding_ti_sn74hc595
- dtbinding_ti_stellaris_gpio
- dtbinding_ti_tca6424a
- dtbinding_ti_tca9538
- dtbinding_ti_tca9554
- dtbinding_ti_tla2528_gpio
- dtbinding_virtio_gpio
- dtbinding_wch_ch5xx_gpio
- dtbinding_wch_gpio
- dtbinding_weact_dcmi_camera_connector
- dtbinding_wisblock_io_slot
- dtbinding_wisblock_sensor_slot
- dtbinding_x_powers_axp192_gpio
- dtbinding_xlnx_ps_gpio
- dtbinding_xlnx_ps_gpio_bank
- dtbinding_xlnx_xps_gpio_1.00.a
- dtbinding_xlnx_xps_gpio_1.00.a_gpio2
- dtbinding_zephyr_gpio_emul
- dtbinding_zephyr_gpio_emul_sdl

Haptics

- dtbinding_cirrus_cs40l26_spi
- dtbinding_cirrus_cs40l26_i2c
- dtbinding_cirrus_cs40l27_i2c
- dtbinding_cirrus_cs40l27_spi
- dtbinding_cirrus_cs40l50_i2c
- dtbinding_cirrus_cs40l50_spi
- dtbinding_cirrus_cs40l51_spi
- dtbinding_cirrus_cs40l51_i2c
- dtbinding_cirrus_cs40l52_i2c

- dtbinding_cirrus_cs40l52_spi
- dtbinding_cirrus_cs40l53_i2c
- dtbinding_cirrus_cs40l53_spi
- dtbinding_ti_drv2605
- dtbinding_titanmec_tm6605

Hardware information

- dtbinding_atmel_sam_rstc
- dtbinding_atmel_sam4l_uid
- dtbinding_cypress_psoc6_uid
- dtbinding_litex_dna0
- dtbinding_microchip_hwinfo_g1
- dtbinding_microchip_xec_ecs
- dtbinding_nxp_imx_src_rev2
- dtbinding_nxp_lpc_pmc_hwinfo
- dtbinding_nxp_lpc_uid
- dtbinding_nxp_mc_rgm
- dtbinding_nxp_otp_uid
- dtbinding_nxp_rcm_hwinfo
- dtbinding_nxp_rstctl_hwinfo
- dtbinding_nxp_sim_uuid
- dtbinding_wch_esig
- dtbinding_zephyr_hwinfo_nvmem

Hardware spinlock

- dtbinding_nxp_sema42
- dtbinding_sqn_hwspinlock

HDA (High Definition Audio)

- dtbinding_intel_hda_dai

I2C (Inter-Integrated Circuit)

- dtbinding_adi_max32_i2c
- dtbinding_altr_nios2_i2c
- dtbinding_ambiq_i2c
- dtbinding_ambiq_ios_i2c
- dtbinding_andestech_atciic100
- dtbinding_arm_versatile_i2c
- dtbinding_asmedia_asm2364
- dtbinding_atmel_sam_i2c_twi

- dtbinding_atmel_sam_i2c_twihs
- dtbinding_atmel_sam_i2c_twim
- dtbinding_atmel_sam0_i2c
- dtbinding_bflb_i2c
- dtbinding_brcm_bcm2711_i2c
- dtbinding_brcm_iproc_i2c
- dtbinding_cdns_i2c
- dtbinding_ene_kb106x_i2c
- dtbinding_ene_kb1200_i2c
- dtbinding_espressif_esp32_i2c
- dtbinding_fsl_imx21_i2c
- dtbinding_gd_gd32_i2c
- dtbinding_gpio_i2c
- dtbinding_gpio_i2c_switch
- dtbinding_infineon_i2c
- dtbinding_infineon_xmc4xxx_i2c
- dtbinding_intel_sedi_i2c
- dtbinding_ite_enhance_i2c
- dtbinding_ite_it51xxx_i2c
- dtbinding_ite_it8xxx2_i2c
- dtbinding_litex_i2c
- dtbinding_litex_litei2c
- dtbinding_microchip_mpfs_i2c
- dtbinding_microchip_sercom_g1_i2c
- dtbinding_microchip_xec_i2c
- dtbinding_microchip_xec_i2c_v2
- dtbinding_nordic_nrf_twi
- dtbinding_nordic_nrf_twim
- dtbinding_nordic_nrf_twis
- dtbinding_nuvoton_npcx_i2c_ctrl
- dtbinding_nuvoton_npcx_i2c_port
- dtbinding_nuvoton_numaker_i2c
- dtbinding_nxp_ii2c
- dtbinding_nxp_kinetis_i2c
- dtbinding_nxp_lpc_i2c
- dtbinding_nxp_lpc11u6x_i2c
- dtbinding_nxp_lpi2c
- dtbinding_nxp_sc18im704_i2c
- dtbinding_openisa_rv32m1_lpi2c

- dtbinding_raspberrypi_pico_i2c
- dtbinding_realtek_ameba_i2c
- dtbinding_realtek_bee_i2c
- dtbinding_realtek_rts5912_i2c
- dtbinding_renesas_ra_i2c_sci
- dtbinding_renesas_ra_i2c_sci_b
- dtbinding_renesas_ra_iic
- dtbinding_renesas_rcar_i2c
- dtbinding_renesas_rx_i2c
- dtbinding_renesas_rz_iic
- dtbinding_renesas_rz_riic
- dtbinding_renesas_rza2m_riic
- dtbinding_renesas_smartbond_i2c
- dtbinding_sensry_sy1xx_i2c
- dtbinding_sifive_i2c0
- dtbinding_sifli_sf32lb_i2c
- dtbinding_silabs_gecko_i2c
- dtbinding_silabs_i2c
- dtbinding_snps_designware_i2c
- dtbinding_st_stm32_i2c_v1
- dtbinding_st_stm32_i2c_v2
- dtbinding_telink_b91_i2c
- dtbinding_ti_cc13xx_cc26xx_i2c
- dtbinding_ti_cc23x0_i2c
- dtbinding_ti_cc32xx_i2c
- dtbinding_ti_omap_i2c
- dtbinding_ti_tca9544a
- dtbinding_ti_tca9546a
- dtbinding_ti_tca9548a
- dtbinding_virtio_i2c
- dtbinding_wch_i2c
- dtbinding_xlnx_xps_iic_2.00.a
- dtbinding_xlnx_xps_iic_2.1
- dtbinding_zephyr_i2c_dump_allowlist
- dtbinding_zephyr_i2c_emul_controller
- dtbinding_zephyr_i2c_target_tmp103

I2S (Inter-Integrated Circuit Sound)

- dtbinding_adi_max32_i2s
- dtbinding_ambiq_i2s
- dtbinding_espressif_esp32_i2s
- dtbinding_infineon_i2s
- dtbinding_intel_cavs_i2s
- dtbinding_intel_ssp
- dtbinding_intel_ssp_dai
- dtbinding_intel_ssp_sspbase
- dtbinding_litex_i2s
- dtbinding_nordic_nrf_i2s
- dtbinding_nordic_nrf_tdm
- dtbinding_nxp_lpc_i2s
- dtbinding_nxp_mcux_i2s
- dtbinding_renesas_ra_i2s_ssie
- dtbinding_silabs_siwx91x_i2s
- dtbinding_st_stm32_i2s
- dtbinding_st_stm32_sai
- dtbinding_st_stm32h7_i2s
- dtbinding_zephyr_native_sim_i2s

I3C (Improved Inter-Integrated Circuit)

- dtbinding_adi_max32_i3c
- dtbinding_cdns_i3c
- dtbinding_ite_it51xxx_i3cm
- dtbinding_ite_it51xxx_i3cs
- dtbinding_microchip_xec_i3c
- dtbinding_nuvoton_npcx_i3c
- dtbinding_nxp_mcux_i3c
- dtbinding_renesas_ra_i3c
- dtbinding_snps_designware_i3c
- dtbinding_st_stm32_i3c

IEEE 802.15.4

- dtbinding_atmel_rf2xx
- dtbinding_bflb_ieee802154
- dtbinding_decawave_dw1000
- dtbinding_espressif_esp32_ieee802154
- dtbinding_nordic_nrf_ieee802154
- dtbinding_nxp_kw41z_ieee802154

- dtbinding_nxp_mcr20a
- dtbinding_nxp_mcxw_ieee802154
- dtbinding_silabs_efr32_ieee802154
- dtbinding_st_stm32wba_ieee802154
- dtbinding_telink_b91_zb
- dtbinding_ti_cc1200
- dtbinding_ti_cc13xx_cc26xx_ieee802154
- dtbinding_ti_cc13xx_cc26xx_ieee802154_subghz
- dtbinding_ti_cc2520
- dtbinding_zephyr_ieee802154_uart_pipe

IEEE 802.15.4 HDLC RCP interface

- dtbinding_nxp_hdlc_rcp_if
- dtbinding_spi_hdlc_rcp_if
- dtbinding_uart_hdlc_rcp_if

IIO (Industrial I/O)

- dtbinding_current_sense_amplifier
- dtbinding_current_sense_shunt
- dtbinding_voltage_divider

Input

- dtbinding_adafruit_seesaw_gamepad
- dtbinding_adc_keys
- dtbinding_adsemi_tsm12
- dtbinding_analog_axis
- dtbinding_arduino_modulino_buttons
- dtbinding_bflb_irx
- dtbinding_chipsemi_chsc5x
- dtbinding_chipsemi_chsc6540
- dtbinding_chipsemi_chsc6x
- dtbinding_cirque_pinnacle_i2c
- dtbinding_cirque_pinnacle_spi
- dtbinding_cypress_cy8cmbr3xxx
- dtbinding_espressif_esp32_touch
- dtbinding_focaltech_ft5336
- dtbinding_focaltech_ft6146
- dtbinding_futaba_sbus
- dtbinding_goodix_gt911
- dtbinding_gpio_kbd_matrix

- dtbinding_gpio_keys
- dtbinding_gpio_qdec
- dtbinding_hynitron_cst8xx
- dtbinding_ilitek_ili2132a
- dtbinding_input_keymap
- dtbinding_ite_it51xxx_kbd
- dtbinding_ite_it8801_kbd
- dtbinding_ite_it8xxx2_kbd
- dtbinding_microchip_cap12xx
- dtbinding_microchip_xec_kbd
- dtbinding_nintendo_nunchuk
- dtbinding_nuvoton_npcx_kbd
- dtbinding_nxp_mcux_kpp
- dtbinding_nxp_tsi_input
- dtbinding_parade_tma525b
- dtbinding_pixart_pat912x
- dtbinding_pixart_paw32xx
- dtbinding_pixart_pmw3610
- dtbinding_realtek_bee_keyscan
- dtbinding_realtek_rts5912_kbd
- dtbinding_renesas_ra_ctsu
- dtbinding_renesas_ra_ctsu_button
- dtbinding_renesas_ra_ctsu_slider
- dtbinding_renesas_ra_ctsu_wheel
- dtbinding_renesas_rx_ctsu
- dtbinding_renesas_rx_ctsu_button
- dtbinding_renesas_rx_ctsu_slider
- dtbinding_renesas_rx_ctsu_wheel
- dtbinding_sitronix_cf1133
- dtbinding_st_stm32_tsc
- dtbinding_st_stmpe811
- dtbinding_tbs_crsf
- dtbinding_tsc_keys
- dtbinding_virtio_input
- dtbinding_vishay_vs1838b
- dtbinding_wch_ch9350l
- dtbinding_xptek_xpt2046
- dtbinding_zephyr_input_double_tap
- dtbinding_zephyr_input_longpress

- dtbinding_zephyr_input_sdl_touch
- dtbinding_zephyr_lvgl_button_input
- dtbinding_zephyr_lvgl_encoder_input
- dtbinding_zephyr_lvgl_keypad_input
- dtbinding_zephyr_lvgl_pointer_input
- dtbinding_zephyr_native_linux_evdev

Interrupt controller

- dtbinding_adi_max32_rv32_intc
- dtbinding_amd_acp_intc
- dtbinding_andestech_nceplic100
- dtbinding_arm_gic
- dtbinding_arm_gic_v1
- dtbinding_arm_gic_v2
- dtbinding_arm_gic_v3
- dtbinding_arm_gic_v3_its
- dtbinding_arm_v6m_nvic
- dtbinding_arm_v7m_nvic
- dtbinding_arm_v8.1m_nvic
- dtbinding_arm_v8m_nvic
- dtbinding_atmel_sam0_eic
- dtbinding_brcm_bcm2835_armctrl_ic
- dtbinding_brcm_bcm2836_l1_intc
- dtbinding_cdac_thejas32_intc
- dtbinding_cdns_swerv_pic
- dtbinding_cdns_xtensa_core_intc
- dtbinding_cypress_psoc6_intmux
- dtbinding_cypress_psoc6_intmux_ch
- dtbinding_espressif_esp32_intc
- dtbinding_gaisler_irqmp
- dtbinding_gd_gd32_exti
- dtbinding_hazard3_hazard3_intc
- dtbinding_infineon_aurix_ir
- dtbinding_infineon_sysintc
- dtbinding_infineon_sysintc_mux
- dtbinding_infineon_xmc4xxx_intc
- dtbinding_intel_ace_intc
- dtbinding_intel_cavs_intc
- dtbinding_intel_ioapic

- dtbinding_intel_loapic
- dtbinding_intel_vt_d
- dtbinding_ite_it51xxx_intc
- dtbinding_ite_it51xxx_wuc
- dtbinding_ite_it51xxx_wuc_map
- dtbinding_ite_it8xxx2_intc
- dtbinding_ite_it8xxx2_intc_v2
- dtbinding_ite_it8xxx2_wuc
- dtbinding_ite_it8xxx2_wuc_map
- dtbinding_litex_vexriscv_intc0
- dtbinding_mediatek_adsp_intc
- dtbinding_microchip_aic_g1_intc
- dtbinding_microchip_dmec_ecia_girq
- dtbinding_microchip_eic_g1_intc
- dtbinding_microchip_smartfusion2_h2f_irqctrl
- dtbinding_microchip_xec_ecia
- dtbinding_microchip_xec_ecia_girq
- dtbinding_mti_cpu_intc
- dtbinding_nordic_nrf_clic
- dtbinding_nuclei_eclic
- dtbinding_nuvoton_npcx_miwu
- dtbinding_nuvoton_npcx_miwu_int_map
- dtbinding_nuvoton_npcx_miwu_wui_map
- dtbinding_nxp_gint
- dtbinding_nxp_irqsteer_intc
- dtbinding_nxp_irqsteer_master
- dtbinding_nxp_pint
- dtbinding_nxp_s32_wkpu
- dtbinding_nxp_siul2_eirq
- dtbinding_nxp_wuu
- dtbinding_opencores_or1k_pic_level
- dtbinding_openisa_rv32m1_event_unit
- dtbinding_openisa_rv32m1_intmux
- dtbinding_openisa_rv32m1_intmux_ch
- dtbinding_qcom_hvm_pic
- dtbinding_renesas_rx_grp_intc
- dtbinding_renesas_rx_icu
- dtbinding_renesas_rx_icum
- dtbinding_renesas_rz_ext_irq

- dtbinding_renesas_rz_icu
- dtbinding_renesas_rz_icu_v2
- dtbinding_renesas_rz_intc
- dtbinding_renesas_rz_intc_v2
- dtbinding_renesas_rz_tint
- dtbinding_riscv_aplic
- dtbinding_riscv_clic
- dtbinding_riscv_cpu_intc
- dtbinding_riscv_imsic
- dtbinding_sensry_sy1xx_event_unit
- dtbinding_shared_irq
- dtbinding_sifive_clic_draft
- dtbinding_sifive_clint0
- dtbinding_sifive_plic_1.0.0
- dtbinding_snps_archs_idu_intc
- dtbinding_snps_arcv2_intc
- dtbinding_snps_designware_intc
- dtbinding_st_stm32_exti
- dtbinding_st_stm32g0_exti
- dtbinding_st_stm32h7rs_exti
- dtbinding_st_stm32wb0_gpio_intc
- dtbinding_swerv_pic
- dtbinding_ti_vim
- dtbinding_wch_exti
- dtbinding_wch_pfic

IPC (Inter-Processor Communication)

- dtbinding_infineon_ipc
- dtbinding_intel_adsp_host_ipc
- dtbinding_intel_adsp_idc
- dtbinding_nordic_nrf_ipc
- dtbinding_nordic_nrf_ipct_global
- dtbinding_nordic_nrf_ipct_local
- dtbinding_nxp_ipc_rpmsg_lite
- dtbinding_zephyr_ipc_icbmsg
- dtbinding_zephyr_ipc_icmsg
- dtbinding_zephyr_ipc_icmsg_me_follower
- dtbinding_zephyr_ipc_icmsg_me_initiator
- dtbinding_zephyr_ipc_openamp_static_vrings

IPM (Inter-Processor Mailbox)

- dtbinding_espressif_esp32_ipm
- dtbinding_intel_adsp_mailbox
- dtbinding_intel_sedi_ipm
- dtbinding_linaro_ivshmem_ipm
- dtbinding_nxp_imx_mu
- dtbinding_nxp_lpc_mailbox
- dtbinding_st_stm32_hsem_mailbox
- dtbinding_st_stm32_ipcc_mailbox
- dtbinding_xlnx_zynqmp_ipi_mailbox
- dtbinding_zephyr_mbox_ipm

LED (Light Emitting Diode)

- dtbinding_arduino_modulino_buttons_leds
- dtbinding_awinic_aw9523b_led
- dtbinding_dac_leds
- dtbinding_gpio_leds
- dtbinding_holtek_ht16k33
- dtbinding_issi_is31fl3193
- dtbinding_issi_is31fl3194
- dtbinding_issi_is31fl3197
- dtbinding_issi_is31fl3216a
- dtbinding_issi_is31fl3733
- dtbinding_leds_group_multicolor
- dtbinding_microchip_xec_bbled
- dtbinding_nordic_npm10xx_led
- dtbinding_nordic_npm1300_led
- dtbinding_nordic_npm1304_led
- dtbinding_nxp_pca9530
- dtbinding_nxp_pca9531
- dtbinding_nxp_pca9532
- dtbinding_nxp_pca9533
- dtbinding_nxp_pca9633
- dtbinding_onnn_ncp5623
- dtbinding_pwm_leds
- dtbinding_sct_sct2024
- dtbinding_ti_lp3943
- dtbinding_ti_lp5009
- dtbinding_ti_lp5012

- dtbinding_ti_lp5018
- dtbinding_ti_lp5024
- dtbinding_ti_lp5030
- dtbinding_ti_lp5036
- dtbinding_ti_lp5562
- dtbinding_ti_lp5569
- dtbinding_ti_lp5860
- dtbinding_ti_lp5861
- dtbinding_ti_lp5862
- dtbinding_ti_lp5864
- dtbinding_ti_lp5866
- dtbinding_ti_lp5868
- dtbinding_ti_tlc59108
- dtbinding_x_powers_axp192_led
- dtbinding_x_powers_axp2101_led
- dtbinding_zephyr_fake_leds
- dtbinding_zephyr_native_linux_leds

LED strip

- dtbinding_apa_apa102
- dtbinding_arduino_modulino_pixels
- dtbinding_greeled_lpd8803
- dtbinding_greeled_lpd8806
- dtbinding_ti_tlc5971
- dtbinding_ti_tlc59731
- dtbinding_worldsemi_ws2812_bflb_wo
- dtbinding_worldsemi_ws2812_gpio
- dtbinding_worldsemi_ws2812_i2s
- dtbinding_worldsemi_ws2812_pulse_io
- dtbinding_worldsemi_ws2812_rpi_pico_pio
- dtbinding_worldsemi_ws2812_spi
- dtbinding_worldsemi_ws2812_uart

LIN (Local Interconnect Network)

- dtbinding_renesas_ra_lin_sci_b

LoRa

- dtbinding_reyax_rylrxxx
- dtbinding_semtech_llcc68
- dtbinding_semtech_lr1121
- dtbinding_semtech_sx1261
- dtbinding_semtech_sx1262
- dtbinding_semtech_sx1268
- dtbinding_semtech_sx1272
- dtbinding_semtech_sx1276
- dtbinding_semtech_sx1278
- dtbinding_st_stm32wl_subghz_radio

Mailbox

- dtbinding_adi_mbox_max32_sema
- dtbinding_andestech_mbox_plic_sw
- dtbinding_arm_mhuv2
- dtbinding_arm_mhuv3
- dtbinding_brcm_bcm2711_mbox
- dtbinding_espressif_mbox_esp32
- dtbinding_infineon_mbox
- dtbinding_linaro_ivshmem_mbox
- dtbinding_nordic_mbox_nrf_ipc
- dtbinding_nordic_nrf_bellboard_rx
- dtbinding_nordic_nrf_bellboard_tx
- dtbinding_nordic_nrf_vevif_event_rx
- dtbinding_nordic_nrf_vevif_event_tx
- dtbinding_nordic_nrf_vevif_task_rx
- dtbinding_nordic_nrf_vevif_task_tx
- dtbinding_nxp_mbox_imx_mu
- dtbinding_nxp_mbox_mailbox
- dtbinding_nxp_s32_mru
- dtbinding_raspberrypi_pico_mbox
- dtbinding_renesas_ra_ipc_mbox
- dtbinding_renesas_rcar_mfis_mbox
- dtbinding_renesas_rz_mhu_mbox
- dtbinding_st_mbox_stm32_hsem
- dtbinding_ti_omap_mailbox
- dtbinding_ti_secure_proxy
- dtbinding_xlnx_mbox_versal_ipi_mailbox

Mailbox Handling Unit

- dtbinding_arm_mhu

MCUmgr

- dtbinding_zephyr_smp_spi

Memory controller

- dtbinding_adi_max32_backup_sram
- dtbinding_adi_max32_hpb
- dtbinding_atmel_sam_smc
- dtbinding_bflb_bl61x_psram
- dtbinding_bflb_bl808_psram_uhs
- dtbinding_bflb_bl808_psram_uhs_controller
- dtbinding_bflb_sf_bank
- dtbinding_bflb_sf_controller
- dtbinding_espressif_esp32_psram
- dtbinding_ite_it8xxx2_bbram
- dtbinding_ite_it8xxx2_ilm
- dtbinding_microchip_xec_bbram
- dtbinding_motorola_mc146818_bbram
- dtbinding_nuvoton_npcx_bbram
- dtbinding_nxp_flexram
- dtbinding_nxp_imx_semc
- dtbinding_nxp_imx_snvs_gpr
- dtbinding_nxp_sramc
- dtbinding_nxp_xspi_psram
- dtbinding_realtek_rts5912_bbram
- dtbinding_renesas_ra_sdram
- dtbinding_renesas_smartbond_nor_psram
- dtbinding_sifive_fu740_c000_ddr
- dtbinding_silabs_siwx91x_qspi_memory
- dtbinding_st_stm32_backup_sram
- dtbinding_st_stm32_bbram
- dtbinding_st_stm32_fmc
- dtbinding_st_stm32_fmc_nor_psram
- dtbinding_st_stm32_fmc_sdram
- dtbinding_st_stm32_ospi_psram
- dtbinding_st_stm32_xspi_psram
- dtbinding_st_stm32h7_fmc
- dtbinding_zephyr_bbram_emul

Memory management

- dtbinding_intel_adsp_imr
- dtbinding_intel_adsp_mtl_tlb
- dtbinding_intel_adsp_tlb

Memory window

- dtbinding_intel_adsp_mem_window

MIPI-DBI (Mobile Industry Processor Interface Display Bus Interface)

- dtbinding_bflb_dbi
- dtbinding_espressif_esp32_lcd_cam_mipi_dbi
- dtbinding_nxp_lcdic
- dtbinding_nxp_mipi_dbi_dcnano_lcdif
- dtbinding_nxp_mipi_dbi_flexio_lcdif
- dtbinding_raspberrypi_pico_mipi_dbi_pio
- dtbinding_renesas_smartbond_mipi_dbi
- dtbinding_sifli_sf32lb_lcdc_mipi_dbi
- dtbinding_st_stm32_fmc_mipi_dbi
- dtbinding_zephyr_mipi_dbi_bitbang
- dtbinding_zephyr_mipi_dbi_spi

MIPI-DSI (Mobile Industry Processor Interface Display Serial Interface)

- dtbinding_espressif_esp_mipi_dsi
- dtbinding_nxp_imx_mipi_dsi
- dtbinding_nxp_mipi_dsi_2l
- dtbinding_nxp_mipi_dsi_dwc
- dtbinding_renesas_ra_mipi_dsi
- dtbinding_renesas_rz_mipi_dsi
- dtbinding_st_stm32_mipi_dsi
- dtbinding_st_stm32u5_mipi_dsi

Miscellaneous

- dtbinding_adi_max2221x_misc
- dtbinding_adi_tmc6460_spi
- dtbinding_adi_tmc6460_uart
- dtbinding_ene_kb106x_gcfg
- dtbinding_ene_kb1200_gcfg
- dtbinding_ene_kb1200_pmu
- dtbinding_espressif_esp32_lcd_cam
- dtbinding_espressif_esp32_ppa

- dtbinding_gd_gd32_syscfg
- dtbinding_intel_adsp_communication_widget
- dtbinding_intel_timeaware_gpio
- dtbinding_nordic_axon
- dtbinding_nordic_ironside_call
- dtbinding_nordic_nrf_bicr
- dtbinding_nordic_nrf_dppic
- dtbinding_nordic_nrf_dppic_global
- dtbinding_nordic_nrf_dppic_local
- dtbinding_nordic_nrf_ficr
- dtbinding_nordic_nrf_mpc
- dtbinding_nordic_nrf_ppi
- dtbinding_nordic_nrf_ppib
- dtbinding_nuvoton_npcx_booter_variant
- dtbinding_nuvoton_npcx_soc_id
- dtbinding_nxp_flexio
- dtbinding_nxp_imx93_mediamix
- dtbinding_nxp_imx93_video_pll
- dtbinding_nxp_mcxw_hw_params
- dtbinding_nxp_rdc
- dtbinding_nxp_rt600_dsp_ctrl
- dtbinding_nxp_rt700_dsp_ctrl_hifi4
- dtbinding_nxp_rtxxx_dsp_ctrl
- dtbinding_nxp_s32_emios
- dtbinding_nxp_s32_lcu
- dtbinding_nxp_s32_trgmux
- dtbinding_raspberrypi_pico_pio
- dtbinding_raspberrypi_pico_pio_device
- dtbinding_raspberrypi_pico_powman
- dtbinding_raspberrypi_pico_sio
- dtbinding_renesas_ra_agt
- dtbinding_renesas_ra_drw
- dtbinding_renesas_ra_elc
- dtbinding_renesas_ra_external_interrupt
- dtbinding_renesas_ra_sau
- dtbinding_renesas_ra_sau_channel
- dtbinding_renesas_ra_sci
- dtbinding_renesas_ra_ulpt
- dtbinding_renesas_rx_dtc

- dtbinding_renesas_rx_external_interrupt
- dtbinding_renesas_rx_mtu
- dtbinding_renesas_rx_sci
- dtbinding_renesas_rx_sci_b
- dtbinding_renesas_rx_system_controller
- dtbinding_renesas_rz_sci
- dtbinding_renesas_rz_sci_b
- dtbinding_seeed_grove_lcd_rgb
- dtbinding_skyworks_sky13317
- dtbinding_skyworks_sky13348
- dtbinding_skyworks_sky13351
- dtbinding_snps_arc_iot_sysconf
- dtbinding_snps_archs_ici
- dtbinding_st_stm32_npu
- dtbinding_st_stm32_npu_cache
- dtbinding_st_stm32n6_ramcfg
- dtbinding_ti_tdp2004
- dtbinding_zephyr_devmux
- dtbinding_zephyr_flash_disk
- dtbinding_zephyr_log_uart
- dtbinding_zephyr_modbus_serial
- dtbinding_zephyr_swdp_gpio

MMU (Memory Management Unit) / MPU (Memory Protection Unit)

- dtbinding_arm_armv6m_mpu
- dtbinding_arm_armv7m_mpu
- dtbinding_arm_armv8.1m_mpu
- dtbinding_arm_armv8m_mpu
- dtbinding_nxp_sysmpu

Modem

- dtbinding_fibocom_le250
- dtbinding_nordic_nrf91_slm
- dtbinding_nordic_nrf91_sm_v2
- dtbinding_nordic_nrf93m1
- dtbinding_openisa_rv32m1_genfsk
- dtbinding_quectel_bc66
- dtbinding_quectel_bc660k
- dtbinding_quectel_bc66x
- dtbinding_quectel_bg770

- dtbinding_quectel_bg95
- dtbinding_quectel_bg96
- dtbinding_quectel_bg9x
- dtbinding_quectel_eg21_g
- dtbinding_quectel_eg25_g
- dtbinding_quectel_eg800q
- dtbinding_quectel_eg915u
- dtbinding_simcom_a76xx
- dtbinding_simcom_sim7080
- dtbinding_sqn_gm02s
- dtbinding_st_st87mxx
- dtbinding_swir_hl7800
- dtbinding_swir_hl7812
- dtbinding_swir_hl7812_gnss
- dtbinding_swir_hl7812_offload
- dtbinding_swir_hl78xx
- dtbinding_swir_hl78xx_gnss
- dtbinding_swir_hl78xx_offload
- dtbinding_telit_le910c1tx
- dtbinding_telit_lex10q1
- dtbinding_telit_me310g1
- dtbinding_telit_me910g1
- dtbinding_trasna_lexi_r10
- dtbinding_u_blox_lara_r6
- dtbinding_u_blox_sara_r4
- dtbinding_u_blox_sara_r5
- dtbinding_wnc_m14a2a

MTD (Memory Technology Device)

- dtbinding_andestech_qspi_nor
- dtbinding_aps_z8_mspi
- dtbinding_aps6404l_mspi
- dtbinding_atmel_at24
- dtbinding_atmel_at24mac402
- dtbinding_atmel_at25
- dtbinding_atmel_at25xv021a
- dtbinding_atmel_at45
- dtbinding_atmel_sam_flash
- dtbinding_atxp032_mspi

- dtbinding_bflb_sf_device
- dtbinding_bflb_sf_flash
- dtbinding_fixed_partitions
- dtbinding_fixed_subpartitions
- dtbinding_fujitsu_mb85rcxx
- dtbinding_fujitsu_mb85rsxx
- dtbinding_gd_gd32_nv_flash_v1
- dtbinding_gd_gd32_nv_flash_v2
- dtbinding_gd_gd32_nv_flash_v3
- dtbinding_infineon_fm25xxx
- dtbinding_infineon_s28hx512t_mspi
- dtbinding_infineon_xmc4xxx_nv_flash
- dtbinding_is25xX0xx_mspi
- dtbinding_is66wv_mspi
- dtbinding_jedec_nor_mspi
- dtbinding_jedec_qspi_nor
- dtbinding_jedec_spi_nand
- dtbinding_jedec_spi_nor
- dtbinding_microchip_flash_g2
- dtbinding_microchip_flash_g3
- dtbinding_microchip_xec_eeprom
- dtbinding_microcrystal_rv3028_eeprom
- dtbinding_mxicy_mx25u_mspi
- dtbinding_mxicy_mx25um51345g
- dtbinding_netsol_s3axx04
- dtbinding_nordic_mram
- dtbinding_nordic_owned_partitions
- dtbinding_nordic_qspi_nor
- dtbinding_nordic_tz_nonsecure
- dtbinding_nordic_tz_secure
- dtbinding_nxp_c40_flash
- dtbinding_nxp_imx_flexspi_aps6404l
- dtbinding_nxp_imx_flexspi_aps6408l
- dtbinding_nxp_imx_flexspi_hyperflash
- dtbinding_nxp_imx_flexspi_is66wvq8m4
- dtbinding_nxp_imx_flexspi_is66wvs8m8
- dtbinding_nxp_imx_flexspi_mx25um51345g
- dtbinding_nxp_imx_flexspi_nand
- dtbinding_nxp_imx_flexspi_nor

- dtbinding_nxp_imx_flexspi_s27ks0641
- dtbinding_nxp_imx_flexspi_w956a8mbya
- dtbinding_nxp_lpc11u6x_eeprom
- dtbinding_nxp_mcxw_ifr
- dtbinding_nxp_s32_qspi_device
- dtbinding_nxp_s32_qspi_hyperflash
- dtbinding_nxp_s32_qspi_nor
- dtbinding_nxp_s32_xspi_device
- dtbinding_nxp_s32_xspi_hyperram
- dtbinding_realtek_bee_nor_flash
- dtbinding_renesas_ra_nv_mram
- dtbinding_renesas_ra_qspi_nor
- dtbinding_renesas_rx_nv_flash
- dtbinding_sifli_sf32lb_mpi_qspi_nor
- dtbinding_soc_nv_flash
- dtbinding_st_m95p32_mspi
- dtbinding_st_stm32_eeprom
- dtbinding_st_stm32_nv_flash
- dtbinding_st_stm32f4_nv_flash
- dtbinding_st_stm32l0_nv_flash
- dtbinding_ti_cc23x0_ccfg_flash
- dtbinding_ti_tmp11x_eeprom
- dtbinding_winbond_w25q512nw
- dtbinding_zephyr_emu_eeprom
- dtbinding_zephyr_emul_flash_mspi
- dtbinding_zephyr_fake_eeprom
- dtbinding_zephyr_i2c_target_eeprom
- dtbinding_zephyr_mapped_partition
- dtbinding_zephyr_sim_eeprom

Multi-bit SPI (Serial Peripheral Interface)

- dtbinding_ambiq_mspi_controller
- dtbinding_ambiq_mspi_device
- dtbinding_cdns_xspi_nor
- dtbinding_microchip_xec_qmspi_controller
- dtbinding_microchip_xec_qmspi_device_mspi
- dtbinding_nordic_nrf_exmif
- dtbinding_nordic_nrf_mspi
- dtbinding_nxp_qspi

- dtbinding_snps_designware_ssi
- dtbinding_st_nor_mspi
- dtbinding_st_psram_device_mspi
- dtbinding_st_stm32_ospi_controller
- dtbinding_st_stm32_qspi_controller
- dtbinding_st_stm32_xspi_controller
- dtbinding_zephyr_emul_device_mspi
- dtbinding_zephyr_mspi_emul_controller
- dtbinding_zephyr_peripheral_device_mspi

Multi-Function Device

- dtbinding_adi_ad559x_spi
- dtbinding_adi_ad559x_i2c
- dtbinding_adi_adp5360
- dtbinding_adi_adp5585
- dtbinding_adi_max22017
- dtbinding_adi_max2221x
- dtbinding_adi_maxq10xx
- dtbinding_ambiq_iom
- dtbinding_ambiq_ios
- dtbinding_awinic_aw9523b
- dtbinding_gd_gd32_rcu
- dtbinding_infineon_autanalog
- dtbinding_infineon_autanalog_ac_state
- dtbinding_infineon_autanalog_ctb
- dtbinding_infineon_autanalog_prb
- dtbinding_infineon_autanalog_ptcomp
- dtbinding_infineon_hppass_ac_state
- dtbinding_infineon_hppass_analog
- dtbinding_infineon_hppass_csg
- dtbinding_infineon_mxcrypto
- dtbinding_infineon_mxcryptolite
- dtbinding_infineon_tle9104
- dtbinding_ite_it8801_altctrl
- dtbinding_ite_it8801_mfd
- dtbinding_ite_it8801_mfd_map
- dtbinding_m5stack_m5pm1
- dtbinding_maxim_ds3231_mfd
- dtbinding_maxim_max20335

- dtbinding_maxim_max31790
- dtbinding_microchip_sam_flexcom
- dtbinding_microcrystal_rv3028
- dtbinding_microcrystal_rv3032_mfd
- dtbinding_motorola_mc146818_mfd
- dtbinding_nordic_npm10xx
- dtbinding_nordic_npm1300
- dtbinding_nordic_npm1304
- dtbinding_nordic_npm2100
- dtbinding_nordic_npm6001
- dtbinding_nuvoton_nct38xx
- dtbinding_nxp_lp_flexcomm
- dtbinding_nxp_pca9422
- dtbinding_nxp_pf1550
- dtbinding_nxp_sc18im704
- dtbinding_nxp_sc18is606
- dtbinding_rohm_bd8lb600fs
- dtbinding_sifli_sf32lb_rcc
- dtbinding_silabs_emu
- dtbinding_ti_tla2528
- dtbinding_x_powers_axp192
- dtbinding_x_powers_axp2101

MUX (Multiplexer)

- dtbinding_adi_adgm3121
- dtbinding_adi_adgm3121_gpio
- dtbinding_gpio_mux
- dtbinding_nxp_inputmux
- dtbinding_nxp_trgmux

Networking

- dtbinding_nordic_nrf_nfct
- dtbinding_nordic_nrf_nfct_v2
- dtbinding_nordic_nrf_radio
- dtbinding_nordic_nrf21540_fem
- dtbinding_nordic_nrf21540_fem_spi
- dtbinding_radio_fem_two_ctrl_pins
- dtbinding_radio_gpio_coex
- dtbinding_silabs_series2_radio
- dtbinding_silabs_siwx91x_nwp

- dtbinding_st_stm32wba_radio
- dtbinding_ti_cc13xx_cc26xx_radio

NVMEM (Non-Volatile Memory)

- dtbinding_fixed_layout

OCTOSPI (Octal Serial Peripheral Interface)

- dtbinding_renesas_ra_ospi_b
- dtbinding_st_stm32_ospi

OPAMP (Operational Amplifier)

- dtbinding_infineon_autanalog_ctb_opamp
- dtbinding_nxp_opamp
- dtbinding_nxp_opamp_fast
- dtbinding_st_stm32_opamp
- dtbinding_st_stm32g4_opamp

Options

- dtbinding_openthread_config

OTP (One-Time Programmable) memory

- dtbinding_adi_axi_sysid
- dtbinding_bflb_efuse
- dtbinding_nordic_npm10xx_uicr
- dtbinding_nxp_ocotp
- dtbinding_nxp_otpc
- dtbinding_nxp_rt7xx_ocotp
- dtbinding_realtek_rts5817_ocotp
- dtbinding_sifli_sf32lb_efuse
- dtbinding_st_stm32_bsec
- dtbinding_st_stm32_nvm_otp
- dtbinding_zephyr_otp_emul

PCIE (Peripheral Component Interconnect Express)

- dtbinding_brcm_brcmstb_pcie
- dtbinding_brcm_iproc_pcie_ep
- dtbinding_brcm_iproc_pcie_ep_v2
- dtbinding_pci_host_ecam_generic
- dtbinding_pcie_controller

PECI (Platform Environment Control Interface)

- dtbinding_ite_it8xxx2_peci
- dtbinding_microchip_xec_peci
- dtbinding_nuvoton_npcx_peci

Performance state

- dtbinding_nxp_imx9_pstate
- dtbinding_nxp_mcxn_pstate
- dtbinding_zephyr_generic_pstate
- dtbinding_zephyr_native_sim_pstate
- dtbinding_zephyr_pstate

PHY

- dtbinding_can_transceiver_gpio
- dtbinding_lin_transceiver_gpio
- dtbinding_renesas_ra_usbphyc
- dtbinding_st_stm32_usbphyc
- dtbinding_st_stm32f7_usbphyc
- dtbinding_st_stm32n6_usbphyc
- dtbinding_st_stm32u5_otghs_phy
- dtbinding_usb_nop_xceiv
- dtbinding_usb_ulpi_phy

Pin control

- dtbinding_adi_max32_pinctrl
- dtbinding_aesc_pinctrl
- dtbinding_alif_pinctrl
- dtbinding_ambiq_apollo2_pinctrl
- dtbinding_ambiq_apollo3_pinctrl
- dtbinding_ambiq_apollo4_pinctrl
- dtbinding_ambiq_apollo5_pinctrl
- dtbinding_arm_mps2_pinctrl
- dtbinding_arm_mps3_pinctrl
- dtbinding_arm_mps4_pinctrl
- dtbinding_arm_v2m_beetle_pinctrl
- dtbinding_arm_v2m_musca_b1_pinctrl
- dtbinding_atmel_sam_pinctrl
- dtbinding_atmel_sam0_pinctrl
- dtbinding_atmel_sam0_pinmux
- dtbinding_bflb_pinctrl

- dtbinding_brcm_bcm2711_pinctrl
- dtbinding_brcm_bcm2712_pinctrl
- dtbinding_elan_em32_pinctrl
- dtbinding_ene_kb106x_pinctrl
- dtbinding_ene_kb1200_pinctrl
- dtbinding_espressif_esp32_pinctrl
- dtbinding_gd_gd32_afio
- dtbinding_gd_gd32_pinctrl_af
- dtbinding_gd_gd32_pinctrl_afio
- dtbinding_infineon_pinctrl
- dtbinding_infineon_xmc4xxx_pinctrl
- dtbinding_ite_it8xxx2_pinctrl
- dtbinding_ite_it8xxx2_pinctrl_func
- dtbinding_mediatek_mt8188_pinctrl
- dtbinding_microchip_port_g1_pinctrl
- dtbinding_microchip_sama7g5_pinctrl
- dtbinding_microchip_xec_pinctrl
- dtbinding_nordic_nrf_pinctrl
- dtbinding_nuvoton_npcx_leakage_io
- dtbinding_nuvoton_npcx_lvolctrl_conf
- dtbinding_nuvoton_npcx_pinctrl
- dtbinding_nuvoton_npcx_pinctrl_conf
- dtbinding_nuvoton_npcx_pinctrl_def
- dtbinding_nuvoton_npcx_pinctrl_npckn
- dtbinding_nuvoton_npcx_scfg
- dtbinding_nuvoton_numaker_pinctrl
- dtbinding_nuvoton_numicro_pinctrl
- dtbinding_nxp_imx_blkctrl_ns_aon
- dtbinding_nxp_imx_blkctrl_wakeup
- dtbinding_nxp_imx_gpr
- dtbinding_nxp_imx_iomuxc
- dtbinding_nxp_imx_iomuxc_scu
- dtbinding_nxp_imx7d_pinctrl
- dtbinding_nxp_imx8_pinctrl
- dtbinding_nxp_imx8m_pinctrl
- dtbinding_nxp_imx8mp_pinctrl
- dtbinding_nxp_imx8ulp_pinctrl
- dtbinding_nxp_imx93_pinctrl
- dtbinding_nxp_lpc_iocon

- dtbinding_nxp_lpc_iocon_pinctrl
- dtbinding_nxp_lpc_iocon_pio
- dtbinding_nxp_lpc11u6x_pinctrl
- dtbinding_nxp_lpc84x_iocon
- dtbinding_nxp_lpc84x_swm
- dtbinding_nxp_mci_io_mux
- dtbinding_nxp_mcux_rt_pinctrl
- dtbinding_nxp_mcux_rt11xx_pinctrl
- dtbinding_nxp_mcux_rt266x_pinctrl
- dtbinding_nxp_mcxe31x_siul2_pinctrl
- dtbinding_nxp_port_pinctrl
- dtbinding_nxp_port_pinmux
- dtbinding_nxp_rt_iocon_pinctrl
- dtbinding_nxp_s32k3_siul2_pinctrl
- dtbinding_nxp_s32k5_pinctrl
- dtbinding_nxp_s32ze_siul2_pinctrl
- dtbinding_openisa_rv32m1_pinctrl
- dtbinding_openisa_rv32m1_pinmux
- dtbinding_quicklogic_eos_s3_pinctrl
- dtbinding_raspberrypi_pico_pinctrl
- dtbinding_raspberrypi_rp1_pinctrl
- dtbinding_realtek_ameba_pinctrl
- dtbinding_realtek_bee_pinctrl
- dtbinding_realtek_rts5817_pinctrl
- dtbinding_realtek_rts5912_pinctrl
- dtbinding_renesas_ra_pinctrl_pfs
- dtbinding_renesas_ra0_pinctrl_pfs
- dtbinding_renesas_rcar_pfc
- dtbinding_renesas_rcar_pfc_x5h
- dtbinding_renesas_rx_pinctrl
- dtbinding_renesas_rx_pinctrl_pfs
- dtbinding_renesas_rx_pinmux
- dtbinding_renesas_rza_pinctrl
- dtbinding_renesas_rza2m_pinctrl
- dtbinding_renesas_rzg_pinctrl
- dtbinding_renesas_rzn_pinctrl
- dtbinding_renesas_rzt_pinctrl
- dtbinding_renesas_rzt2m_pinctrl
- dtbinding_renesas_rzv_pinctrl

- dtbinding_renesas_smartbond_pinctrl
- dtbinding_sensry_sy1xx_pinctrl
- dtbinding_sifive_pinctrl
- dtbinding_sifli_sf32lb52x_pinmux
- dtbinding_silabs_dbus_pinctrl
- dtbinding_silabs_gecko_pinctrl
- dtbinding_silabs_si32_pinctrl
- dtbinding_silabs_siwx91x_pinctrl
- dtbinding_snps_emsdp_pinctrl
- dtbinding_st_stm32_pinctrl
- dtbinding_st_stm32f1_pinctrl
- dtbinding_st_stm32h5_pinctrl
- dtbinding_st_stm32n6_pinctrl
- dtbinding_syna_sr100_pinctrl
- dtbinding_telink_b91_pinctrl
- dtbinding_ti_cc13xx_cc26xx_pinctrl
- dtbinding_ti_cc23x0_pinctrl
- dtbinding_ti_cc32xx_pinctrl
- dtbinding_ti_k3_pinctrl
- dtbinding_ti_mspm0_pinctrl
- dtbinding_ti_tiva_c_pinctrl
- dtbinding_wch_00x_afio
- dtbinding_wch_20x_30x_afio
- dtbinding_wch_afio
- dtbinding_wch_ch570_pinctrl
- dtbinding_wch_h41x_afio
- dtbinding_xlnx_pinctrl_zynq
- dtbinding_xlnx_pinctrl_zynqmp

PMCI (Platform Management Components Intercommunication)

- dtbinding_zephyr_mctp_i2c_gpio_controller
- dtbinding_zephyr_mctp_i2c_gpio_target
- dtbinding_zephyr_mctp_i3c_controller
- dtbinding_zephyr_mctp_i3c_endpoint
- dtbinding_zephyr_mctp_i3c_target

Power domain

- dtbinding_arm_scmi_power_domain
- dtbinding_intel_adsp_power_domain
- dtbinding_nordic_nrfs_gdpwr
- dtbinding_nordic_nrfs_swext
- dtbinding_nxp_scu_pd
- dtbinding_power_domain
- dtbinding_power_domain_gpio
- dtbinding_power_domain_gpio_monitor
- dtbinding_power_domain_soc_state_change
- dtbinding_raspberrypi_bcm283x_power
- dtbinding_silabs_siwx91x_power_domain
- dtbinding_ti_sci_pm_domain

Power management

- dtbinding_atmel_sam_supc
- dtbinding_bflb_power_controller
- dtbinding_infineon_cat1b_power
- dtbinding_infineon_hibernate_wakeup
- dtbinding_ite_it8xxx2_power_elpm
- dtbinding_lowrisc_opentitan_pwrmgr
- dtbinding_microchip_supc_g1
- dtbinding_nordic_nrf_power
- dtbinding_nordic_nrf_usbreg
- dtbinding_nordic_nrf_vmc
- dtbinding_nuvoton_npcx_power_psl
- dtbinding_nuvoton_npcx_power_state
- dtbinding_nxp_aon_wakeup_pin
- dtbinding_nxp_cmc
- dtbinding_nxp_pdcfg_power
- dtbinding_nxp_rw_pmu
- dtbinding_nxp_s32_mc_me
- dtbinding_nxp_s32_mc_rgm
- dtbinding_nxp_s32k3_pmc
- dtbinding_nxp_smc
- dtbinding_nxp_spc
- dtbinding_nxp_vbat
- dtbinding_realtek_rts5912_ulpm
- dtbinding_renesas_ra_battery_backup

- dtbinding_sifli_sf32lb_aon
- dtbinding_sifli_sf32lb52x_pmuc
- dtbinding_st_stm32_dualreg_pwr
- dtbinding_st_stm32_iocell
- dtbinding_st_stm32_pwr
- dtbinding_st_stm32_pwr_wkupctrl
- dtbinding_st_stm32f1_pwr_wkupctrl
- dtbinding_st_stm32f7_pwr_wkupctrl
- dtbinding_st_stm32h5_iocell
- dtbinding_st_stm32h7_pwr
- dtbinding_st_stm32h7rs_pwr
- dtbinding_st_stm32u5_pwr
- dtbinding_st_stm32wb0_pwr
- dtbinding_st_stm32wba_pwr
- dtbinding_telink_b91_power
- dtbinding_zephyr_power_state

Power management CPU operations

- dtbinding_arm_fvp_pwrc
- dtbinding_arm_psci_0.2
- dtbinding_arm_psci_1.1

PPC architecture

- dtbinding_nuvoton_numaker_ppc
- dtbinding_nxp_nx20p3483

PS/2 (Personal System/2)

- dtbinding_ite_it51xxx_ps2
- dtbinding_microchip_xec_ps2
- dtbinding_nuvoton_npcx_ps2_channel
- dtbinding_nuvoton_npcx_ps2_ctrl

PSI5 (Peripheral Sensor Interface, 5th generation)

- dtbinding_nxp_s32_psi5

Pulse IO

- dtbinding_espressif_esp32_rmt
- dtbinding_zephyr_pulse_io_loopback

PWM (Pulse Width Modulation)

- dtbinding_adi_axi_pwmgen
- dtbinding_adi_max2221x_pwm
- dtbinding_adi_max32_pwm
- dtbinding_ambiq_ctimer_pwm
- dtbinding_ambiq_timer_pwm
- dtbinding_arduino_header_pwm
- dtbinding_atmel_sam_pwm
- dtbinding_atmel_sam0_tc_pwm
- dtbinding_atmel_sam0_tcc_pwm
- dtbinding_bflb_pwm_1
- dtbinding_bflb_pwm_2
- dtbinding_elan_em32_pwm
- dtbinding_ene_kb106x_pwm
- dtbinding_ene_kb1200_pwm
- dtbinding_espressif_esp32_ledc
- dtbinding_espressif_esp32_mcpwm
- dtbinding_fsl_imx27_pwm
- dtbinding_gd_gd32_pwm
- dtbinding_infineon_tcpwm_pwm
- dtbinding_infineon_xmc4xxx_ccu4_pwm
- dtbinding_infineon_xmc4xxx_ccu8_pwm
- dtbinding_intel_blinky_pwm
- dtbinding_ite_it51xxx_pwm
- dtbinding_ite_it8801_pwm
- dtbinding_ite_it8xxx2_pwm
- dtbinding_ite_it8xxx2_pwmprs
- dtbinding_litex_pwm
- dtbinding_maxim_max31790_pwm
- dtbinding_microchip_pwm_g1
- dtbinding_microchip_tc_g1_pwm
- dtbinding_microchip_tcc_g1_pwm
- dtbinding_microchip_xec_pwm
- dtbinding_microchip_xec_pwmbbled
- dtbinding_neorv32_pwm
- dtbinding_nordic_nrf_pwm
- dtbinding_nordic_nrf_sw_pwm
- dtbinding_nuvoton_npcx_pwm
- dtbinding_nuvoton_numaker_pwm

- dtbinding_nxp_ctimer_pwm
- dtbinding_nxp_emios
- dtbinding_nxp_emios_pwm
- dtbinding_nxp_flexio_pwm
- dtbinding_nxp_flexpwm
- dtbinding_nxp_ftm_pwm
- dtbinding_nxp_imx_pwm
- dtbinding_nxp_ipwm
- dtbinding_nxp_kinetis_pwt
- dtbinding_nxp_kinetis_tpm
- dtbinding_nxp_pca9685_pwm
- dtbinding_nxp_qtmr_pwm
- dtbinding_nxp_s32_emios_pwm
- dtbinding_nxp_sctimer_pwm
- dtbinding_openisa_rv32m1_tpm
- dtbinding_raspberrypi_pico_pwm
- dtbinding_realtek_ameba_pwm
- dtbinding_realtek_bee_pwm
- dtbinding_realtek_rts5912_pwm
- dtbinding_renesas_pwm_rcar
- dtbinding_renesas_ra_pwm
- dtbinding_renesas_rx_mtu_pwm
- dtbinding_renesas_rz_gpt_pwm
- dtbinding_renesas_rz_mtu_pwm
- dtbinding_renesas_rza2m_gpt_pwm
- dtbinding_sifive_pwm0
- dtbinding_sifli_sf32lb_atim_pwm
- dtbinding_sifli_sf32lb_gpt_pwm
- dtbinding_silabs_gecko_pwm
- dtbinding_silabs_letimer_pwm
- dtbinding_silabs_siwx91x_pwm
- dtbinding_silabs_timer_pwm
- dtbinding_st_stm32_pwm
- dtbinding_telink_b91_pwm
- dtbinding_ti_am3352_ecap
- dtbinding_ti_am3352_ehrpwm
- dtbinding_ti_cc13xx_cc26xx_timer_pwm
- dtbinding_ti_cc23x0_lgpt_pwm
- dtbinding_ti_mspm0_timer_pwm

- dtbinding_wch_adtm_pwm
- dtbinding_wch_gptm_pwm
- dtbinding_wisblock_pwm
- dtbinding_xlnx_xps_timer_1.00.a_pwm
- dtbinding_zephyr_fake_pwm
- dtbinding_zephyr_pwm_bitbang

QSPI (Quad Serial Peripheral Interface)

- dtbinding_adi_max32_spixf
- dtbinding_altr_nios2_qspi
- dtbinding_nxp_s32_qspi
- dtbinding_nxp_s32_qspi_sfp_frad
- dtbinding_nxp_s32_qspi_sfp_mdad
- dtbinding_renesas_ra_qspi
- dtbinding_renesas_rz_spibsc
- dtbinding_renesas_rz_xspi
- dtbinding_st_stm32_qspi

Regulator

- dtbinding_adi_adp5360_regulator
- dtbinding_adi_max20362_regulator
- dtbinding_arduino_modulino_latch_relay
- dtbinding_bflb_aon_regulator
- dtbinding_bflb_rt_regulator
- dtbinding_bflb_soc_regulator
- dtbinding_cirrus_cp9314
- dtbinding_espressif_esp32_regulator
- dtbinding_gd_gd32_bldo
- dtbinding_infineon_autanalog_prb_vref
- dtbinding_m5stack_m5pm1_regulator
- dtbinding_maxim_max20335_regulator
- dtbinding_mps_mpm54304
- dtbinding_nordic_npm10xx_regulator
- dtbinding_nordic_npm1100
- dtbinding_nordic_npm1300_regulator
- dtbinding_nordic_npm1304_regulator
- dtbinding_nordic_npm2100_regulator
- dtbinding_nordic_npm6001_regulator
- dtbinding_nordic_nrf52x_regulator_hv
- dtbinding_nordic_nrf53x_regulator_hv

- dtbinding_nordic_nrf53x_regulators
- dtbinding_nordic_nrf54l_regulators
- dtbinding_nordic_nrf5x_regulator
- dtbinding_nordic_nrf91x_regulators
- dtbinding_nordic_vregusb_regulator
- dtbinding_nxp_pca9420
- dtbinding_nxp_pca9422_regulator
- dtbinding_nxp_pf1550_regulator
- dtbinding_nxp_spc_bandgap_buffer
- dtbinding_nxp_vref
- dtbinding_nxp_vrefv1
- dtbinding_raspberrypi_core_supply_regulator
- dtbinding_realtek_rts5817_regulator
- dtbinding_regulator_fixed
- dtbinding_regulator_gpio
- dtbinding_renesas_smartbond_regulator
- dtbinding_sifli_sf32lb52x_ldo
- dtbinding_silabs_series2_dcdc
- dtbinding_st_stm32_vrefbuf
- dtbinding_ti_mspm0_vref
- dtbinding_ti_tps55287
- dtbinding_ti_tps6287x
- dtbinding_x_powers_axp192_regulator
- dtbinding_x_powers_axp2101_regulator
- dtbinding_zephyr_fake_regulator

Renesas RX

- dtbinding_renesas_rx_swint

Reserved memory

- dtbinding_nordic_owned_memory
- dtbinding_renesas_ofs_memory

Reset controller

- dtbinding_aspeed_ast10x0_reset
- dtbinding_focaltech_ft9001_cpm_rctl
- dtbinding_gd_gd32_rctl
- dtbinding_intel_socfpga_reset
- dtbinding_microchip_mpfs_reset
- dtbinding_microchip_rstc_g1_reset

- dtbinding_nuvoton_npcx_rst
- dtbinding_nuvoton_numaker_rst
- dtbinding_nxp_lpc_syscon_reset
- dtbinding_nxp_mrcc_reset
- dtbinding_nxp_rstctl
- dtbinding_raspberrypi_pico_reset
- dtbinding_realtek_rts5817_reset
- dtbinding_reset_mmio
- dtbinding_rockchip_rk3588_cru_rst
- dtbinding_sifli_sf32lb_rcc_rctl
- dtbinding_st_stm32_rcc_rctl
- dtbinding_syna_sr100_reset
- dtbinding_wch_ch32_rcc_rctl

Retained memory

- dtbinding_gd_gd32_backup_sram
- dtbinding_microcrystal_rv3028_ram
- dtbinding_nordic_nrf_gpregret
- dtbinding_sifli_sf32lb_rtc_backup
- dtbinding_silabs_buram
- dtbinding_zephyr_retained_ram
- dtbinding_zephyr_retained_reg

Retention

- dtbinding_zephyr_retention

RISC-V architecture

- dtbinding_litex_soc_controller
- dtbinding_nordic_nrf_vpr_coprocessor
- dtbinding_openisa_rv32m1_pcc

RNG (Random Number Generator)

- dtbinding_adi_max32_trng
- dtbinding_adi_maxq10xx_trng
- dtbinding_ambiq_puf_trng
- dtbinding_atmel_sam_trng
- dtbinding_brcm_bcm2835_rng
- dtbinding_brcm_iproc_rng200
- dtbinding_espressif_esp32_trng
- dtbinding_gd_gd32_trng

- dtbinding_litex_prbs
- dtbinding_microchip_trng_g1_entropy
- dtbinding_microchip_trng_g2_entropy
- dtbinding_neorv32_trng
- dtbinding_nordic_nrf_cracen_ctrdrbg
- dtbinding_nordic_nrf_rng
- dtbinding_nuvoton_npcx_drbg
- dtbinding_nxp_ele_trng
- dtbinding_nxp_els_trng
- dtbinding_nxp_imx_caam
- dtbinding_nxp_kinetis_rnga
- dtbinding_nxp_kinetis_trng
- dtbinding_nxp_lpc_rng
- dtbinding_openisa_rv32m1_trng
- dtbinding_raspberrypi_pico_rng
- dtbinding_realtek_ameba_trng
- dtbinding_realtek_bee_trng
- dtbinding_renesas_ra_rsip_e50d_trng
- dtbinding_renesas_ra_rsip_e51a_trng
- dtbinding_renesas_ra_sce5_rng
- dtbinding_renesas_ra_sce7_rng
- dtbinding_renesas_ra_sce9_rng
- dtbinding_renesas_ra_trng
- dtbinding_renesas_smartbond_trng
- dtbinding_sensry_sy1xx_trng
- dtbinding_sifli_sf32lb_trng
- dtbinding_silabs_gecko_trng
- dtbinding_silabs_siwx91x_rng
- dtbinding_st_stm32_rng
- dtbinding_st_stm32_rng_noirq
- dtbinding_telink_b91_trng
- dtbinding_ti_cc13xx_cc26xx_trng
- dtbinding_ti_mspm0_trng
- dtbinding_virtio_device4
- dtbinding_wch_rng
- dtbinding_zephyr_native_sim_rng
- dtbinding_zephyr_psa_crypto_rng

RTC (Real Time Clock)

- dtbinding_adi_max31331
- dtbinding_ambiq_am1805
- dtbinding_ambiq_rtc
- dtbinding_atmel_sam_rtc
- dtbinding_atmel_sam0_rtc
- dtbinding_epson_rx8130ce_rtc
- dtbinding_infineon_rtc
- dtbinding_infineon_xmc4xxx_rtc
- dtbinding_ite_it8xxx2_rtc
- dtbinding_maxim_ds1302
- dtbinding_maxim_ds1307
- dtbinding_maxim_ds1337
- dtbinding_maxim_ds3231_rtc
- dtbinding_microchip_mcp7940n
- dtbinding_microchip_rtc_g1
- dtbinding_microchip_rtc_g2
- dtbinding_microchip_rtc_mss
- dtbinding_microchip_xec_hibtimer
- dtbinding_microchip_xec_rtc
- dtbinding_microchip_xec_timer
- dtbinding_microcrystal_rv_8263_c8
- dtbinding_microcrystal_rv3028_rtc
- dtbinding_microcrystal_rv3032
- dtbinding_microcrystal_rv8803
- dtbinding_motorola_mc146818
- dtbinding_nordic_nrf_rtc
- dtbinding_nuvoton_numaker_rtc
- dtbinding_nxp_irtc
- dtbinding_nxp_lpc_rtc
- dtbinding_nxp_pcf2123
- dtbinding_nxp_pcf85063a
- dtbinding_nxp_pcf8523
- dtbinding_nxp_pcf8563
- dtbinding_nxp_rtc
- dtbinding_nxp_rtc_analog
- dtbinding_nxp_rtc_jdp
- dtbinding_raspberrypi_pico_rtc
- dtbinding_realtek_ameba_rtc

- dtbinding_realtek_rts5912_rtc
- dtbinding_renesas_ra_rtc
- dtbinding_renesas_smartbond_rtc
- dtbinding_sifli_sf32lb_rtc
- dtbinding_silabs_gecko_rtcc
- dtbinding_silabs_gecko_stimer
- dtbinding_silabs_rtcc
- dtbinding_silabs_siwx91x_rtc
- dtbinding_silabs_sysrtc
- dtbinding_st_stm32_rtc
- dtbinding_ti_bq32002
- dtbinding_ti_cc13xx_cc26xx_rtc_timer
- dtbinding_ti_mspm0_rtc
- dtbinding_xlnx_xps_timer_1.00.a
- dtbinding_zephyr_fake_rtc
- dtbinding_zephyr_rtc_counter
- dtbinding_zephyr_rtc_emul

SD (Secure Digital)

- dtbinding_zephyr_mmc_disk
- dtbinding_zephyr_sdmmc_disk

SDHC (Secure Digital High Capacity)

- dtbinding_adi_max32_sdhc
- dtbinding_ambiq_sdio
- dtbinding_atmel_sam_hsmci
- dtbinding_bflb_sdhc
- dtbinding_cdns_sdhc
- dtbinding_espressif_esp32_sdhc
- dtbinding_espressif_esp32_sdhc_slot
- dtbinding_infineon_sdhc_sdio
- dtbinding_intel_emmc_host
- dtbinding_litex_mmc
- dtbinding_microchip_sama7g5_sdmmc
- dtbinding_microchip_sdhc_g1
- dtbinding_nuvoton_numaker_sdhc
- dtbinding_nxp_imx_usdhc
- dtbinding_nxp_lpc_sdif
- dtbinding_realtek_ameba_sdhost
- dtbinding_renesas_ra_sdhc

- dtbinding_renesas_rcar_mmc
- dtbinding_st_stm32_sdmmc
- dtbinding_ti_am654_sdhci
- dtbinding_xlnx_versal_8.9a
- dtbinding_zephyr_sdhc_spi_slot

Sensors

- dtbinding_adi_ad2s1210
- dtbinding_adi_ade7978
- dtbinding_adi_adis1647x
- dtbinding_adi_adltc2990
- dtbinding_adi_adt7310
- dtbinding_adi_adt7410
- dtbinding_adi_adt7420
- dtbinding_adi_adt7422
- dtbinding_adi_adxl313
- dtbinding_adi_adxl345_spi
- dtbinding_adi_adxl345_i2c
- dtbinding_adi_adxl355_i2c
- dtbinding_adi_adxl355_spi
- dtbinding_adi_adxl362
- dtbinding_adi_adxl366_spi
- dtbinding_adi_adxl366_i2c
- dtbinding_adi_adxl367_i2c
- dtbinding_adi_adxl367_spi
- dtbinding_adi_adxl372_i2c
- dtbinding_adi_adxl372_spi
- dtbinding_adi_ltc4286
- dtbinding_adi_max30009_spi
- dtbinding_adi_max30009_i2c
- dtbinding_adi_max30210
- dtbinding_allegro_als31300
- dtbinding_amd_sb_tsi
- dtbinding_ams_as5048
- dtbinding_ams_as5600
- dtbinding_ams_as6212
- dtbinding_ams_as6221
- dtbinding_ams_ccs811
- dtbinding_ams_ens210

- dtbinding_ams_iaqcore
- dtbinding_ams_tcs3400
- dtbinding_ams_tcs34725
- dtbinding_ams_tmd2620
- dtbinding_ams_tsl2540
- dtbinding_ams_tsl2561
- dtbinding_ams_tsl2591
- dtbinding_aosong_ags10
- dtbinding_aosong_aht20
- dtbinding_aosong_am2301b
- dtbinding_aosong_dht
- dtbinding_aosong_dht20
- dtbinding_ap_fcx_mldx5
- dtbinding_asahi_kasei_ak8975
- dtbinding_asahi_kasei_akm09918c
- dtbinding_atmel_sam_tc_qdec
- dtbinding_avago_apds9253
- dtbinding_avago_apds9306
- dtbinding_avago_apds9960
- dtbinding_avia_hx711_spi
- dtbinding_bflb_tsen
- dtbinding_bosch_bma280
- dtbinding_bosch_bma4xx_spi
- dtbinding_bosch_bma4xx_i2c
- dtbinding_bosch_bmc150_magn
- dtbinding_bosch_bme280_spi
- dtbinding_bosch_bme280_i2c
- dtbinding_bosch_bme680_i2c
- dtbinding_bosch_bme680_spi
- dtbinding_bosch_bmg160
- dtbinding_bosch_bmi08x_accel_spi
- dtbinding_bosch_bmi08x_accel_i2c
- dtbinding_bosch_bmi08x_gyro_i2c
- dtbinding_bosch_bmi08x_gyro_spi
- dtbinding_bosch_bmi160_spi
- dtbinding_bosch_bmi160_i2c
- dtbinding_bosch_bmi270
- dtbinding_bosch_bmi270_i2c
- dtbinding_bosch_bmi270_spi

- dtbinding_bosch_bmi323_i2c
- dtbinding_bosch_bmi323_spi
- dtbinding_bosch_bmm150_spi
- dtbinding_bosch_bmm150_i2c
- dtbinding_bosch_bmm350_i3c
- dtbinding_bosch_bmm350_i2c
- dtbinding_bosch_bmp180
- dtbinding_bosch_bmp388_i2c
- dtbinding_bosch_bmp388_spi
- dtbinding_bosch_bmp390_spi
- dtbinding_bosch_bmp390_i2c
- dtbinding_bosch_bmp581_i2c
- dtbinding_bosch_bmp581_spi
- dtbinding_bosch_bmp581_i3c
- dtbinding_brcm_afbr_s50
- dtbinding_cfsensor_xgzp68xxd
- dtbinding_dfrobot_a01nyub
- dtbinding_epcos_b57861s0103a039
- dtbinding_espressif_esp32_pcnt
- dtbinding_espressif_esp32_temp
- dtbinding_everlight_als_pt19
- dtbinding_festo_veaa_x_3
- dtbinding_fintek_f75303
- dtbinding_gss_explorir_m
- dtbinding_hamamatsu_s11059
- dtbinding_hamamatsu_s9706
- dtbinding_hc_sr04
- dtbinding_honeywell_hmc5883l
- dtbinding_honeywell_mpr
- dtbinding_honeywell_sm351lt
- dtbinding_hoperf_hp206c
- dtbinding_hoperf_th02
- dtbinding_hzgrow_r502a
- dtbinding_iclegend_s3km1110
- dtbinding_infineon_dps310
- dtbinding_infineon_tle9104_diagnostics
- dtbinding_infineon_xmc4xxx_temp
- dtbinding_invensense_icm40627_i2c
- dtbinding_invensense_icm42370p_i2c

- dtbinding_invensense_icm42370p_spi
- dtbinding_invensense_icm42605
- dtbinding_invensense_icm42670p_spi
- dtbinding_invensense_icm42670p_i2c
- dtbinding_invensense_icm42670s_spi
- dtbinding_invensense_icm42670s_i2c
- dtbinding_invensense_icm42686
- dtbinding_invensense_icm42688
- dtbinding_invensense_icm4268x
- dtbinding_invensense_icm45605_i2c
- dtbinding_invensense_icm45605_i3c
- dtbinding_invensense_icm45605_spi
- dtbinding_invensense_icm45605s_i3c
- dtbinding_invensense_icm45605s_spi
- dtbinding_invensense_icm45605s_i2c
- dtbinding_invensense_icm45686_i2c
- dtbinding_invensense_icm45686_i3c
- dtbinding_invensense_icm45686_spi
- dtbinding_invensense_icm45686s_i2c
- dtbinding_invensense_icm45686s_i3c
- dtbinding_invensense_icm45686s_spi
- dtbinding_invensense_icm45688p_i3c
- dtbinding_invensense_icm45688p_i2c
- dtbinding_invensense_icm45688p_spi
- dtbinding_invensense_icm56622_i3c
- dtbinding_invensense_icm56622_i2c
- dtbinding_invensense_icm56622_spi
- dtbinding_invensense_icm56686_spi
- dtbinding_invensense_icm56686_i3c
- dtbinding_invensense_icm56686_i2c
- dtbinding_invensense_icp101xx
- dtbinding_invensense_icp201xx
- dtbinding_invensense_icp201xx_spi
- dtbinding_invensense_icp201xx_i2c
- dtbinding_invensense_mpu6050
- dtbinding_invensense_mpu9250
- dtbinding_invensense_tad2144_i2c
- dtbinding_invensense_tad2144
- dtbinding_invensense_tad2144_spi

- dtbinding_isentek_ist8310
- dtbinding_isil_isl29035
- dtbinding_ist_tsic_xx6
- dtbinding_ite_it8xxx2_vcmp
- dtbinding_jedec_jc_42.4_temp
- dtbinding_liteon_ltr329
- dtbinding_liteon_ltr553
- dtbinding_liteon_ltrf216a
- dtbinding_lm35
- dtbinding_lm75
- dtbinding_lm77
- dtbinding_maxbotix_mb7040
- dtbinding_maxim_ds18b20
- dtbinding_maxim_ds18s20
- dtbinding_maxim_ds3231_sensor
- dtbinding_maxim_max17055
- dtbinding_maxim_max17262
- dtbinding_maxim_max30101
- dtbinding_maxim_max30102
- dtbinding_maxim_max31790_fan_fault
- dtbinding_maxim_max31790_fan_speed
- dtbinding_maxim_max31826
- dtbinding_maxim_max31855_spi
- dtbinding_maxim_max31865
- dtbinding_maxim_max31875
- dtbinding_maxim_max32664c
- dtbinding_maxim_max44009
- dtbinding_maxim_max6675
- dtbinding_meas_htu21d
- dtbinding_meas_htu31d
- dtbinding_meas_ms5607_spi
- dtbinding_meas_ms5607_i2c
- dtbinding_meas_ms5637
- dtbinding_meas_ms5837_02ba
- dtbinding_meas_ms5837_30ba
- dtbinding_melexis_mlx90394
- dtbinding_memsic_mc3419
- dtbinding_memsic_mmc56x3
- dtbinding_microchip_mcp9600

- dtbinding_microchip_mcp970x
- dtbinding_microchip_mtch9010
- dtbinding_microchip_pac194x
- dtbinding_microchip_tcn75a
- dtbinding_microcrystal_rv3032_temp
- dtbinding_murata_ncp15wb473
- dtbinding_murata_ncp15xh103
- dtbinding_national_lm95234
- dtbinding_nordic_npm10xx_adc
- dtbinding_nordic_npm1300_charger
- dtbinding_nordic_npm1304_charger
- dtbinding_nordic_npm2100_vbat
- dtbinding_nordic_nrf_qdec
- dtbinding_nordic_nrf_temp
- dtbinding_nordic_nrf_temp_nrfs
- dtbinding_nordic_nrf_vbat
- dtbinding_ntc_thermistor_generic
- dtbinding_nuvoton_adc_cmp
- dtbinding_nuvoton_npcx_adc_v2t
- dtbinding_nxp_fxas21002_spi
- dtbinding_nxp_fxas21002_i2c
- dtbinding_nxp_fxls8974_i2c
- dtbinding_nxp_fxls8974_spi
- dtbinding_nxp_fxos8700_i2c
- dtbinding_nxp_fxos8700_spi
- dtbinding_nxp_kinetis_temperature
- dtbinding_nxp_lpadc_temp40
- dtbinding_nxp_mcux_eqdc
- dtbinding_nxp_mcux_qdc
- dtbinding_nxp_mcux_qdec
- dtbinding_nxp_p3t1755_i3c
- dtbinding_nxp_p3t1755_i2c
- dtbinding_nxp_pmc_tmpsns
- dtbinding_nxp_qdec_s32
- dtbinding_nxp_tempmon
- dtbinding_nxp_tempsense
- dtbinding_nxp_tmpsns
- dtbinding_nxp_tpm_qdec
- dtbinding_omron_2smpb_02e

- dtbinding_omron_d6f_p0001
- dtbinding_omron_d6f_p0010
- dtbinding_onnn_nct75
- dtbinding_panasonic_amg88xx
- dtbinding_peacefair_pzem004t
- dtbinding_phosense_xbr818
- dtbinding_pixart_paa3905
- dtbinding_pixart_paj7620
- dtbinding_pixart_pat9136
- dtbinding_plantower_pms7003
- dtbinding_plantower_pmsa003i
- dtbinding_pni_rm3100_i2c
- dtbinding_pni_rm3100_spi
- dtbinding_qst_qmi8658a
- dtbinding_raspberrypi_bcm283x_vc_thermal
- dtbinding_raspberrypi_pico_temp
- dtbinding_realtek_bee_aon_qdec
- dtbinding_realtek_bee_basic_qdec
- dtbinding_realtek_bee_qdec
- dtbinding_renesas_hs300x
- dtbinding_renesas_hs400x
- dtbinding_rohm_bd8lb600fs_diagnostics
- dtbinding_rohm_bh1730
- dtbinding_rohm_bh1750
- dtbinding_rohm_bh1790
- dtbinding_sbs_sbs_gauge
- dtbinding_sciosense_ens160_spi
- dtbinding_sciosense_ens160_i2c
- dtbinding_seeed_grove_light
- dtbinding_seeed_grove_temperature
- dtbinding_seeed_hm330x
- dtbinding_semtech_sx9500
- dtbinding_sensirion_scd40
- dtbinding_sensirion_scd41
- dtbinding_sensirion_sgp40
- dtbinding_sensirion_sht21
- dtbinding_sensirion_sht3xd
- dtbinding_sensirion_sht4x
- dtbinding_sensirion_shtcx

- dtbinding_sensirion_stcc4
- dtbinding_sensirion_sts4x
- dtbinding_sensylink_cht8315
- dtbinding_sifli_sf32lb_tsen
- dtbinding_silabs_emu_temp
- dtbinding_silabs_si7006
- dtbinding_silabs_si7055
- dtbinding_silabs_si7060
- dtbinding_silabs_si7210
- dtbinding_st_hts221_i2c
- dtbinding_st_hts221_spi
- dtbinding_st_i3g4250d
- dtbinding_st_iis2dh_i2c
- dtbinding_st_iis2dh_spi
- dtbinding_st_iis2dlpc_spi
- dtbinding_st_iis2dlpc_i2c
- dtbinding_st_iis2iclx_i2c
- dtbinding_st_iis2iclx_spi
- dtbinding_st_iis2mdc_i2c
- dtbinding_st_iis2mdc_spi
- dtbinding_st_iis328dq_spi
- dtbinding_st_iis328dq_i2c
- dtbinding_st_iis3dhhc_spi
- dtbinding_st_iis3dwb
- dtbinding_st_ilps22qs_spi
- dtbinding_st_ilps22qs_i3c
- dtbinding_st_ilps22qs_i2c
- dtbinding_st_ism330dhcx_spi
- dtbinding_st_ism330dhcx_i2c
- dtbinding_st_ism6hg256x_i3c
- dtbinding_st_ism6hg256x_i2c
- dtbinding_st_ism6hg256x_spi
- dtbinding_st_lis2de12_spi
- dtbinding_st_lis2de12_i2c
- dtbinding_st_lis2dh_spi
- dtbinding_st_lis2dh_i2c
- dtbinding_st_lis2dh12_i2c
- dtbinding_st_lis2ds12_i2c
- dtbinding_st_lis2ds12_spi

- dtbinding_st_lis2du12_i2c
- dtbinding_st_lis2du12_spi
- dtbinding_st_lis2dux12_i2c
- dtbinding_st_lis2dux12_spi
- dtbinding_st_lis2duxs12_i2c
- dtbinding_st_lis2duxs12_spi
- dtbinding_st_lis2dw12_i2c
- dtbinding_st_lis2dw12_spi
- dtbinding_st_lis2mdl_i2c
- dtbinding_st_lis2mdl_spi
- dtbinding_st_lis3dh_i2c
- dtbinding_st_lis3mdl_magn
- dtbinding_st_lps22df_i2c
- dtbinding_st_lps22df_i3c
- dtbinding_st_lps22df_spi
- dtbinding_st_lps22hb_press
- dtbinding_st_lps22hh_i2c
- dtbinding_st_lps22hh_i3c
- dtbinding_st_lps22hh_spi
- dtbinding_st_lps25hb_press
- dtbinding_st_lps28dfw_i3c
- dtbinding_st_lps28dfw_i2c
- dtbinding_st_lsm303agr_accel_spi
- dtbinding_st_lsm303agr_accel_i2c
- dtbinding_st_lsm303dlhc_accel
- dtbinding_st_lsm303dlhc_magn
- dtbinding_st_lsm6ds0
- dtbinding_st_lsm6dsl_spi
- dtbinding_st_lsm6dsl_i2c
- dtbinding_st_lsm6dso_i2c
- dtbinding_st_lsm6dso_spi
- dtbinding_st_lsm6dso16is_spi
- dtbinding_st_lsm6dso16is_i2c
- dtbinding_st_lsm6dso32_i2c
- dtbinding_st_lsm6dso32_spi
- dtbinding_st_lsm6dsv16x_i3c
- dtbinding_st_lsm6dsv16x_i2c
- dtbinding_st_lsm6dsv16x_spi
- dtbinding_st_lsm6dsv320x_i3c

- dtbinding_st_lsm6dsv320x_spi
- dtbinding_st_lsm6dsv320x_i2c
- dtbinding_st_lsm6dsv32x_i3c
- dtbinding_st_lsm6dsv32x_i2c
- dtbinding_st_lsm6dsv32x_spi
- dtbinding_st_lsm6dsv80x_spi
- dtbinding_st_lsm6dsv80x_i3c
- dtbinding_st_lsm6dsv80x_i2c
- dtbinding_st_lsm9ds0_gyro_i2c
- dtbinding_st_lsm9ds0_mfd_i2c
- dtbinding_st_lsm9ds1
- dtbinding_st_lsm9ds1_mag
- dtbinding_st_stm32_digi_temp
- dtbinding_st_stm32_qdec
- dtbinding_st_stm32_temp
- dtbinding_st_stm32_temp_cal
- dtbinding_st_stm32_vbat
- dtbinding_st_stm32_vddcore
- dtbinding_st_stm32_vref
- dtbinding_st_stm32c0_temp_cal
- dtbinding_st_stts22h_i2c
- dtbinding_st_stts751_i2c
- dtbinding_st_vl53l0x
- dtbinding_st_vl53l1x
- dtbinding_st_vl53l5cx
- dtbinding_tdk_ntcgxx3jx103x
- dtbinding_ti_bq274xx
- dtbinding_ti_fdc1004
- dtbinding_ti_fdc2x1x
- dtbinding_ti_hdc
- dtbinding_ti_hdc2010
- dtbinding_ti_hdc2021
- dtbinding_ti_hdc2022
- dtbinding_ti_hdc2080
- dtbinding_ti_hdc302x
- dtbinding_ti_ina219
- dtbinding_ti_ina226
- dtbinding_ti_ina228
- dtbinding_ti_ina230

- dtbinding_ti_ina232
- dtbinding_ti_ina236
- dtbinding_ti_ina237
- dtbinding_ti_ina3221
- dtbinding_ti_ina7xx
- dtbinding_ti_opt3001
- dtbinding_ti_opt3004
- dtbinding_ti_tmag3001
- dtbinding_ti_tmag5170
- dtbinding_ti_tmag5273
- dtbinding_ti_tmp007
- dtbinding_ti_tmp1075
- dtbinding_ti_tmp108
- dtbinding_ti_tmp112
- dtbinding_ti_tmp114
- dtbinding_ti_tmp11x
- dtbinding_ti_tmp435
- dtbinding_ti_tmp451
- dtbinding_vishay_vcnl36825t
- dtbinding_vishay_vcnl4040
- dtbinding_vishay_veml6031
- dtbinding_vishay_veml6046
- dtbinding_vishay_veml7700
- dtbinding_we_wsen_hids_2525020210002
- dtbinding_we_wsen_isds_2536030320001_i2c
- dtbinding_we_wsen_isds_2536030320001_spi
- dtbinding_we_wsen_itds_2533020201601_i2c
- dtbinding_we_wsen_itds_2533020201601_spi
- dtbinding_we_wsen_pads_2511020213301_i2c
- dtbinding_we_wsen_pads_2511020213301_spi
- dtbinding_we_wsen_pdms_25131308XXX05_i2c
- dtbinding_we_wsen_pdms_25131308XXX05_spi
- dtbinding_we_wsen_pdus_25131308XXXXX
- dtbinding_we_wsen_tids_2521020222501
- dtbinding_winsen_mhz19b
- dtbinding_zephyr_flow_meter
- dtbinding_zephyr_generic_emul_sensor
- dtbinding_zephyr_native_linux_temp
- dtbinding_zephyr_sensing

- dtbinding_zephyr_sensing_hinge_angle
- dtbinding_zephyr_sensing_phy_3d_sensor

SENT (Single Edge Nibble Transmission)

- dtbinding_nxp_s32_sent

Serial controller

- dtbinding_adi_max32_uart
- dtbinding_aesc_uart
- dtbinding_altr_jtag_uart
- dtbinding_altr_uart
- dtbinding_ambiq_pl011_uart
- dtbinding_ambiq_uart
- dtbinding_arm_cmsdk_uart
- dtbinding_arm_pl011
- dtbinding_arm_sbsa_uart
- dtbinding_atmel_sam_uart
- dtbinding_atmel_sam_usart
- dtbinding_atmel_sam0_uart
- dtbinding_bflb_uart
- dtbinding_brcm_bcm283x_aux_uart
- dtbinding_cdns_uart
- dtbinding_cypress_psoc6_uart
- dtbinding_efinix_sapphire_uart0
- dtbinding_elan_em32_uart
- dtbinding_ene_kb106x_uart
- dtbinding_ene_kb1200_uart
- dtbinding_espressif_esp32_lpuart
- dtbinding_espressif_esp32_uart
- dtbinding_espressif_esp32_usb_serial
- dtbinding_focaltech_ft9001_usart
- dtbinding_gaisler_apbuart
- dtbinding_gd_gd32_usart
- dtbinding_infineon_asclin_uart
- dtbinding_infineon_uart
- dtbinding_infineon_xmc4xxx_uart
- dtbinding_intel_lw_uart
- dtbinding_intel_sedi_uart
- dtbinding_ite_it51xxx_uart
- dtbinding_ite_it8xxx2_uart

- dtbinding_litex_uart
- dtbinding_lowrisc_opentitan_uart
- dtbinding_mediatek_mt8188_uart
- dtbinding_microchip_coreuart
- dtbinding_microchip_dbgu_g1_uart
- dtbinding_microchip_sercom_g1_uart
- dtbinding_microchip_uart_g1
- dtbinding_microchip_xec_uart
- dtbinding_neorv32_uart
- dtbinding_nordic_nrf_uart
- dtbinding_nordic_nrf_uarte
- dtbinding_ns16550
- dtbinding_nuvoton_npcx_uart
- dtbinding_nuvoton_npcx_uart_npckn
- dtbinding_nuvoton_numaker_uart
- dtbinding_nuvoton_numicro_uart
- dtbinding_nxp_imx_iuart
- dtbinding_nxp_imx_uart
- dtbinding_nxp_kinetis_lpsci
- dtbinding_nxp_kinetis_uart
- dtbinding_nxp_lpc_usart
- dtbinding_nxp_lpc11u6x_uart
- dtbinding_nxp_lpc84x_uart
- dtbinding_nxp_lpuart
- dtbinding_nxp_s32_linflexd
- dtbinding_openisa_rv32m1_lpuart
- dtbinding_quicklogic_usbserialport_s3b
- dtbinding_raspberrypi_pico_uart
- dtbinding_raspberrypi_pico_uart_pio
- dtbinding_realtek_ameba_loguart
- dtbinding_realtek_bee_uart
- dtbinding_realtek_rts5912_uart
- dtbinding_renesas_ra_sci_uart
- dtbinding_renesas_ra_uart_sau
- dtbinding_renesas_ra8_uart_sci_b
- dtbinding_renesas_rcar_hscif
- dtbinding_renesas_rcar_scif
- dtbinding_renesas_rx_uart_sci
- dtbinding_renesas_rx_uart_sci_b

- dtbinding_renesas_rx_uart_sci_qemu
- dtbinding_renesas_rz_sci_b_uart
- dtbinding_renesas_rz_sci_uart
- dtbinding_renesas_rz_scif_uart
- dtbinding_renesas_rza2m_scif_uart
- dtbinding_renesas_rzt2m_uart
- dtbinding_renesas_smartbond_uart
- dtbinding_rpmsg_uart
- dtbinding_segger_rtt_uart
- dtbinding_sensry_sy1xx_uart
- dtbinding_shakti_uart
- dtbinding_sifive_uart0
- dtbinding_sifli_sf32lb_usart
- dtbinding_silabs_eusart_uart
- dtbinding_silabs_gecko_leuart
- dtbinding_silabs_gecko_uart
- dtbinding_silabs_gecko_usart
- dtbinding_silabs_si32_usart
- dtbinding_silabs_usart_uart
- dtbinding_snps_dw_apb_uart
- dtbinding_snps_hostlink_uart
- dtbinding_snps_nsim_uart
- dtbinding_st_stm32_lpuart
- dtbinding_st_stm32_uart
- dtbinding_st_stm32_usart
- dtbinding_telink_b91_uart
- dtbinding_ti_cc13xx_cc26xx_uart
- dtbinding_ti_cc23x0_uart
- dtbinding_ti_cc32xx_uart
- dtbinding_ti_msp432p4xx_uart
- dtbinding_ti_mspm0_uart
- dtbinding_ti_stellaris_uart
- dtbinding_ti_tiva_c_uart
- dtbinding_virtio_console
- dtbinding_wch_ch5xx_uart
- dtbinding_wch_sdi_console
- dtbinding_wch_usart
- dtbinding_xen_hvc_consoleio
- dtbinding_xen_hvc_uart

- dtbinding_xlnx_xps_uartlite_1.00.a
- dtbinding_xlnx_xuartps
- dtbinding_zephyr_cdc_acm_uart
- dtbinding_zephyr_native_pty_uart
- dtbinding_zephyr_native_tty_uart
- dtbinding_zephyr_nus_uart
- dtbinding_zephyr_uart_bitbang
- dtbinding_zephyr_uart_bridge
- dtbinding_zephyr_uart_emul

Service in Platform

- dtbinding_intel_socfpga_agilex_sip_smc

SHI (Serial Host Interface)

- dtbinding_ite_it8xxx2_shi
- dtbinding_nuvoton_npcx_shi
- dtbinding_nuvoton_npcx_shi_enhanced

SMBUS (System Management Bus)

- dtbinding_intel_pch_smbus
- dtbinding_ite_it51xxx_smbus
- dtbinding_st_stm32_smbus

Sound

- dtbinding_cirrus_cs47l63

SPI

- dtbinding_adi_max32_spi
- dtbinding_ambiq_mspi
- dtbinding_ambiq_spi
- dtbinding_ambiq_spi_bleif
- dtbinding_ambiq_spid
- dtbinding_andestech_atcspi200
- dtbinding_arm_pl022
- dtbinding_atmel_sam_spi
- dtbinding_atmel_sam0_spi
- dtbinding_bflb_spi
- dtbinding_cdns_spi
- dtbinding_cypress_psoc6_spi
- dtbinding_egis_et171_spi
- dtbinding_espressif_esp32_spi

- dtbinding_gaisler_spimctrl
- dtbinding_gd_gd32_spi
- dtbinding_infineon_spi
- dtbinding_infineon_xmc4xxx_spi
- dtbinding_intel_penwell_spi
- dtbinding_intel_sedi_spi
- dtbinding_ite_it51xxx_spi
- dtbinding_ite_it8xxx2_spi
- dtbinding_litex_spi
- dtbinding_litex_spi_litespi
- dtbinding_lowrisc_opentitan_spi
- dtbinding_microchip_flexcom_g1_spi
- dtbinding_microchip_mpfs_qspi
- dtbinding_microchip_mpfs_spi
- dtbinding_microchip_sercom_g1_spi
- dtbinding_microchip_xec_qmspi
- dtbinding_microchip_xec_qmspi_ldma
- dtbinding_nordic_nrf_spi
- dtbinding_nordic_nrf_spim
- dtbinding_nordic_nrf_spis
- dtbinding_nuvoton_npcx_spip
- dtbinding_nuvoton_numaker_spi
- dtbinding_nuvoton_numaker_usci_spi
- dtbinding_nxp_dspi
- dtbinding_nxp_flexio_spi
- dtbinding_nxp_imx_ecspi
- dtbinding_nxp_imx_flexspi
- dtbinding_nxp_kinetis_l_spi
- dtbinding_nxp_lpc_spi
- dtbinding_nxp_lpspi
- dtbinding_nxp_s32_spi
- dtbinding_nxp_sc18is606_spi
- dtbinding_opencores_spi_simple
- dtbinding_openisa_rv32m1_lpspi
- dtbinding_raspberrypi_pico_spi
- dtbinding_raspberrypi_pico_spi_pio
- dtbinding_realtek_ameba_spi
- dtbinding_realtek_bee_spi
- dtbinding_realtek_rts5817_spi

- dtbinding_realtek_rts5912_spi
- dtbinding_renesas_ra_spi
- dtbinding_renesas_ra_spi_sci
- dtbinding_renesas_ra_spi_sci_b
- dtbinding_renesas_ra8_spi_b
- dtbinding_renesas_rx_rspi
- dtbinding_renesas_rz_rspi
- dtbinding_renesas_rz_spi
- dtbinding_renesas_rz_spi_b
- dtbinding_renesas_smartbond_spi
- dtbinding_sensry_sy1xx_spi
- dtbinding_sifive_spi0
- dtbinding_sifli_sf32lb_spi
- dtbinding_silabs_eusart_spi
- dtbinding_silabs_gspi
- dtbinding_silabs_usart_spi
- dtbinding_snps_designware_spi
- dtbinding_st_stm32_spi
- dtbinding_st_stm32_spi_fifo
- dtbinding_st_stm32_spi_host_cmd
- dtbinding_st_stm32_spi_subghz
- dtbinding_st_stm32h7_spi
- dtbinding_telink_b91_spi
- dtbinding_ti_cc13xx_cc26xx_spi
- dtbinding_ti_cc23x0_spi
- dtbinding_ti_mspm0_spi
- dtbinding_ti_omap_mcspi
- dtbinding_virtio_spi
- dtbinding_wch_spi
- dtbinding_xlnx_xps_spi_2.00.a
- dtbinding_xlnx_zynqmp_qspi_1.0
- dtbinding_zephyr_spi_bitbang
- dtbinding_zephyr_spi_emul_controller

SRAM (Static Random Access Memory)

- dtbinding_mmio_sram
- dtbinding_sifive_dtim0

Stepper

- dtbinding_adi_tmc2209
- dtbinding_adi_tmc50xx
- dtbinding_adi_tmc50xx_stepper_ctrl
- dtbinding_adi_tmc50xx_stepper_driver
- dtbinding_adi_tmc51xx_spi
- dtbinding_adi_tmc51xx_uart
- dtbinding_adi_tmc51xx_stepper_ctrl
- dtbinding_adi_tmc51xx_stepper_driver
- dtbinding_adi_tmcm3216
- dtbinding_adi_tmcm3216_stepper_ctrl
- dtbinding_adi_tmcm3216_stepper_driver
- dtbinding_allegro_a4979
- dtbinding_ti_drv84xx
- dtbinding_zephyr_fake_stepper_ctrl
- dtbinding_zephyr_fake_stepper_driver
- dtbinding_zephyr_gpio_step_dir_stepper_ctrl
- dtbinding_zephyr_h_bridge_stepper_ctrl

System controller

- dtbinding_aesc_syscon
- dtbinding_sifli_sf32lb_cfg
- dtbinding_syscon
- dtbinding_ti_control_module

Tachometer

- dtbinding_ene_kb106x_tach
- dtbinding_ene_kb1200_tach
- dtbinding_ite_it51xxx_tach
- dtbinding_ite_it8xxx2_tach
- dtbinding_microchip_xec_tach
- dtbinding_nuvoton_npcx_tach
- dtbinding_realtek_rts5912_tach
- dtbinding_zephyr_tach_gpio

Test

- dtbinding_test_gpio_enable_disable_interrupt

Timer

- dtbinding_adi_max32_rv32_sys_timer
- dtbinding_adi_max32_timer
- dtbinding_adi_max32_wut_timer
- dtbinding_ambiq_ctimer
- dtbinding_ambiq_stimer
- dtbinding_ambiq_timer
- dtbinding_amd_xps_timer_1.00.a
- dtbinding_arm_armv6m_systick
- dtbinding_arm_armv7_timer
- dtbinding_arm_armv7m_systick
- dtbinding_arm_armv8_timer
- dtbinding_arm_armv8.1m_systick
- dtbinding_arm_armv8m_systick
- dtbinding_efinix_sapphire_timer0
- dtbinding_espressif_esp32_systimer
- dtbinding_gaisler_gptimer
- dtbinding_infineon_aurix_stm
- dtbinding_infineon_cat1_lp_timer_pdl
- dtbinding_infineon_lp_timer
- dtbinding_infineon_tcpwm
- dtbinding_intel_adsp_timer
- dtbinding_intel_hpet
- dtbinding_ite_it51xxx_timer
- dtbinding_ite_it8xxx2_timer
- dtbinding_litex_timer0
- dtbinding_mediatek_ostimer64
- dtbinding_microchip_pit_g1_timer
- dtbinding_microchip_sam_pit64b
- dtbinding_microchip_xec_basic_timer
- dtbinding_microchip_xec_rtos_timer
- dtbinding_nordic_nrf_grtc
- dtbinding_nuclei_systimer
- dtbinding_nuvoton_npcx_itim_timer
- dtbinding_nxp_ftm
- dtbinding_nxp_gpt_hw_timer
- dtbinding_nxp_os_timer
- dtbinding_openisa_rv32m1_lptmr
- dtbinding_qcom_hvm_timer

- dtbinding_realtek_bee_basic_timer
- dtbinding_realtek_bee_enhanced_timer
- dtbinding_realtek_bee_timer
- dtbinding_realtek_rts5912_rtmr
- dtbinding_renesas_ra_ulpt_timer
- dtbinding_renesas_rcar_cmt
- dtbinding_renesas_rx_timer_cmt
- dtbinding_renesas_rx_timer_cmt_start_control
- dtbinding_renesas_rz_cmtw
- dtbinding_renesas_rz_gpt
- dtbinding_renesas_rz_gtm
- dtbinding_renesas_rz_gtm_os_timer
- dtbinding_renesas_rz_mtu
- dtbinding_renesas_rza2m_gpt
- dtbinding_renesas_rza2m_ostm
- dtbinding_renesas_rza2m_ostm_timer
- dtbinding_riscv_machine_timer
- dtbinding_sensry_sy1xx_sys_timer
- dtbinding_sifli_sf32lb_atim
- dtbinding_sifli_sf32lb_gptim
- dtbinding_silabs_gecko_burtc
- dtbinding_silabs_gecko_timer
- dtbinding_silabs_series2_letimer
- dtbinding_silabs_series2_timer
- dtbinding_snps_arc_timer
- dtbinding_st_stm32_lptim
- dtbinding_st_stm32_timers
- dtbinding_st_stm32u5_lptim
- dtbinding_st_stm32wb0_radio_timer
- dtbinding_ti_am26_rtitimer
- dtbinding_ti_am654_timer
- dtbinding_ti_cc13xx_cc26xx_timer
- dtbinding_ti_cc23x0_systim_timer
- dtbinding_ti_mspm0_timer
- dtbinding_wch_systick
- dtbinding_xlnx_ttcps

Timestamp

- dtbinding_intel_ace_timestamp

USB (Universal Serial Bus)

- dtbinding_adi_max32_usbhs
- dtbinding_ambiq_usb
- dtbinding_atmel_sam_udp
- dtbinding_atmel_sam_usbc
- dtbinding_atmel_sam_usbhs
- dtbinding_atmel_sam0_usb
- dtbinding_bflb_udc_1
- dtbinding_brcm_bcm2835_usb
- dtbinding_diodes_pi3usb9201
- dtbinding_espressif_esp32_usb_otg_fs
- dtbinding_espressif_esp32_usb_otg_hs
- dtbinding_infineon_usbhs
- dtbinding_infineon_xmc4xxx_usb
- dtbinding_ite_it51xxx_usb
- dtbinding_ite_it82xx2_usb
- dtbinding_maxim_max3421e_spi
- dtbinding_microchip_udphs_g1_udc
- dtbinding_nordic_nrf_usbd
- dtbinding_nordic_nrf_usbhs
- dtbinding_nordic_nrf_usbhs_bc12
- dtbinding_nordic_nrf_usbhs_phy_type1
- dtbinding_nordic_nrf_usbhs_phy_type2
- dtbinding_nordic_nrf_usbhs_wrapper
- dtbinding_nuvoton_numaker_hsusbd
- dtbinding_nuvoton_numaker_usbd
- dtbinding_nxp_ehci
- dtbinding_nxp_kinetis_usbd
- dtbinding_nxp_lpcip3511
- dtbinding_nxp_uhc_ehci
- dtbinding_nxp_uhc_ip3516hs
- dtbinding_nxp_uhc_khci
- dtbinding_nxp_uhc_ohci
- dtbinding_nxp_usbphy
- dtbinding_raspberrypi_pico_usbd
- dtbinding_renesas_ra_udc
- dtbinding_renesas_ra_uhc
- dtbinding_renesas_ra_usbfs
- dtbinding_renesas_ra_usbhs

- dtbinding_renesas_smartbond_usbd
- dtbinding_snps_designware_usb
- dtbinding_snps_dwc2
- dtbinding_st_stm32_otgfs
- dtbinding_st_stm32_otghs
- dtbinding_st_stm32_usb
- dtbinding_st_stm32f4_fsotg
- dtbinding_st_stm32n6_otghs
- dtbinding_usb_audio
- dtbinding_usb_audio_feature_volume
- dtbinding_usb_audio_hp
- dtbinding_usb_audio_hs
- dtbinding_usb_audio_mic
- dtbinding_zephyr_hid_device
- dtbinding_zephyr_midi2_device
- dtbinding_zephyr_native_posix_udc
- dtbinding_zephyr_uac2
- dtbinding_zephyr_uac2_audio_streaming
- dtbinding_zephyr_uac2_clock_source
- dtbinding_zephyr_uac2_feature_unit
- dtbinding_zephyr_uac2_input_terminal
- dtbinding_zephyr_uac2_output_terminal
- dtbinding_zephyr_udc_skeleton
- dtbinding_zephyr_udc_virtual
- dtbinding_zephyr_uhc_virtual
- dtbinding_zephyr_uvc_device

USB Audio Offload Link

- dtbinding_intel_adsp_uaol
- dtbinding_intel_uaol_dai

USB Type-C

- dtbinding_ite_it8xxx2_usbpd
- dtbinding_nuvoton_numaker_vbus
- dtbinding_usb_c_connector
- dtbinding_zephyr_usb_c_pwrctrl
- dtbinding_zephyr_usb_c_vbus_adc
- dtbinding_zephyr_usb_c_vbus_tcpci

USB Type-C Port Controller

- dtbinding_nuvoton_numaker_tcpc
- dtbinding_onnn_fusb307_tcpc
- dtbinding_parade_ps8xxx
- dtbinding_richtek_rt1715
- dtbinding_st_stm32_ucpd

Video

- dtbinding_aptina_mt9m114
- dtbinding_arducam_mega
- dtbinding_espressif_esp32_lcd_cam_dvp
- dtbinding_galaxycore_gc2145
- dtbinding_himax_hm01b0
- dtbinding_himax_hm0360
- dtbinding_nxp_flexio_camera
- dtbinding_nxp_imx_csi
- dtbinding_nxp_jpegdec
- dtbinding_nxp_mipi_csi2rx
- dtbinding_nxp_pngdec
- dtbinding_nxp_video_smartdma
- dtbinding_ovti_ov2640
- dtbinding_ovti_ov5640
- dtbinding_ovti_ov5642
- dtbinding_ovti_ov7670
- dtbinding_ovti_ov7675
- dtbinding_ovti_ov7725
- dtbinding_ovti_ov9655
- dtbinding_renesas_ra_ceu
- dtbinding_sony_imx219
- dtbinding_sony_imx335
- dtbinding_st_mipid02
- dtbinding_st_stm32_dcmi
- dtbinding_st_stm32_dcmipp
- dtbinding_st_stm32_jpeg
- dtbinding_st_stm32_venc
- dtbinding_zephyr_native_sim_video_fifo
- dtbinding_zephyr_video_emul_imager
- dtbinding_zephyr_video_emul_rx
- dtbinding_zephyr_video_sw_generator

Virtio

- dtbinding_virtio_mmio
- dtbinding_virtio_pci

Virtualization

- dtbinding_qemu_ivshmem

Wakeup Controller

- dtbinding_nxp_llwu
- dtbinding_nxp_sleepcon_wuc
- dtbinding_nxp_wuc_wuu

Watchdog

- dtbinding_adi_max32_watchdog
- dtbinding_adi_max42500_watchdog
- dtbinding_ambiq_watchdog
- dtbinding_andestech_atcwdt200
- dtbinding_arm_cmsdk_watchdog
- dtbinding_arm_crsas_ma2_watchdog
- dtbinding_atmel_sam_watchdog
- dtbinding_atmel_sam0_watchdog
- dtbinding_atmel_sam4l_watchdog
- dtbinding_bflb_wdt
- dtbinding_ene_kb106x_watchdog
- dtbinding_ene_kb1200_watchdog
- dtbinding_espressif_esp32_watchdog
- dtbinding_espressif_esp32_xt_wdt
- dtbinding_gd_gd32_fwdgt
- dtbinding_gd_gd32_wwdgt
- dtbinding_infineon_watchdog
- dtbinding_infineon_xmc4xxx_watchdog
- dtbinding_intel_adsp_watchdog
- dtbinding_intel_tco_wdt
- dtbinding_ite_it51xxx_watchdog
- dtbinding_ite_it8xxx2_watchdog
- dtbinding_litex_watchdog
- dtbinding_lowrisc_opentitan_aontimer
- dtbinding_m5stack_m5pm1_wdt
- dtbinding_microchip_wdt_g1
- dtbinding_microchip_xec_watchdog

- dtbinding_nordic_npm10xx_wdt
- dtbinding_nordic_npm1300_wdt
- dtbinding_nordic_npm1304_wdt
- dtbinding_nordic_npm2100_wdt
- dtbinding_nordic_npm6001_wdt
- dtbinding_nordic_nrf_gswdt
- dtbinding_nordic_nrf_wdt
- dtbinding_nuvoton_npcx_watchdog
- dtbinding_nuvoton_numaker_wdt
- dtbinding_nuvoton_numaker_wwdt
- dtbinding_nxp_cop
- dtbinding_nxp_ewm
- dtbinding_nxp_fs26_wdog
- dtbinding_nxp_imx_wdog
- dtbinding_nxp_kinetis_wdog
- dtbinding_nxp_lpc_wwdt
- dtbinding_nxp_rtwdog
- dtbinding_nxp_s32_swt
- dtbinding_nxp_wdog32
- dtbinding_raspberrypi_pico_watchdog
- dtbinding_realtek_ameba_watchdog
- dtbinding_realtek_bee_core_wdt
- dtbinding_realtek_rts5817_watchdog
- dtbinding_realtek_rts5912_watchdog
- dtbinding_renesas_ra_wdt
- dtbinding_renesas_rx_iwdt
- dtbinding_renesas_rz_wdt
- dtbinding_renesas_smartbond_watchdog
- dtbinding_sifive_wdt
- dtbinding_sifli_sf32lb_wdt
- dtbinding_silabs_gecko_wdog
- dtbinding_silabs_siwx91x_wdt
- dtbinding_silabs_wdog
- dtbinding_snps_designware_watchdog
- dtbinding_st_stm32_watchdog
- dtbinding_st_stm32_window_watchdog
- dtbinding_ti_cc13xx_cc26xx_watchdog
- dtbinding_ti_cc23x0_wdt
- dtbinding_ti_cc32xx_watchdog

- dtbinding_ti_j7_rti_wdt
- dtbinding_ti_mspm0_watchdog
- dtbinding_ti_tps382x
- dtbinding_wch_iwdg
- dtbinding_xen_watchdog
- dtbinding_xlnx_versal_wwdt
- dtbinding_xlnx_xps_timebase_wdt_1.00.a
- dtbinding_zephyr_counter_watchdog

Wi-Fi

- dtbinding_atmel_winc1500
- dtbinding_bflb_wifi4
- dtbinding_bflb_wifi6
- dtbinding_espressif_esp_at
- dtbinding_espressif_esp_hosted
- dtbinding_espressif_esp_hosted_mcu_sd
- dtbinding_espressif_esp_hosted_mcu
- dtbinding_espressif_esp_hosted_mcu_wifi
- dtbinding_espressif_esp32_wifi
- dtbinding_infineon_airoc_wifi
- dtbinding_infineon_airoc_wifi_spi
- dtbinding_inventek_eswifi
- dtbinding_inventek_eswifi_uart
- dtbinding_nordic_nrf7000_coex
- dtbinding_nordic_nrf7000_qspi
- dtbinding_nordic_nrf7000_spi
- dtbinding_nordic_nrf7001_coex
- dtbinding_nordic_nrf7001_qspi
- dtbinding_nordic_nrf7001_spi
- dtbinding_nordic_nrf7002_coex
- dtbinding_nordic_nrf7002_qspi
- dtbinding_nordic_nrf7002_spi
- dtbinding_nordic_nrf7120_wifi
- dtbinding_nordic_wlan
- dtbinding_nxp_wifi
- dtbinding_realtek_ameba_wifi
- dtbinding_silabs_siwx91x_wifi
- dtbinding_st_st67w611m1
- dtbinding_zephyr_native_sim_wifi

- dtbinding_zephyr_wifi_hwsim

Xen

- dtbinding_xen_xen

xSPI (Expanded Serial Peripheral Interface)

- dtbinding_nxp_s32_xspi
- dtbinding_nxp_s32_xspi_sfp_frad
- dtbinding_nxp_s32_xspi_sfp_mdad
- dtbinding_nxp_xspi
- dtbinding_st_stm32_xspi
- dtbinding_st_stm32_xspim

Bindings by vendor **Vendor name (vendor prefix)**

Generic or vendor-independent

- dtbinding_adafruit_feather_header
- dtbinding_adc_keys
- dtbinding_ambiq_header
- dtbinding_analog_axis
- dtbinding_aps_z8_mspi
- dtbinding_aps6404l_mspi
- dtbinding_arduino_header_pwm
- dtbinding_arduino_header_r3
- dtbinding_arduino_mega_header
- dtbinding_arduino_mkr_header
- dtbinding_arduino_nano_header
- dtbinding_atmel_xplained_header
- dtbinding_atmel_xplained_pro_header
- dtbinding_atxp032_mspi
- dtbinding_can_transceiver_gpio
- dtbinding_charlieplex_led_matrix
- dtbinding_current_sense_amplifier
- dtbinding_current_sense_shunt
- dtbinding_dac_leds
- dtbinding_esp_01_header
- dtbinding_ethernet_phy
- dtbinding_ethernet_phy_fixed_link
- dtbinding_fixed_clock
- dtbinding_fixed_factor_clock

- dtbinding_fixed_layout
- dtbinding_fixed_partitions
- dtbinding_fixed_subpartitions
- dtbinding_gnss_nmea_generic
- dtbinding_gpio_7_segment
- dtbinding_gpio_buzzer
- dtbinding_gpio_i2c
- dtbinding_gpio_i2c_switch
- dtbinding_gpio_kbd_matrix
- dtbinding_gpio_keys
- dtbinding_gpio_leds
- dtbinding_gpio_mmio_latch
- dtbinding_gpio_mux
- dtbinding_gpio_qdec
- dtbinding_grove_header
- dtbinding_hc_sr04
- dtbinding_input_keymap
- dtbinding_is25xX0xx_mspi
- dtbinding_is66wv_mspi
- dtbinding_led_strip_matrix
- dtbinding_leds_group_multicolor
- dtbinding_lin_transceiver_gpio
- dtbinding_lm35
- dtbinding_lm75
- dtbinding_lm77
- dtbinding_mikro_bus
- dtbinding_mmio_sram
- dtbinding_mr_mcxn_t1_header
- dtbinding_nordic_thingy53_edge_connector
- dtbinding_ns16550
- dtbinding_ntc_thermistor_generic
- dtbinding_nvme_controller
- dtbinding_particle_gen3_header
- dtbinding_pci_host_ecam_generic
- dtbinding_pcie_controller
- dtbinding_power_domain
- dtbinding_power_domain_gpio
- dtbinding_power_domain_gpio_monitor
- dtbinding_power_domain_soc_state_change

- dtbinding_pwm_buzzer
- dtbinding_pwm_clock
- dtbinding_pwm_leds
- dtbinding_radio_fem_two_ctrl_pins
- dtbinding_radio_gpio_coex
- dtbinding_raspberrypi_40pins_header
- dtbinding_regulator_fixed
- dtbinding_regulator_gpio
- dtbinding_reset_mmio
- dtbinding_riscv
- dtbinding_rpmsg_uart
- dtbinding_sample_controller
- dtbinding_shared_irq
- dtbinding_slcd_panel
- dtbinding_soc_nv_flash
- dtbinding_st_morpho_header
- dtbinding_st_zio_header
- dtbinding_stemma_qt_connector
- dtbinding_swj_connector
- dtbinding_syscon
- dtbinding_test_gpio_enable_disable_interrupt
- dtbinding_tsc_keys
- dtbinding_usb_audio
- dtbinding_usb_audio_feature_volume
- dtbinding_usb_audio_hp
- dtbinding_usb_audio_hs
- dtbinding_usb_audio_mic
- dtbinding_usb_c_connector
- dtbinding_usb_nop_xceiv
- dtbinding_usb_ulpi_phy
- dtbinding_voltage_divider
- dtbinding_wisblock_adc
- dtbinding_wisblock_io_slot
- dtbinding_wisblock_pwm
- dtbinding_wisblock_sensor_slot

AD Semiconductor Co., Ltd. (adsemi)

- dtbinding_adsemi_tsm12

Adafruit Industries, LLC (adafruit)

- dtbinding_adafruit_seesaw_gamepad

ADH Technology Co. Ltd. (adh-tech)

- dtbinding_adh_tech_gt5x

Advanced Micro Devices (AMD), Inc. (amd)

- dtbinding_amd_acp_host_dma
- dtbinding_amd_acp_intc
- dtbinding_amd_acp_sdw_dai
- dtbinding_amd_acp_sdw_dma
- dtbinding_amd_acp_tdm_dma
- dtbinding_amd_sb_tsi
- dtbinding_amd_tdm_dai
- dtbinding_amd_versal2_dma_1.0
- dtbinding_amd_xps_timer_1.00.a

Aesc Silicon (aesc)

- dtbinding_aesc_clock_controller
- dtbinding_aesc_gpio
- dtbinding_aesc_pinctrl
- dtbinding_aesc_spi_flash_controller
- dtbinding_aesc_syscon
- dtbinding_aesc_uart

Alibaba Xuantie (xuantie)

- dtbinding_xuantie_e907

Alif Semiconductor (alif)

- dtbinding_alif_clock_mux
- dtbinding_alif_clockctrl
- dtbinding_alif_pinctrl

Allegro DVT (allegro)

- dtbinding_allegro_a4979
- dtbinding_allegro_als31300

Allwinner Technology Co., Ltd. (allwinner)

- dtbinding_allwinner_sun50i_h618_gpio
- dtbinding_allwinner_sunxi_gpio

Altera Corp. (altr)

- dtbinding_altr_jtag_uart
- dtbinding_altr_msgdma
- dtbinding_altr_nios2_i2c
- dtbinding_altr_nios2_qspi
- dtbinding_altr_nios2_qspi_nor
- dtbinding_altr_pio_1.0
- dtbinding_altr_uart

Ambiq Micro, Inc. (ambiq)

- dtbinding_ambiq_adc
- dtbinding_ambiq_am1805
- dtbinding_ambiq_apollo2_pinctrl
- dtbinding_ambiq_apollo3_pinctrl
- dtbinding_ambiq_apollo4_pinctrl
- dtbinding_ambiq_apollo5_pinctrl
- dtbinding_ambiq_bt_hci_spi
- dtbinding_ambiq_clkctrl
- dtbinding_ambiq_counter
- dtbinding_ambiq_ctimer
- dtbinding_ambiq_ctimer_pwm
- dtbinding_ambiq_flash_controller
- dtbinding_ambiq_gpio
- dtbinding_ambiq_gpio_bank
- dtbinding_ambiq_hw_crc32
- dtbinding_ambiq_i2c
- dtbinding_ambiq_i2s
- dtbinding_ambiq_iom
- dtbinding_ambiq_ios
- dtbinding_ambiq_ios_i2c
- dtbinding_ambiq_mspi
- dtbinding_ambiq_mspi_controller
- dtbinding_ambiq_mspi_device
- dtbinding_ambiq_pdm
- dtbinding_ambiq_pl011_uart
- dtbinding_ambiq_puf_trng
- dtbinding_ambiq_rtc
- dtbinding_ambiq_sdio
- dtbinding_ambiq_spi

- dtbinding_ambiq_spi_bleif
- dtbinding_ambiq_spid
- dtbinding_ambiq_stimer
- dtbinding_ambiq_timer
- dtbinding_ambiq_timer_pwm
- dtbinding_ambiq_uart
- dtbinding_ambiq_usb
- dtbinding_ambiq_watchdog

AMS AG (ams)

- dtbinding_ams_as5048
- dtbinding_ams_as5600
- dtbinding_ams_as6212
- dtbinding_ams_as6221
- dtbinding_ams_ccs811
- dtbinding_ams_ens210
- dtbinding_ams_iaqcore
- dtbinding_ams_tcs3400
- dtbinding_ams_tcs34725
- dtbinding_ams_tmd2620
- dtbinding_ams_tsl2540
- dtbinding_ams_tsl2561
- dtbinding_ams_tsl2591

Analog Devices, Inc. (adi)

- dtbinding_adi_ad2s1210
- dtbinding_adi_ad4050_adc
- dtbinding_adi_ad4052_adc
- dtbinding_adi_ad4114_adc
- dtbinding_adi_ad4130_adc
- dtbinding_adi_ad4170_adc
- dtbinding_adi_ad4190_8_adc
- dtbinding_adi_ad4190_adc
- dtbinding_adi_ad4195_8_adc
- dtbinding_adi_ad4195_adc
- dtbinding_adi_ad5529r
- dtbinding_adi_ad559x_spi
- dtbinding_adi_ad559x_i2c
- dtbinding_adi_ad559x_adc
- dtbinding_adi_ad559x_dac

- dtbinding_adi_ad559x_gpio
- dtbinding_adi_ad5601
- dtbinding_adi_ad5611
- dtbinding_adi_ad5621
- dtbinding_adi_ad5628
- dtbinding_adi_ad5648
- dtbinding_adi_ad5668
- dtbinding_adi_ad5672
- dtbinding_adi_ad5674
- dtbinding_adi_ad5676
- dtbinding_adi_ad5679
- dtbinding_adi_ad5684
- dtbinding_adi_ad5686
- dtbinding_adi_ad5687
- dtbinding_adi_ad5689
- dtbinding_adi_ad5691
- dtbinding_adi_ad5692
- dtbinding_adi_ad5693
- dtbinding_adi_ad7124_adc
- dtbinding_adi_ade7913
- dtbinding_adi_ade7978
- dtbinding_adi_adgm3121
- dtbinding_adi_adgm3121_gpio
- dtbinding_adi_adin1100_phy
- dtbinding_adi_adin1110
- dtbinding_adi_adin1140
- dtbinding_adi_adin1140_mdio
- dtbinding_adi_adin1140_phy
- dtbinding_adi_adin2111
- dtbinding_adi_adin2111_mdio
- dtbinding_adi_adin2111_phy
- dtbinding_adi_adis1647x
- dtbinding_adi_adltc2990
- dtbinding_adi_adp5360
- dtbinding_adi_adp5360_charger
- dtbinding_adi_adp5360_fuel_gauge
- dtbinding_adi_adp5360_regulator
- dtbinding_adi_adp5585
- dtbinding_adi_adp5585_gpio

- dtbinding_adi_adt7310
- dtbinding_adi_adt7410
- dtbinding_adi_adt7420
- dtbinding_adi_adt7422
- dtbinding_adi_adv7535
- dtbinding_adi_adv7535_dsi
- dtbinding_adi_adxl313
- dtbinding_adi_adxl345_spi
- dtbinding_adi_adxl345_i2c
- dtbinding_adi_adxl355_i2c
- dtbinding_adi_adxl355_spi
- dtbinding_adi_adxl362
- dtbinding_adi_adxl366_spi
- dtbinding_adi_adxl366_i2c
- dtbinding_adi_adxl367_i2c
- dtbinding_adi_adxl367_spi
- dtbinding_adi_adxl372_i2c
- dtbinding_adi_adxl372_spi
- dtbinding_adi_axi_clkgen
- dtbinding_adi_axi_dmac
- dtbinding_adi_axi_pwmgen
- dtbinding_adi_axi_sysid
- dtbinding_adi_ltc2959
- dtbinding_adi_ltc4286
- dtbinding_adi_max14906_gpio
- dtbinding_adi_max14915_gpio
- dtbinding_adi_max14916_gpio
- dtbinding_adi_max14917_gpio
- dtbinding_adi_max20362_regulator
- dtbinding_adi_max22017
- dtbinding_adi_max22017_dac
- dtbinding_adi_max22017_gpio
- dtbinding_adi_max22190_gpio
- dtbinding_adi_max22199_gpio
- dtbinding_adi_max2221x
- dtbinding_adi_max2221x_misc
- dtbinding_adi_max2221x_pwm
- dtbinding_adi_max30009_spi
- dtbinding_adi_max30009_i2c

- dtbinding_adi_max30210
- dtbinding_adi_max31331
- dtbinding_adi_max32_adc
- dtbinding_adi_max32_adc_10b
- dtbinding_adi_max32_adc_b_me18
- dtbinding_adi_max32_adc_sar
- dtbinding_adi_max32_backup_sram
- dtbinding_adi_max32_can
- dtbinding_adi_max32_counter
- dtbinding_adi_max32_dma
- dtbinding_adi_max32_erfo
- dtbinding_adi_max32_extclk
- dtbinding_adi_max32_flash_controller
- dtbinding_adi_max32_gcr
- dtbinding_adi_max32_gpio
- dtbinding_adi_max32_hpb
- dtbinding_adi_max32_i2c
- dtbinding_adi_max32_i2s
- dtbinding_adi_max32_i3c
- dtbinding_adi_max32_m4f_cpu1
- dtbinding_adi_max32_pinctrl
- dtbinding_adi_max32_pwm
- dtbinding_adi_max32_rtc_counter
- dtbinding_adi_max32_rv32
- dtbinding_adi_max32_rv32_intc
- dtbinding_adi_max32_rv32_sys_timer
- dtbinding_adi_max32_sdhc
- dtbinding_adi_max32_spi
- dtbinding_adi_max32_spixf
- dtbinding_adi_max32_spixf_nor
- dtbinding_adi_max32_timer
- dtbinding_adi_max32_trng
- dtbinding_adi_max32_uart
- dtbinding_adi_max32_usbhs
- dtbinding_adi_max32_w1
- dtbinding_adi_max32_watchdog
- dtbinding_adi_max32_wut
- dtbinding_adi_max32_wut_timer
- dtbinding_adi_max42500_watchdog

- dtbinding_adi_maxq10xx
- dtbinding_adi_maxq10xx_trng
- dtbinding_adi_mbox_max32_sema
- dtbinding_adi_sdp_120
- dtbinding_adi_tmc2209
- dtbinding_adi_tmc50xx
- dtbinding_adi_tmc50xx_stepper_ctrl
- dtbinding_adi_tmc50xx_stepper_driver
- dtbinding_adi_tmc51xx_spi
- dtbinding_adi_tmc51xx_uart
- dtbinding_adi_tmc51xx_stepper_ctrl
- dtbinding_adi_tmc51xx_stepper_driver
- dtbinding_adi_tmc6460_spi
- dtbinding_adi_tmc6460_uart
- dtbinding_adi_tmcm3216
- dtbinding_adi_tmcm3216_stepper_ctrl
- dtbinding_adi_tmcm3216_stepper_driver

Andes Technology Corporation (andestech)

- dtbinding_andestech_andescore_v5
- dtbinding_andestech_atcdmacx00
- dtbinding_andestech_atcgpio100
- dtbinding_andestech_atciic100
- dtbinding_andestech_atcpit100
- dtbinding_andestech_atcspi200
- dtbinding_andestech_atcwdt200
- dtbinding_andestech_l2c
- dtbinding_andestech_mbox_plic_sw
- dtbinding_andestech_nceplic100
- dtbinding_andestech_qspi_nor

Angst+Pfister (ap)

- dtbinding_ap_fcx_mldx5

Apa Electronic Co., Ltd (apa)

- dtbinding_apa_apa102

Aptina Imaging (aptina)

- dtbinding_aptina_mt9m114

Arducam Technology Co., Limited (arducam)

- dtbinding_arducam_dvp_20pin_connector
- dtbinding_arducam_ffc_40pin_connector
- dtbinding_arducam_mega

Arduino (arduino)

- dtbinding_arduino_modulino_buttons
- dtbinding_arduino_modulino_buttons_leds
- dtbinding_arduino_modulino_latch_relay
- dtbinding_arduino_modulino_pixels
- dtbinding_arduino_uno_adc

ARM Ltd. (arm)

- dtbinding_arm_armv6m_mpu
- dtbinding_arm_armv6m_systick
- dtbinding_arm_armv7_timer
- dtbinding_arm_armv7m_itm
- dtbinding_arm_armv7m_mpu
- dtbinding_arm_armv7m_systick
- dtbinding_arm_armv8
- dtbinding_arm_armv8_timer
- dtbinding_arm_armv8.1m_mpu
- dtbinding_arm_armv8.1m_systick
- dtbinding_arm_armv8m_itm
- dtbinding_arm_armv8m_mpu
- dtbinding_arm_armv8m_systick
- dtbinding_arm_axi_timing_adapter
- dtbinding_arm_beetle_syscon
- dtbinding_arm_cmsdk_dtimer
- dtbinding_arm_cmsdk_gpio
- dtbinding_arm_cmsdk_timer
- dtbinding_arm_cmsdk_uart
- dtbinding_arm_cmsdk_watchdog
- dtbinding_arm_cortex_a32
- dtbinding_arm_cortex_a320
- dtbinding_arm_cortex_a510
- dtbinding_arm_cortex_a53
- dtbinding_arm_cortex_a55
- dtbinding_arm_cortex_a57
- dtbinding_arm_cortex_a7

- dtbinding_arm_cortex_a72
- dtbinding_arm_cortex_a720
- dtbinding_arm_cortex_a76
- dtbinding_arm_cortex_a78
- dtbinding_arm_cortex_a9
- dtbinding_arm_cortex_m0
- dtbinding_arm_cortex_m0+
- dtbinding_arm_cortex_m1
- dtbinding_arm_cortex_m23
- dtbinding_arm_cortex_m3
- dtbinding_arm_cortex_m33
- dtbinding_arm_cortex_m33f
- dtbinding_arm_cortex_m4
- dtbinding_arm_cortex_m4f
- dtbinding_arm_cortex_m52
- dtbinding_arm_cortex_m52f
- dtbinding_arm_cortex_m55
- dtbinding_arm_cortex_m55f
- dtbinding_arm_cortex_m7
- dtbinding_arm_cortex_m85
- dtbinding_arm_cortex_m85f
- dtbinding_arm_cortex_r4
- dtbinding_arm_cortex_r4f
- dtbinding_arm_cortex_r5
- dtbinding_arm_cortex_r52
- dtbinding_arm_cortex_r5f
- dtbinding_arm_cortex_r7
- dtbinding_arm_cortex_r8
- dtbinding_arm_cortex_r82
- dtbinding_arm_cortex_r8f
- dtbinding_arm_crsas_ma2_counter
- dtbinding_arm_crsas_ma2_timer
- dtbinding_arm_crsas_ma2_watchdog
- dtbinding_arm_cryptocell_310
- dtbinding_arm_cryptocell_312
- dtbinding_arm_dma_pl330
- dtbinding_arm_dtcm
- dtbinding_arm_ethos_u
- dtbinding_arm_fvp_pwrc

- dtbinding_arm_gic
- dtbinding_arm_gic_v1
- dtbinding_arm_gic_v2
- dtbinding_arm_gic_v3
- dtbinding_arm_gic_v3_its
- dtbinding_arm_itcm
- dtbinding_arm_mhu
- dtbinding_arm_mhuv2
- dtbinding_arm_mhuv3
- dtbinding_arm_mmio32_gpio
- dtbinding_arm_mps2_pinctrl
- dtbinding_arm_mps3_pinctrl
- dtbinding_arm_mps4_pinctrl
- dtbinding_arm_pl011
- dtbinding_arm_pl022
- dtbinding_arm_psci_0.2
- dtbinding_arm_psci_1.1
- dtbinding_arm_sbsa_uart
- dtbinding_arm_scc
- dtbinding_arm_scmi
- dtbinding_arm_scmi_clock
- dtbinding_arm_scmi_pinctrl
- dtbinding_arm_scmi_power
- dtbinding_arm_scmi_power_domain
- dtbinding_arm_scmi_reset
- dtbinding_arm_scmi_shmem
- dtbinding_arm_scmi_smc
- dtbinding_arm_scmi_system
- dtbinding_arm_stmesp
- dtbinding_arm_v2m_beetle_pinctrl
- dtbinding_arm_v2m_musca_b1_pinctrl
- dtbinding_arm_v6m_nvic
- dtbinding_arm_v7m_nvic
- dtbinding_arm_v8.1m_nvic
- dtbinding_arm_v8m_nvic
- dtbinding_arm_versatile_i2c

Asahi Kasei Corp. (asahi-kasei)

- dtbinding_asahi_kasei_ak8975
- dtbinding_asahi_kasei_akm09918c

ASMedia Technology Inc. (asmedia)

- dtbinding_asmedia_asm2364

ASPEED Technology Inc. (aspeed)

- dtbinding_aspeed_ast10x0_clock
- dtbinding_aspeed_ast10x0_reset

Atmel Corporation (atmel)

- dtbinding_atmel_at24
- dtbinding_atmel_at24mac402
- dtbinding_atmel_at25
- dtbinding_atmel_at25xv021a
- dtbinding_atmel_at45
- dtbinding_atmel_ataes132a
- dtbinding_atmel_rf2xx
- dtbinding_atmel_sam_adc
- dtbinding_atmel_sam_afec
- dtbinding_atmel_sam_can
- dtbinding_atmel_sam_dac
- dtbinding_atmel_sam_flash
- dtbinding_atmel_sam_flash_controller
- dtbinding_atmel_sam_gmac
- dtbinding_atmel_sam_gpio
- dtbinding_atmel_sam_hsmci
- dtbinding_atmel_sam_i2c_twi
- dtbinding_atmel_sam_i2c_twihs
- dtbinding_atmel_sam_i2c_twim
- dtbinding_atmel_sam_mdio
- dtbinding_atmel_sam_pinctrl
- dtbinding_atmel_sam_pmc
- dtbinding_atmel_sam_pwm
- dtbinding_atmel_sam_rstc
- dtbinding_atmel_sam_rtc
- dtbinding_atmel_sam_smc
- dtbinding_atmel_sam_spi
- dtbinding_atmel_sam_ssc

- dtbinding_atmel_sam_supc
- dtbinding_atmel_sam_tc
- dtbinding_atmel_sam_tc_qdec
- dtbinding_atmel_sam_trng
- dtbinding_atmel_sam_uart
- dtbinding_atmel_sam_udp
- dtbinding_atmel_sam_usart
- dtbinding_atmel_sam_usbc
- dtbinding_atmel_sam_usbhs
- dtbinding_atmel_sam_watchdog
- dtbinding_atmel_sam_xdmac
- dtbinding_atmel_sam0_adc
- dtbinding_atmel_sam0_can
- dtbinding_atmel_sam0_dac
- dtbinding_atmel_sam0_dmac
- dtbinding_atmel_sam0_eic
- dtbinding_atmel_sam0_gclk
- dtbinding_atmel_sam0_gmac
- dtbinding_atmel_sam0_gpio
- dtbinding_atmel_sam0_i2c
- dtbinding_atmel_sam0_id
- dtbinding_atmel_sam0_mclk
- dtbinding_atmel_sam0_nvmctrl
- dtbinding_atmel_sam0_osc32kctrl
- dtbinding_atmel_sam0_pinctrl
- dtbinding_atmel_sam0_pinmux
- dtbinding_atmel_sam0_rtc
- dtbinding_atmel_sam0_sercom
- dtbinding_atmel_sam0_spi
- dtbinding_atmel_sam0_tc_pwm
- dtbinding_atmel_sam0_tc32
- dtbinding_atmel_sam0_tcc_pwm
- dtbinding_atmel_sam0_uart
- dtbinding_atmel_sam0_usb
- dtbinding_atmel_sam0_watchdog
- dtbinding_atmel_sam4l_flashcalw_controller
- dtbinding_atmel_sam4l_gpio
- dtbinding_atmel_sam4l_uid
- dtbinding_atmel_sam4l_watchdog

- dtbinding_atmel_samd5x_dac
- dtbinding_atmel_winc1500

Avago Technologies (avago)

- dtbinding_avago_apds9253
- dtbinding_avago_apds9306
- dtbinding_avago_apds9960

avia semiconductor (avia)

- dtbinding_avia_hx711_spi

Beijing Phosense Electronic Technology Co., Ltd. (phosense)

- dtbinding_phosense_xbr818

Bosch Sensortec GmbH (bosch)

- dtbinding_bosch_bma280
- dtbinding_bosch_bma4xx_spi
- dtbinding_bosch_bma4xx_i2c
- dtbinding_bosch_bmc150_magn
- dtbinding_bosch_bme280_spi
- dtbinding_bosch_bme280_i2c
- dtbinding_bosch_bme680_i2c
- dtbinding_bosch_bme680_spi
- dtbinding_bosch_bmg160
- dtbinding_bosch_bmi08x_accel_spi
- dtbinding_bosch_bmi08x_accel_i2c
- dtbinding_bosch_bmi08x_gyro_i2c
- dtbinding_bosch_bmi08x_gyro_spi
- dtbinding_bosch_bmi160_spi
- dtbinding_bosch_bmi160_i2c
- dtbinding_bosch_bmi270
- dtbinding_bosch_bmi270_i2c
- dtbinding_bosch_bmi270_spi
- dtbinding_bosch_bmi323_i2c
- dtbinding_bosch_bmi323_spi
- dtbinding_bosch_bmm150_spi
- dtbinding_bosch_bmm150_i2c
- dtbinding_bosch_bmm350_i3c
- dtbinding_bosch_bmm350_i2c
- dtbinding_bosch_bmp180

- dtbinding_bosch_bmp388_i2c
- dtbinding_bosch_bmp388_spi
- dtbinding_bosch_bmp390_spi
- dtbinding_bosch_bmp390_i2c
- dtbinding_bosch_bmp581_i2c
- dtbinding_bosch_bmp581_spi
- dtbinding_bosch_bmp581_i3c

Bouffalo Lab (Nanjing) Co., Ltd. (bflb)

- dtbinding_bflb_adc
- dtbinding_bflb_aon_regulator
- dtbinding_bflb_bclk
- dtbinding_bflb_bl60x_clock_controller
- dtbinding_bflb_bl60x_70x_gpio
- dtbinding_bflb_bl616cl_clock_controller
- dtbinding_bflb_bl61x_can
- dtbinding_bflb_bl61x_clock_controller
- dtbinding_bflb_bl61x_psram
- dtbinding_bflb_bl61x_808_gpio
- dtbinding_bflb_bl70x_clock_controller
- dtbinding_bflb_bl70x_l_clock_controller
- dtbinding_bflb_bl70x_l_dll
- dtbinding_bflb_bl808_clock_controller
- dtbinding_bflb_bl808_psram_uhs
- dtbinding_bflb_bl808_psram_uhs_controller
- dtbinding_bflb_bt_hci
- dtbinding_bflb_clock_controller
- dtbinding_bflb_dbi
- dtbinding_bflb_dma
- dtbinding_bflb_efuse
- dtbinding_bflb_f32k
- dtbinding_bflb_flash_clk
- dtbinding_bflb_i2c
- dtbinding_bflb_ieee802154
- dtbinding_bflb_irx
- dtbinding_bflb_l1c
- dtbinding_bflb_mm_clk
- dtbinding_bflb_pinctrl
- dtbinding_bflb_pll

- dtbinding_bflb_power_controller
- dtbinding_bflb_pwm_1
- dtbinding_bflb_pwm_2
- dtbinding_bflb_root_clk
- dtbinding_bflb_rt_regulator
- dtbinding_bflb_rtc
- dtbinding_bflb_sdhc
- dtbinding_bflb_sec_eng_aes
- dtbinding_bflb_sec_eng_sha
- dtbinding_bflb_sec_eng_trng
- dtbinding_bflb_sf_bank
- dtbinding_bflb_sf_controller
- dtbinding_bflb_sf_device
- dtbinding_bflb_sf_flash
- dtbinding_bflb_soc_regulator
- dtbinding_bflb_spi
- dtbinding_bflb_timer
- dtbinding_bflb_timer_channel
- dtbinding_bflb_tsen
- dtbinding_bflb_uart
- dtbinding_bflb_udc_1
- dtbinding_bflb_wdt
- dtbinding_bflb_wifi4
- dtbinding_bflb_wifi6

Broadcom Corporation (brcm)

- dtbinding_brcm_afbr_s50
- dtbinding_brcm_bcm2711_gpio
- dtbinding_brcm_bcm2711_i2c
- dtbinding_brcm_bcm2711_mbox
- dtbinding_brcm_bcm2711_pinctrl
- dtbinding_brcm_bcm2712_pinctrl
- dtbinding_brcm_bcm2835_armctrl_ic
- dtbinding_brcm_bcm2835_rng
- dtbinding_brcm_bcm2835_usb
- dtbinding_brcm_bcm2836_l1_intc
- dtbinding_brcm_bcm283x_aux_uart
- dtbinding_brcm_brcmstb_gpio
- dtbinding_brcm_brcmstb_pcie

- dtbinding_brcm_genet
- dtbinding_brcm_genet_mdio
- dtbinding_brcm_iproc_gpio
- dtbinding_brcm_iproc_i2c
- dtbinding_brcm_iproc_pax_dma_v1
- dtbinding_brcm_iproc_pax_dma_v2
- dtbinding_brcm_iproc_pcie_ep
- dtbinding_brcm_iproc_pcie_ep_v2
- dtbinding_brcm_iproc_rng200

Cadence Design Systems Inc. (cdns)

- dtbinding_cdns_i2c
- dtbinding_cdns_i3c
- dtbinding_cdns_nand
- dtbinding_cdns_qspi_nor
- dtbinding_cdns_sdhc
- dtbinding_cdns_spi
- dtbinding_cdns_swerv_s400
- dtbinding_cdns_swerv_s420
- dtbinding_cdns_swerv_pic
- dtbinding_cdns_tensilica_xtensa_lx3
- dtbinding_cdns_tensilica_xtensa_lx4
- dtbinding_cdns_tensilica_xtensa_lx6
- dtbinding_cdns_tensilica_xtensa_lx7
- dtbinding_cdns_uart
- dtbinding_cdns_xspi_nor
- dtbinding_cdns_xtensa_core_intc

Centre for Development of Advanced Computing (C-DAC) (cdac)

- dtbinding_cdac_thejas32_gpio
- dtbinding_cdac_thejas32_intc

CFSensor Co., Ltd (cfsensor)

- dtbinding_cfsensor_xgzp68xxd

Chip Wealth Technology Ltd. (chipwealth)

- dtbinding_chipwealth_ch1115_spi
- dtbinding_chipwealth_ch1115_i2c
- dtbinding_chipwealth_ch1116_spi
- dtbinding_chipwealth_ch1116_i2c

ChipOne (chipone)

- dtbinding_chipone_co5300_mipi_dsi
- dtbinding_chipone_co5300_mspi
- dtbinding_chipone_co5300_spi

Chipsemi Corp. (chipsemi)

- dtbinding_chipsemi_chsc5x
- dtbinding_chipsemi_chsc6540
- dtbinding_chipsemi_chsc6x

Cirque Corporation (cirque)

- dtbinding_cirque_pinnacle_i2c
- dtbinding_cirque_pinnacle_spi

Cirrus Logic, Inc. (cirrus)

- dtbinding_cirrus_cp9314
- dtbinding_cirrus_cs35l56_i2c
- dtbinding_cirrus_cs35l56_spi
- dtbinding_cirrus_cs35l57_i2c
- dtbinding_cirrus_cs35l57_spi
- dtbinding_cirrus_cs40l26_spi
- dtbinding_cirrus_cs40l26_i2c
- dtbinding_cirrus_cs40l27_i2c
- dtbinding_cirrus_cs40l27_spi
- dtbinding_cirrus_cs40l50_i2c
- dtbinding_cirrus_cs40l50_spi
- dtbinding_cirrus_cs40l51_spi
- dtbinding_cirrus_cs40l51_i2c
- dtbinding_cirrus_cs40l52_i2c
- dtbinding_cirrus_cs40l52_spi
- dtbinding_cirrus_cs40l53_i2c
- dtbinding_cirrus_cs40l53_spi
- dtbinding_cirrus_cs43l22
- dtbinding_cirrus_cs47l63

Cirrus Logic, Inc. (formerly Wolfson Microelectronics plc) (wolfson)

- dtbinding_wolfson_wm8904
- dtbinding_wolfson_wm8960
- dtbinding_wolfson_wm8962

Cypress Semiconductor Corporation (cypress)

- dtbinding_cypress_cy8c95xx_gpio
- dtbinding_cypress_cy8c95xx_gpio_port
- dtbinding_cypress_cy8cmbr3xxx
- dtbinding_cypress_psoc6_flash_controller
- dtbinding_cypress_psoc6_gpio
- dtbinding_cypress_psoc6_hsiom
- dtbinding_cypress_psoc6_intmux
- dtbinding_cypress_psoc6_intmux_ch
- dtbinding_cypress_psoc6_spi
- dtbinding_cypress_psoc6_uart
- dtbinding_cypress_psoc6_uid

DAVICOM Semiconductor, Inc. (davicom)

- dtbinding_davicom_dm8806_phy
- dtbinding_davicom_dm9051

DFRobot (dfrobot)

- dtbinding_dfrobot_a01nyub

Dialog Semiconductor (dlg)

- dtbinding_dlg_da7212

Digilent, Inc. (digilent)

- dtbinding_digilent_pmod

Diodes Incorporated (diodes)

- dtbinding_diodes_pi3usb9201
- dtbinding_diodes_pi4ioe5v6408

E Ink Corporation (eink)

- dtbinding_eink_ac057tc1
- dtbinding_eink_ed2208_doa
- dtbinding_eink_ed2208_gca

Efinix Inc (efinix)

- dtbinding_efinix_sapphire_gpio
- dtbinding_efinix_sapphire_timer0
- dtbinding_efinix_sapphire_uart0
- dtbinding_efinix_vexriscv_sapphire

Egis Technology Inc (egis)

- dtbinding_egis_et171_spi

Elan Microelectronic Corp. (elan)

- dtbinding_elan_em32_ahb
- dtbinding_elan_em32_apb
- dtbinding_elan_em32_gpio
- dtbinding_elan_em32_pinctrl
- dtbinding_elan_em32_pwm
- dtbinding_elan_em32_uart

ENE Technology, Inc. (ene)

- dtbinding_ene_kb106x_adc
- dtbinding_ene_kb106x_gcfg
- dtbinding_ene_kb106x_gpio
- dtbinding_ene_kb106x_i2c
- dtbinding_ene_kb106x_pinctrl
- dtbinding_ene_kb106x_pwm
- dtbinding_ene_kb106x_tach
- dtbinding_ene_kb106x_uart
- dtbinding_ene_kb106x_watchdog
- dtbinding_ene_kb1200_adc
- dtbinding_ene_kb1200_gcfg
- dtbinding_ene_kb1200_gpio
- dtbinding_ene_kb1200_i2c
- dtbinding_ene_kb1200_pinctrl
- dtbinding_ene_kb1200_pmu
- dtbinding_ene_kb1200_pwm
- dtbinding_ene_kb1200_tach
- dtbinding_ene_kb1200_uart
- dtbinding_ene_kb1200_watchdog

EPCOS AG (epcos)

- dtbinding_epcos_b57861s0103a039

Espressif Systems (espressif)

- dtbinding_espressif_esp_at
- dtbinding_espressif_esp_dsi_display
- dtbinding_espressif_esp_hosted
- dtbinding_espressif_esp_hosted_mcu_sd

- dtbinding_espressif_esp_hosted_mcu
- dtbinding_espressif_esp_hosted_mcu_bt_hci
- dtbinding_espressif_esp_hosted_mcu_wifi
- dtbinding_espressif_esp_mipi_dsi
- dtbinding_espressif_esp32_adc
- dtbinding_espressif_esp32_aes
- dtbinding_espressif_esp32_ana_cmpr
- dtbinding_espressif_esp32_bt_hci
- dtbinding_espressif_esp32_clock
- dtbinding_espressif_esp32_counter
- dtbinding_espressif_esp32_dac
- dtbinding_espressif_esp32_eth
- dtbinding_espressif_esp32_flash_controller
- dtbinding_espressif_esp32_gdma
- dtbinding_espressif_esp32_gpio
- dtbinding_espressif_esp32_i2c
- dtbinding_espressif_esp32_i2s
- dtbinding_espressif_esp32_ieee802154
- dtbinding_espressif_esp32_intc
- dtbinding_espressif_esp32_ipm
- dtbinding_espressif_esp32_lcd_cam
- dtbinding_espressif_esp32_lcd_cam_dvp
- dtbinding_espressif_esp32_lcd_cam_mipi_dbi
- dtbinding_espressif_esp32_ledc
- dtbinding_espressif_esp32_lpgpio
- dtbinding_espressif_esp32_lpuart
- dtbinding_espressif_esp32_mcpwm
- dtbinding_espressif_esp32_pcnt
- dtbinding_espressif_esp32_pinctrl
- dtbinding_espressif_esp32_ppa
- dtbinding_espressif_esp32_psram
- dtbinding_espressif_esp32_regulator
- dtbinding_espressif_esp32_rmt
- dtbinding_espressif_esp32_rtc_timer
- dtbinding_espressif_esp32_sdhc
- dtbinding_espressif_esp32_sdhc_slot
- dtbinding_espressif_esp32_sha
- dtbinding_espressif_esp32_spi
- dtbinding_espressif_esp32_systimer

- dtbinding_espressif_esp32_temp
- dtbinding_espressif_esp32_timer
- dtbinding_espressif_esp32_touch
- dtbinding_espressif_esp32_trng
- dtbinding_espressif_esp32_twai
- dtbinding_espressif_esp32_twaifd
- dtbinding_espressif_esp32_uart
- dtbinding_espressif_esp32_usb_otg_fs
- dtbinding_espressif_esp32_usb_otg_hs
- dtbinding_espressif_esp32_usb_serial
- dtbinding_espressif_esp32_watchdog
- dtbinding_espressif_esp32_wifi
- dtbinding_espressif_esp32_xt_wdt
- dtbinding_espressif_mbox_esp32
- dtbinding_espressif_riscv
- dtbinding_espressif_xtensa_lx6
- dtbinding_espressif_xtensa_lx7

Everlight Electronics Co. Ltd. (everlight)

- dtbinding_everlight_als_pt19

Fairchild Semiconductor (fcs)

- dtbinding_fcs_fxl6408

Feature Integration Technology Inc. (fintek)

- dtbinding_fintek_f75303

Festo SE & Co. KG (festo)

- dtbinding_festo_veaa_x_3

Fibocom Wireless Inc. (fibocom)

- dtbinding_fibocom_le250

Fitipower Integrated Technology Inc. (fitipower)

- dtbinding_fitipower_ek79007

FoBE Studio (fobe)

- dtbinding_fobe_quill_header

FocalTech Systems Co.,Ltd (focaltech)

- dtbinding_focaltech_ft5336
- dtbinding_focaltech_ft6146
- dtbinding_focaltech_ft9001_cpm
- dtbinding_focaltech_ft9001_cpm_rctl
- dtbinding_focaltech_ft9001_usart

Freescale Semiconductor (fsl)

- dtbinding_fsl_imx21_i2c
- dtbinding_fsl_imx27_pwm

Fujitsu Ltd. (fujitsu)

- dtbinding_fujitsu_mb85rcxx
- dtbinding_fujitsu_mb85rsxx

Futaba Corporation (futaba)

- dtbinding_futaba_sbus

Future Technology Devices International Ltd. (ftdi)

- dtbinding_ftdi_ft800

Fuzhou Rockchip Electronics Co., Ltd (rockchip)

- dtbinding_rockchip_rk3588_cru
- dtbinding_rockchip_rk3588_cru_clk
- dtbinding_rockchip_rk3588_cru_rst
- dtbinding_rockchip_rk3588_gmac

Gaisler (gaisler)

- dtbinding_gaisler_apbuart
- dtbinding_gaisler_gptimer
- dtbinding_gaisler_grgpio
- dtbinding_gaisler_irqmp
- dtbinding_gaisler_leon3
- dtbinding_gaisler_spimctrl

Galaxycore, Inc. (galaxycore)

- dtbinding_galaxycore_gc2145
- dtbinding_galaxycore_gc9x01x

Gas Sensing Solutions Ltd. (gss)

- dtbinding_gss_explorir_m

GigaDevice Semiconductor (gd)

- dtbinding_gd_gd32_adc
- dtbinding_gd_gd32_afio
- dtbinding_gd_gd32_backup_sram
- dtbinding_gd_gd32_bldo
- dtbinding_gd_gd32_cctl
- dtbinding_gd_gd32_dac
- dtbinding_gd_gd32_dma
- dtbinding_gd_gd32_dma_v1
- dtbinding_gd_gd32_exti
- dtbinding_gd_gd32_flash_controller
- dtbinding_gd_gd32_fwdgt
- dtbinding_gd_gd32_gpio
- dtbinding_gd_gd32_i2c
- dtbinding_gd_gd32_nv_flash_v1
- dtbinding_gd_gd32_nv_flash_v2
- dtbinding_gd_gd32_nv_flash_v3
- dtbinding_gd_gd32_pinctrl_af
- dtbinding_gd_gd32_pinctrl_afio
- dtbinding_gd_gd32_pwm
- dtbinding_gd_gd32_rctl
- dtbinding_gd_gd32_rcu
- dtbinding_gd_gd32_spi
- dtbinding_gd_gd32_syscfg
- dtbinding_gd_gd32_timer
- dtbinding_gd_gd32_trng
- dtbinding_gd_gd32_usart
- dtbinding_gd_gd32_wwdgt

GlobalTop Technology, Inc. (globaltop)

- dtbinding_globaltop_pa6h

GreeLed Electronic Ltd. (greeled)

- dtbinding_greeled_lpd8803
- dtbinding_greeled_lpd8806

Guangzhou Aosong Electronic Co., Ltd. (aosong)

- dtbinding_aosong_ags10
- dtbinding_aosong_aht20
- dtbinding_aosong_am2301b
- dtbinding_aosong_dht
- dtbinding_aosong_dht20

Hamamatsu Photonics K.K. (hamamatsu)

- dtbinding_hamamatsu_s11059
- dtbinding_hamamatsu_s9706

Hangzhou Grow Technology Co., Ltd. (hzgrow)

- dtbinding_hzgrow_r502a

Hangzhou Zhian Technology Co., Ltd. (zhiantec)

- dtbinding_zhiantec_zfm_x0

Hazard3 by Luke Wren (hazard3)

- dtbinding_hazard3_hazard3_intc

Himax Technologies, Inc. (himax)

- dtbinding_himax_hm01b0
- dtbinding_himax_hm0360
- dtbinding_himax_hx8353e
- dtbinding_himax_hx8379c
- dtbinding_himax_hx8394

Hitachi Ltd. (hit)

- dtbinding_hit_hd44780

Holtek Semiconductor, Inc. (holtek)

- dtbinding_holtek_ht16k33

Honeywell (honeywell)

- dtbinding_honeywell_hmc5883l
- dtbinding_honeywell_mpr
- dtbinding_honeywell_sm351lt

HOPERF Microelectronics Co. Ltd (hoperf)

- dtbinding_hoperf_hp206c
- dtbinding_hoperf_th02

Hycon Technology Corp. (hycon)

- dtbinding_hycon_hy4245

Hynitron (hynitron)

- dtbinding_hynitron_cst8xx

Iclegend Microelectronics Co., Ltd. (iclegend)

- dtbinding_iclegend_s3km1110

ILI Technology Corporation (ILITEK) (ilitek)

- dtbinding_ilitek_ili2132a
- dtbinding_ilitek_ili9163c
- dtbinding_ilitek_ili9340
- dtbinding_ilitek_ili9341
- dtbinding_ilitek_ili9342c
- dtbinding_ilitek_ili9488
- dtbinding_ilitek_ili9806e
- dtbinding_ilitek_ili9881c

Imagination Technologies Ltd. (formerly MIPS Technologies Inc.) (mti)

- dtbinding_mti_cpu_intc

Infineon Technologies (infineon)

- dtbinding_infineon_adc
- dtbinding_infineon_airoc_wifi
- dtbinding_infineon_airoc_wifi_spi
- dtbinding_infineon_asclin_uart
- dtbinding_infineon_aurix_ir
- dtbinding_infineon_aurix_stm
- dtbinding_infineon_autanalog
- dtbinding_infineon_autanalog_ac_state
- dtbinding_infineon_autanalog_ctb
- dtbinding_infineon_autanalog_ctb_opamp
- dtbinding_infineon_autanalog_ctdac
- dtbinding_infineon_autanalog_prb
- dtbinding_infineon_autanalog_prb_vref
- dtbinding_infineon_autanalog_ptcomp
- dtbinding_infineon_autanalog_ptcomp_comp
- dtbinding_infineon_autanalog_sar_adc
- dtbinding_infineon_autanalog_sar_fifo
- dtbinding_infineon_autanalog_sar_fir

- dtbinding_infineon_bless_hci
- dtbinding_infineon_bt_hci_uart
- dtbinding_infineon_can
- dtbinding_infineon_canfd_controller
- dtbinding_infineon_cat1_lp_timer_pdl
- dtbinding_infineon_cat1b_power
- dtbinding_infineon_cat1c_clk_hf
- dtbinding_infineon_clk_osc
- dtbinding_infineon_clk_pll
- dtbinding_infineon_counter
- dtbinding_infineon_cyw208xx_hci
- dtbinding_infineon_cyw43_gpio
- dtbinding_infineon_dma
- dtbinding_infineon_dmac
- dtbinding_infineon_dps310
- dtbinding_infineon_edge_npu
- dtbinding_infineon_fixed_clock
- dtbinding_infineon_fixed_factor_clock
- dtbinding_infineon_flash_controller
- dtbinding_infineon_fm25xxx
- dtbinding_infineon_gpio
- dtbinding_infineon_hibernate_wakeup
- dtbinding_infineon_hppass_ac_state
- dtbinding_infineon_hppass_analog
- dtbinding_infineon_hppass_csg
- dtbinding_infineon_hppass_csg_comp
- dtbinding_infineon_hppass_csg_dac
- dtbinding_infineon_hppass_sar_adc
- dtbinding_infineon_i2c
- dtbinding_infineon_i2s
- dtbinding_infineon_ipc
- dtbinding_infineon_lp_comp
- dtbinding_infineon_lp_comp_channel
- dtbinding_infineon_lp_timer
- dtbinding_infineon_mbox
- dtbinding_infineon_mdma
- dtbinding_infineon_mxcrypto
- dtbinding_infineon_mxcrypto_crypto
- dtbinding_infineon_mxcrypto_trng

- dtbinding_infineon_mxcryptolite
- dtbinding_infineon_mxcryptolite_crypto
- dtbinding_infineon_mxcryptolite_trng
- dtbinding_infineon_pdm
- dtbinding_infineon_peri_clk
- dtbinding_infineon_peri_div
- dtbinding_infineon_pinctrl
- dtbinding_infineon_qspi_flash
- dtbinding_infineon_rram_controller
- dtbinding_infineon_rtc
- dtbinding_infineon_s28hx512t_mspi
- dtbinding_infineon_sar_adc
- dtbinding_infineon_scb
- dtbinding_infineon_sdhc_sdio
- dtbinding_infineon_shared_gpio
- dtbinding_infineon_spi
- dtbinding_infineon_sysintc
- dtbinding_infineon_sysintc_mux
- dtbinding_infineon_tcpwm
- dtbinding_infineon_tcpwm_counter
- dtbinding_infineon_tcpwm_pwm
- dtbinding_infineon_tle9104
- dtbinding_infineon_tle9104_diagnostics
- dtbinding_infineon_tle9104_gpio
- dtbinding_infineon_tricore
- dtbinding_infineon_uart
- dtbinding_infineon_usbhs
- dtbinding_infineon_watchdog
- dtbinding_infineon_xmc4xxx_adc
- dtbinding_infineon_xmc4xxx_can
- dtbinding_infineon_xmc4xxx_can_node
- dtbinding_infineon_xmc4xxx_ccu4_pwm
- dtbinding_infineon_xmc4xxx_ccu8_pwm
- dtbinding_infineon_xmc4xxx_dma
- dtbinding_infineon_xmc4xxx_ethernet
- dtbinding_infineon_xmc4xxx_flash_controller
- dtbinding_infineon_xmc4xxx_gpio
- dtbinding_infineon_xmc4xxx_i2c
- dtbinding_infineon_xmc4xxx_intc

- dtbinding_infineon_xmc4xxx_nv_flash
- dtbinding_infineon_xmc4xxx_pinctrl
- dtbinding_infineon_xmc4xxx_rtc
- dtbinding_infineon_xmc4xxx_spi
- dtbinding_infineon_xmc4xxx_temp
- dtbinding_infineon_xmc4xxx_uart
- dtbinding_infineon_xmc4xxx_usb
- dtbinding_infineon_xmc4xxx_watchdog

Innovative Sensor Technology IST AG (ist)

- dtbinding_ist_tsic_xx6

Integrated Silicon Solutions Inc. (issi)

- dtbinding_issi_is31fl3193
- dtbinding_issi_is31fl3194
- dtbinding_issi_is31fl3197
- dtbinding_issi_is31fl3216a
- dtbinding_issi_is31fl3733

Integrated Solutions Technology Inc. (istech)

- dtbinding_istech_ist3931

Intel Corporation (intel)

- dtbinding_intel_ace_art_counter
- dtbinding_intel_ace_intc
- dtbinding_intel_ace_rtc_counter
- dtbinding_intel_ace_timestamp
- dtbinding_intel_adsp_communication_widget
- dtbinding_intel_adsp_dfpmcch
- dtbinding_intel_adsp_dfpmccu
- dtbinding_intel_adsp_dmic_vss
- dtbinding_intel_adsp_gpdma
- dtbinding_intel_adsp_hda_dmic_cap
- dtbinding_intel_adsp_hda_host_in
- dtbinding_intel_adsp_hda_host_out
- dtbinding_intel_adsp_hda_link_in
- dtbinding_intel_adsp_hda_link_out
- dtbinding_intel_adsp_hda_ssp_cap
- dtbinding_intel_adsp_host_ipc
- dtbinding_intel_adsp_idc

- dtbinding_intel_adsp_imr
- dtbinding_intel_adsp_mailbox
- dtbinding_intel_adsp_mem_window
- dtbinding_intel_adsp_mic_privacy
- dtbinding_intel_adsp_mtl_tlb
- dtbinding_intel_adsp_power_domain
- dtbinding_intel_adsp_sha
- dtbinding_intel_adsp_shim_clkctl
- dtbinding_intel_adsp_timer
- dtbinding_intel_adsp_tlb
- dtbinding_intel_adsp_uaol
- dtbinding_intel_adsp_watchdog
- dtbinding_intel_agilex_clock
- dtbinding_intel_agilex5_clock
- dtbinding_intel_alder_lake
- dtbinding_intel_alh_dai
- dtbinding_intel_apollo_lake
- dtbinding_intel_bartlett_lake
- dtbinding_intel_blinky_pwm
- dtbinding_intel_cavs_i2s
- dtbinding_intel_cavs_intc
- dtbinding_intel_dai_dmic
- dtbinding_intel_e1000
- dtbinding_intel_elkhart_lake
- dtbinding_intel_emmc_host
- dtbinding_intel_espi_peci
- dtbinding_intel_eth_plat
- dtbinding_intel_gpio
- dtbinding_intel_hda_dai
- dtbinding_intel_hpet
- dtbinding_intel_ibecc
- dtbinding_intel_igc_mac
- dtbinding_intel_igc_mdio
- dtbinding_intel_ioapic
- dtbinding_intel_ish
- dtbinding_intel_lakemont
- dtbinding_intel_loapic
- dtbinding_intel_lpss
- dtbinding_intel_lw_uart

- dtbinding_intel_multiboot_framebuffer
- dtbinding_intel_niosv
- dtbinding_intel_nova_lake
- dtbinding_intel_panther_lake
- dtbinding_intel_pch_smbus
- dtbinding_intel_penwell_spi
- dtbinding_intel_pflash_cfi01
- dtbinding_intel_raptor_lake
- dtbinding_intel_sedi_dma
- dtbinding_intel_sedi_gpio
- dtbinding_intel_sedi_i2c
- dtbinding_intel_sedi_ipm
- dtbinding_intel_sedi_spi
- dtbinding_intel_sedi_uart
- dtbinding_intel_socfpga_agilex_sip_smc
- dtbinding_intel_socfpga_reset
- dtbinding_intel_ssp
- dtbinding_intel_ssp_dai
- dtbinding_intel_ssp_sspbase
- dtbinding_intel_tco_wdt
- dtbinding_intel_timeaware_gpio
- dtbinding_intel_uaol_dai
- dtbinding_intel_vt_d
- dtbinding_intel_wildcat_lake
- dtbinding_intel_x86
- dtbinding_intel_x86_64

Intersil (isil)

- dtbinding_isil_isl29035

InvenSense Inc. (invensense)

- dtbinding_invensense_icm40627_i2c
- dtbinding_invensense_icm42370p_i2c
- dtbinding_invensense_icm42370p_spi
- dtbinding_invensense_icm42605
- dtbinding_invensense_icm42670p_spi
- dtbinding_invensense_icm42670p_i2c
- dtbinding_invensense_icm42670s_spi
- dtbinding_invensense_icm42670s_i2c
- dtbinding_invensense_icm42686

- dtbinding_invensense_icm42688
- dtbinding_invensense_icm4268x
- dtbinding_invensense_icm45605_i2c
- dtbinding_invensense_icm45605_i3c
- dtbinding_invensense_icm45605_spi
- dtbinding_invensense_icm45605s_i3c
- dtbinding_invensense_icm45605s_spi
- dtbinding_invensense_icm45605s_i2c
- dtbinding_invensense_icm45686_i2c
- dtbinding_invensense_icm45686_i3c
- dtbinding_invensense_icm45686_spi
- dtbinding_invensense_icm45686s_i2c
- dtbinding_invensense_icm45686s_i3c
- dtbinding_invensense_icm45686s_spi
- dtbinding_invensense_icm45688p_i3c
- dtbinding_invensense_icm45688p_i2c
- dtbinding_invensense_icm45688p_spi
- dtbinding_invensense_icm56622_i3c
- dtbinding_invensense_icm56622_i2c
- dtbinding_invensense_icm56622_spi
- dtbinding_invensense_icm56686_spi
- dtbinding_invensense_icm56686_i3c
- dtbinding_invensense_icm56686_i2c
- dtbinding_invensense_icp101xx
- dtbinding_invensense_icp201xx
- dtbinding_invensense_icp201xx_spi
- dtbinding_invensense_icp201xx_i2c
- dtbinding_invensense_mpu6050
- dtbinding_invensense_mpu9250
- dtbinding_invensense_tad2144_i2c
- dtbinding_invensense_tad2144
- dtbinding_invensense_tad2144_spi

Inventek Systems (inventek)

- dtbinding_inventek_eswifi
- dtbinding_inventek_eswifi_uart

Isentek Inc. (isentek)

- dtbinding_isentek_ist8310

ITE Tech. Inc. (ite)

- dtbinding_ite_enhance_i2c
- dtbinding_ite_it51xxx_adc
- dtbinding_ite_it51xxx_counter
- dtbinding_ite_it51xxx_ecpm
- dtbinding_ite_it51xxx_gpio
- dtbinding_ite_it51xxx_i2c
- dtbinding_ite_it51xxx_i3cm
- dtbinding_ite_it51xxx_i3cs
- dtbinding_ite_it51xxx_intc
- dtbinding_ite_it51xxx_kbd
- dtbinding_ite_it51xxx_manual_flash_1k
- dtbinding_ite_it51xxx_ps2
- dtbinding_ite_it51xxx_pwm
- dtbinding_ite_it51xxx_sha
- dtbinding_ite_it51xxx_smbus
- dtbinding_ite_it51xxx_spi
- dtbinding_ite_it51xxx_tach
- dtbinding_ite_it51xxx_timer
- dtbinding_ite_it51xxx_uart
- dtbinding_ite_it51xxx_usb
- dtbinding_ite_it51xxx_vcmp
- dtbinding_ite_it51xxx_watchdog
- dtbinding_ite_it51xxx_wuc
- dtbinding_ite_it51xxx_wuc_map
- dtbinding_ite_it82xx2_usb
- dtbinding_ite_it8801_altctrl
- dtbinding_ite_it8801_gpio
- dtbinding_ite_it8801_kbd
- dtbinding_ite_it8801_mfd
- dtbinding_ite_it8801_mfd_map
- dtbinding_ite_it8801_pwm
- dtbinding_ite_it8951
- dtbinding_ite_it8xxx2_adc
- dtbinding_ite_it8xxx2_bbram
- dtbinding_ite_it8xxx2_counter
- dtbinding_ite_it8xxx2_espi
- dtbinding_ite_it8xxx2_flash_controller
- dtbinding_ite_it8xxx2_gpio

- dtbinding_ite_it8xxx2_gpio_v2
- dtbinding_ite_it8xxx2_gpiokscan
- dtbinding_ite_it8xxx2_i2c
- dtbinding_ite_it8xxx2_ilm
- dtbinding_ite_it8xxx2_intc
- dtbinding_ite_it8xxx2_intc_v2
- dtbinding_ite_it8xxx2_kbd
- dtbinding_ite_it8xxx2_peci
- dtbinding_ite_it8xxx2_pinctrl
- dtbinding_ite_it8xxx2_pinctrl_func
- dtbinding_ite_it8xxx2_power_elpm
- dtbinding_ite_it8xxx2_pwm
- dtbinding_ite_it8xxx2_pwmprs
- dtbinding_ite_it8xxx2_rtc
- dtbinding_ite_it8xxx2_sha
- dtbinding_ite_it8xxx2_sha_v2
- dtbinding_ite_it8xxx2_shi
- dtbinding_ite_it8xxx2_spi
- dtbinding_ite_it8xxx2_tach
- dtbinding_ite_it8xxx2_timer
- dtbinding_ite_it8xxx2_uart
- dtbinding_ite_it8xxx2_usbpd
- dtbinding_ite_it8xxx2_vcmp
- dtbinding_ite_it8xxx2_watchdog
- dtbinding_ite_it8xxx2_wuc
- dtbinding_ite_it8xxx2_wuc_map
- dtbinding_ite_riscv_ite

Japan Display Inc. (jdi)

- dtbinding_jdi_lpm013m126

JEDEC Solid State Technology Association (jedec)

- dtbinding_jedec_jc_42.4_temp
- dtbinding_jedec_nor_mspi
- dtbinding_jedec_qspi_nor
- dtbinding_jedec_spi_nand
- dtbinding_jedec_spi_nor

Koninklijke Philips N.V. (philips)

- dtbinding_philips_pcd8544

Kvaser (kvaser)

- dtbinding_kvaser_pcican

Lattice Semiconductor (lattice)

- dtbinding_lattice_ice40_fpga
- dtbinding_lattice_ice40_fpga_base
- dtbinding_lattice_ice40_fpga_bitbang

LeveTop Semiconductor (levetop)

- dtbinding_levetop_lt7680

Linaro Limited (linaro)

- dtbinding_linaro_96b_lscon_1v8
- dtbinding_linaro_96b_lscon_3v3
- dtbinding_linaro_ivshmem_ipm
- dtbinding_linaro_ivshmem_mbox
- dtbinding_linaro_optee_tz

Linear Technology Corporation (lltc)

- dtbinding_lltc_ltc1660
- dtbinding_lltc_ltc1665
- dtbinding_lltc_ltc2451

LiteOn OptoElectronics (liteon)

- dtbinding_liteon_ltr329
- dtbinding_liteon_ltr553
- dtbinding_liteon_ltrf216a

LiteX SoC builder (litex)

- dtbinding_litex_clk
- dtbinding_litex_clkout
- dtbinding_litex_dna0
- dtbinding_litex_gpio
- dtbinding_litex_i2c
- dtbinding_litex_i2s
- dtbinding_litex_liteeth
- dtbinding_litex_liteeth_mdio
- dtbinding_litex_litei2c
- dtbinding_litex_mmc
- dtbinding_litex_prbs
- dtbinding_litex_pwm

- dtbinding_litex_soc_controller
- dtbinding_litex_spi
- dtbinding_litex_spi_litespi
- dtbinding_litex_timer0
- dtbinding_litex_uart
- dtbinding_litex_vexiiriscv_standard
- dtbinding_litex_vexriscv_intc0
- dtbinding_litex_vexriscv_standard
- dtbinding_litex_watchdog

lowRISC Community Interest Company (lowrisc)

- dtbinding_lowrisc_ibex
- dtbinding_lowrisc_opentitan_aontimer
- dtbinding_lowrisc_opentitan_pwrmgr
- dtbinding_lowrisc_opentitan_spi
- dtbinding_lowrisc_opentitan_uart

LuatOS Team (luatos)

- dtbinding_luatos_air530z

M5Stack (m5stack)

- dtbinding_m5stack_atom_header
- dtbinding_m5stack_m5pm1
- dtbinding_m5stack_m5pm1_adc
- dtbinding_m5stack_m5pm1_gpio
- dtbinding_m5stack_m5pm1_regulator
- dtbinding_m5stack_m5pm1_wdt
- dtbinding_m5stack_mbus_header
- dtbinding_m5stack_stamps3_header

Macronix International Co., Ltd. (mxicy)

- dtbinding_mxicy_mx25u_mspi
- dtbinding_mxicy_mx25um51345g

MaxBotix Inc. (maxbotix)

- dtbinding_maxbotix_mb7040

Maxim Integrated Products (maxim)

- dtbinding_maxim_ds1302
- dtbinding_maxim_ds1307
- dtbinding_maxim_ds1337
- dtbinding_maxim_ds18b20
- dtbinding_maxim_ds18s20
- dtbinding_maxim_ds2482_800
- dtbinding_maxim_ds2482_800_channel
- dtbinding_maxim_ds2484
- dtbinding_maxim_ds2485
- dtbinding_maxim_ds3231_mfd
- dtbinding_maxim_ds3231_rtc
- dtbinding_maxim_ds3231_sensor
- dtbinding_maxim_max11102
- dtbinding_maxim_max11103
- dtbinding_maxim_max11105
- dtbinding_maxim_max11106
- dtbinding_maxim_max11110
- dtbinding_maxim_max11111
- dtbinding_maxim_max11115
- dtbinding_maxim_max11116
- dtbinding_maxim_max11117
- dtbinding_maxim_max11253
- dtbinding_maxim_max11254
- dtbinding_maxim_max17048
- dtbinding_maxim_max17055
- dtbinding_maxim_max17262
- dtbinding_maxim_max20335
- dtbinding_maxim_max20335_charger
- dtbinding_maxim_max20335_regulator
- dtbinding_maxim_max2253x
- dtbinding_maxim_max30101
- dtbinding_maxim_max30102
- dtbinding_maxim_max31790
- dtbinding_maxim_max31790_fan_fault
- dtbinding_maxim_max31790_fan_speed
- dtbinding_maxim_max31790_pwm
- dtbinding_maxim_max31826
- dtbinding_maxim_max31855_spi

- dtbinding_maxim_max31865
- dtbinding_maxim_max31875
- dtbinding_maxim_max32664c
- dtbinding_maxim_max3421e_spi
- dtbinding_maxim_max44009
- dtbinding_maxim_max6675
- dtbinding_maxim_max7219
- dtbinding_maxim_max98091

Maxlinear, Inc. (maxlinear)

- dtbinding_maxlinear_gpy111

Measurement Specialties (meas)

- dtbinding_meas_htu21d
- dtbinding_meas_htu31d
- dtbinding_meas_ms5607_spi
- dtbinding_meas_ms5607_i2c
- dtbinding_meas_ms5637
- dtbinding_meas_ms5837_02ba
- dtbinding_meas_ms5837_30ba

MediaTek Inc. (mediatek)

- dtbinding_mediatek_adsp_intc
- dtbinding_mediatek_afe
- dtbinding_mediatek_mt8188_infracfg_ao
- dtbinding_mediatek_mt8188_pinctrl
- dtbinding_mediatek_mt8188_uart
- dtbinding_mediatek_mt818x_cpuclk
- dtbinding_mediatek_mt8195_cpuclk
- dtbinding_mediatek_ostimer64

Melexis N.V. (melexis)

- dtbinding_melexis_mlx90394

MEMSIC Inc. (memsic)

- dtbinding_memsic_mc3419
- dtbinding_memsic_mmc56x3

Micro Crystal AG (microcrystal)

- dtbinding_microcrystal_rv_8263_c8
- dtbinding_microcrystal_rv3028
- dtbinding_microcrystal_rv3028_counter
- dtbinding_microcrystal_rv3028_eeprom
- dtbinding_microcrystal_rv3028_ram
- dtbinding_microcrystal_rv3028_rtc
- dtbinding_microcrystal_rv3032
- dtbinding_microcrystal_rv3032_counter
- dtbinding_microcrystal_rv3032_mfd
- dtbinding_microcrystal_rv3032_temp
- dtbinding_microcrystal_rv8803

Micro:bit Educational Foundation (microbit)

- dtbinding_microbit_edge_connector

Microchip Technology Inc. (microchip)

- dtbinding_microchip_ac_g1_comparator
- dtbinding_microchip_adc_g1
- dtbinding_microchip_aes_g1
- dtbinding_microchip_aic_g1_intc
- dtbinding_microchip_cap12xx
- dtbinding_microchip_coreuart
- dtbinding_microchip_dac_g1
- dtbinding_microchip_dac_g2
- dtbinding_microchip_dbgu_g1_uart
- dtbinding_microchip_dmac_g1_dma
- dtbinding_microchip_dmac_g2_dma
- dtbinding_microchip_dmac_g3_dma
- dtbinding_microchip_dmec_ecia_girq
- dtbinding_microchip_eic_g1_intc
- dtbinding_microchip_enc28j60
- dtbinding_microchip_enc424j600
- dtbinding_microchip_flash_g2
- dtbinding_microchip_flash_g3
- dtbinding_microchip_flexcom_g1_spi
- dtbinding_microchip_gmac_g1_eth
- dtbinding_microchip_gmac_g1_mdio
- dtbinding_microchip_hwinfo_g1
- dtbinding_microchip_igloo2_envm_controller

- dtbinding_microchip_ksz8081
- dtbinding_microchip_ksz9131
- dtbinding_microchip_lan865x
- dtbinding_microchip_lan865x_mdio
- dtbinding_microchip_lan8742
- dtbinding_microchip_lan8840
- dtbinding_microchip_lan9250
- dtbinding_microchip_mcp23008
- dtbinding_microchip_mcp23009
- dtbinding_microchip_mcp23016
- dtbinding_microchip_mcp23017
- dtbinding_microchip_mcp23018
- dtbinding_microchip_mcp23s08
- dtbinding_microchip_mcp23s09
- dtbinding_microchip_mcp23s17
- dtbinding_microchip_mcp23s18
- dtbinding_microchip_mcp2515
- dtbinding_microchip_mcp251xfd
- dtbinding_microchip_mcp3204
- dtbinding_microchip_mcp3208
- dtbinding_microchip_mcp3221
- dtbinding_microchip_mcp356xr
- dtbinding_microchip_mcp4725
- dtbinding_microchip_mcp4728
- dtbinding_microchip_mcp7940n
- dtbinding_microchip_mcp9600
- dtbinding_microchip_mcp970x
- dtbinding_microchip_mec5_gpio
- dtbinding_microchip_mpfs_gpio
- dtbinding_microchip_mpfs_i2c
- dtbinding_microchip_mpfs_qspi
- dtbinding_microchip_mpfs_reset
- dtbinding_microchip_mpfs_spi
- dtbinding_microchip_mtch9010
- dtbinding_microchip_nvmctrl_g1_flash
- dtbinding_microchip_nvmctrl_g2
- dtbinding_microchip_nvmctrl_g3
- dtbinding_microchip_pac194x
- dtbinding_microchip_pic32ck_sg_gc_clock

- dtbinding_microchip_pic32ck_sg_gc_dfll48m
- dtbinding_microchip_pic32ck_sg_gc_gclkgen
- dtbinding_microchip_pic32ck_sg_gc_gclkperiph
- dtbinding_microchip_pic32ck_sg_gc_mclkcpu
- dtbinding_microchip_pic32ck_sg_gc_mclkperiph
- dtbinding_microchip_pic32ck_sg_gc_pll
- dtbinding_microchip_pic32ck_sg_gc_rtc
- dtbinding_microchip_pic32ck_sg_gc_xosc
- dtbinding_microchip_pic32ck_sg_gc_xosc32k
- dtbinding_microchip_pic32cm_jh_clock
- dtbinding_microchip_pic32cm_jh_fdpll
- dtbinding_microchip_pic32cm_jh_gclkgen
- dtbinding_microchip_pic32cm_jh_gclkperiph
- dtbinding_microchip_pic32cm_jh_mclkcpu
- dtbinding_microchip_pic32cm_jh_mclkperiph
- dtbinding_microchip_pic32cm_jh_osc32k
- dtbinding_microchip_pic32cm_jh_osc48m
- dtbinding_microchip_pic32cm_jh_rtc
- dtbinding_microchip_pic32cm_jh_xosc
- dtbinding_microchip_pic32cm_jh_xosc32k
- dtbinding_microchip_pic32cm_pl_clock
- dtbinding_microchip_pic32cm_pl_gclkgen
- dtbinding_microchip_pic32cm_pl_gclkperiph
- dtbinding_microchip_pic32cm_pl_mclkcpu
- dtbinding_microchip_pic32cm_pl_mclkperiph
- dtbinding_microchip_pic32cm_pl_osc32k
- dtbinding_microchip_pic32cm_pl_oschf
- dtbinding_microchip_pic32cm_pl_rtc
- dtbinding_microchip_pic32cm_pl_xosc32k
- dtbinding_microchip_pic32cm_sg_gc_clock
- dtbinding_microchip_pic32cm_sg_gc_dfll48m
- dtbinding_microchip_pic32cm_sg_gc_dpll
- dtbinding_microchip_pic32cm_sg_gc_gclkgen
- dtbinding_microchip_pic32cm_sg_gc_gclkperiph
- dtbinding_microchip_pic32cm_sg_gc_mclkdomain
- dtbinding_microchip_pic32cm_sg_gc_mclkperiph
- dtbinding_microchip_pic32cm_sg_gc_rtc
- dtbinding_microchip_pic32cm_sg_gc_xosc
- dtbinding_microchip_pic32cm_sg_gc_xosc32k

- dtbinding_microchip_pic32cz_ca_clock
- dtbinding_microchip_pic32cz_ca_dfll48m
- dtbinding_microchip_pic32cz_ca_dpll
- dtbinding_microchip_pic32cz_ca_gclkgen
- dtbinding_microchip_pic32cz_ca_gclkperiph
- dtbinding_microchip_pic32cz_ca_mclkdomain
- dtbinding_microchip_pic32cz_ca_mclkperiph
- dtbinding_microchip_pic32cz_ca_rtc
- dtbinding_microchip_pic32cz_ca_xosc
- dtbinding_microchip_pic32cz_ca_xosc32k
- dtbinding_microchip_pit_g1_timer
- dtbinding_microchip_port_g1_gpio
- dtbinding_microchip_port_g1_pinctrl
- dtbinding_microchip_pwm_g1
- dtbinding_microchip_rstc_g1_reset
- dtbinding_microchip_rtc_g1
- dtbinding_microchip_rtc_g1_counter
- dtbinding_microchip_rtc_g2
- dtbinding_microchip_rtc_mss
- dtbinding_microchip_sam_d5x_e5x_clock
- dtbinding_microchip_sam_d5x_e5x_dfll
- dtbinding_microchip_sam_d5x_e5x_fdpll
- dtbinding_microchip_sam_d5x_e5x_gclkgen
- dtbinding_microchip_sam_d5x_e5x_gclkperiph
- dtbinding_microchip_sam_d5x_e5x_mclkcpu
- dtbinding_microchip_sam_d5x_e5x_mclkperiph
- dtbinding_microchip_sam_d5x_e5x_rtc
- dtbinding_microchip_sam_d5x_e5x_xosc
- dtbinding_microchip_sam_d5x_e5x_xosc32k
- dtbinding_microchip_sam_ethernet_controller
- dtbinding_microchip_sam_flexcom
- dtbinding_microchip_sam_pio4
- dtbinding_microchip_sam_pit64b
- dtbinding_microchip_sam_pit64b_counter
- dtbinding_microchip_sam_pmc
- dtbinding_microchip_sama7g5_pinctrl
- dtbinding_microchip_sama7g5_sckc
- dtbinding_microchip_sama7g5_sdmmc
- dtbinding_microchip_sdhc_g1

- dtbinding_microchip_sercom_g1
- dtbinding_microchip_sercom_g1_i2c
- dtbinding_microchip_sercom_g1_spi
- dtbinding_microchip_sercom_g1_uart
- dtbinding_microchip_sha_g1_crypto
- dtbinding_microchip_smartfusion2_clock
- dtbinding_microchip_smartfusion2_flash_controller
- dtbinding_microchip_smartfusion2_h2f_irqctrl
- dtbinding_microchip_supc_g1
- dtbinding_microchip_t1s_phy
- dtbinding_microchip_tc_g1
- dtbinding_microchip_tc_g1_counter
- dtbinding_microchip_tc_g1_pwm
- dtbinding_microchip_tc_g2_counter
- dtbinding_microchip_tcc_g1
- dtbinding_microchip_tcc_g1_counter
- dtbinding_microchip_tcc_g1_pwm
- dtbinding_microchip_tcn75a
- dtbinding_microchip_trng_g1_entropy
- dtbinding_microchip_trng_g2_entropy
- dtbinding_microchip_uart_g1
- dtbinding_microchip_udphs_g1_udc
- dtbinding_microchip_vsc8541
- dtbinding_microchip_wdt_g1
- dtbinding_microchip_xec_adc
- dtbinding_microchip_xec_basic_timer
- dtbinding_microchip_xec_bbled
- dtbinding_microchip_xec_bbram
- dtbinding_microchip_xec_dmac
- dtbinding_microchip_xec_ecia
- dtbinding_microchip_xec_ecia_girq
- dtbinding_microchip_xec_ecs
- dtbinding_microchip_xec_eeprom
- dtbinding_microchip_xec_espi
- dtbinding_microchip_xec_espi_host_dev
- dtbinding_microchip_xec_espi_saf
- dtbinding_microchip_xec_espi_saf_v2
- dtbinding_microchip_xec_espi_v2
- dtbinding_microchip_xec_espi_vw_routing

- dtbinding_microchip_xec_gpio
- dtbinding_microchip_xec_gpio_v2
- dtbinding_microchip_xec_hibtimer
- dtbinding_microchip_xec_i2c
- dtbinding_microchip_xec_i2c_v2
- dtbinding_microchip_xec_i3c
- dtbinding_microchip_xec_kbd
- dtbinding_microchip_xec_pcr
- dtbinding_microchip_xec_peci
- dtbinding_microchip_xec_pinctrl
- dtbinding_microchip_xec_ps2
- dtbinding_microchip_xec_pwm
- dtbinding_microchip_xec_pwmbbled
- dtbinding_microchip_xec_qmspi
- dtbinding_microchip_xec_qmspi_controller
- dtbinding_microchip_xec_qmspi_device_mspi
- dtbinding_microchip_xec_qmspi_ldma
- dtbinding_microchip_xec_rtc
- dtbinding_microchip_xec_rtos_timer
- dtbinding_microchip_xec_symcr
- dtbinding_microchip_xec_tach
- dtbinding_microchip_xec_timer
- dtbinding_microchip_xec_uart
- dtbinding_microchip_xec_watchdog
- dtbinding_microchip_xpro_header

Micron Technology Inc. (micron)

- dtbinding_micron_mt25qu02g

Monolithic Power Systems Inc. (mps)

- dtbinding_mps_mpm54304

Motorcomm Electronic Technology Co., Ltd. (motorcomm)

- dtbinding_motorcomm_yt8521
- dtbinding_motorcomm_yt8531

Motorola, Inc. (motorola)

- dtbinding_motorola_mc146818
- dtbinding_motorola_mc146818_bbram
- dtbinding_motorola_mc146818_mfd

Murata Manufacturing Co., Ltd. (murata)

- dtbinding_murata_ncp15wb473
- dtbinding_murata_ncp15xh103

National Semiconductor (national)

- dtbinding_national_lm95234

NEORV32 RISC-V Processor (neorv32)

- dtbinding_neorv32_cpu
- dtbinding_neorv32_gpio
- dtbinding_neorv32_gptmr
- dtbinding_neorv32_pwm
- dtbinding_neorv32_trng
- dtbinding_neorv32_uart

Netsol, CO., LTD (netsol)

- dtbinding_netsol_s3axx04

Ningbo Peacefair Electronic Technology Co., Ltd (peacefair)

- dtbinding_peacefair_pzem004t

Nintendo (nintendo)

- dtbinding_nintendo_nunchuk

Nordic Semiconductor (nordic)

- dtbinding_nordic_axon
- dtbinding_nordic_coresight_nrf
- dtbinding_nordic_expansion_board_header
- dtbinding_nordic_ironside_call
- dtbinding_nordic_mbox_nrf_ipc
- dtbinding_nordic_mram
- dtbinding_nordic_npm10xx
- dtbinding_nordic_npm10xx_adc
- dtbinding_nordic_npm10xx_charger
- dtbinding_nordic_npm10xx_gpio
- dtbinding_nordic_npm10xx_led
- dtbinding_nordic_npm10xx_regulator
- dtbinding_nordic_npm10xx_uicr
- dtbinding_nordic_npm10xx_wdt
- dtbinding_nordic_npm1100
- dtbinding_nordic_npm1300

- dtbinding_nordic_npm1300_charger
- dtbinding_nordic_npm1300_gpio
- dtbinding_nordic_npm1300_led
- dtbinding_nordic_npm1300_regulator
- dtbinding_nordic_npm1300_wdt
- dtbinding_nordic_npm1304
- dtbinding_nordic_npm1304_charger
- dtbinding_nordic_npm1304_gpio
- dtbinding_nordic_npm1304_led
- dtbinding_nordic_npm1304_regulator
- dtbinding_nordic_npm1304_wdt
- dtbinding_nordic_npm2100
- dtbinding_nordic_npm2100_gpio
- dtbinding_nordic_npm2100_regulator
- dtbinding_nordic_npm2100_vbat
- dtbinding_nordic_npm2100_wdt
- dtbinding_nordic_npm6001
- dtbinding_nordic_npm6001_gpio
- dtbinding_nordic_npm6001_regulator
- dtbinding_nordic_npm6001_wdt
- dtbinding_nordic_nrf_acl
- dtbinding_nordic_nrf_adc
- dtbinding_nordic_nrf_auxpll
- dtbinding_nordic_nrf_bellboard_rx
- dtbinding_nordic_nrf_bellboard_tx
- dtbinding_nordic_nrf_bicr
- dtbinding_nordic_nrf_bprot
- dtbinding_nordic_nrf_can
- dtbinding_nordic_nrf_ccm
- dtbinding_nordic_nrf_clic
- dtbinding_nordic_nrf_clock
- dtbinding_nordic_nrf_clock_hfclk
- dtbinding_nordic_nrf_clock_hfclk192m
- dtbinding_nordic_nrf_clock_hfclkaudio
- dtbinding_nordic_nrf_clock_lfclk
- dtbinding_nordic_nrf_clock_xo
- dtbinding_nordic_nrf_clock_xo24m
- dtbinding_nordic_nrf_comp
- dtbinding_nordic_nrf_cracen_ctrdrbg

- dtbinding_nordic_nrf_ctrlapperi
- dtbinding_nordic_nrf_dcnf
- dtbinding_nordic_nrf_dppic
- dtbinding_nordic_nrf_dppic_global
- dtbinding_nordic_nrf_dppic_local
- dtbinding_nordic_nrf_ecb
- dtbinding_nordic_nrf_egu
- dtbinding_nordic_nrf_exmif
- dtbinding_nordic_nrf_ficr
- dtbinding_nordic_nrf_fll16m
- dtbinding_nordic_nrf_gpio
- dtbinding_nordic_nrf_gpio_forwarder
- dtbinding_nordic_nrf_gpiote
- dtbinding_nordic_nrf_gpregret
- dtbinding_nordic_nrf_grtc
- dtbinding_nordic_nrf_gswdt
- dtbinding_nordic_nrf_hsfll_global
- dtbinding_nordic_nrf_hsfll_local
- dtbinding_nordic_nrf_i2s
- dtbinding_nordic_nrf_ieee802154
- dtbinding_nordic_nrf_ipc
- dtbinding_nordic_nrf_ipct_global
- dtbinding_nordic_nrf_ipct_local
- dtbinding_nordic_nrf_iron_hsfll_local
- dtbinding_nordic_nrf_kmu
- dtbinding_nordic_nrf_led_matrix
- dtbinding_nordic_nrf_lfrc
- dtbinding_nordic_nrf_lpcomp
- dtbinding_nordic_nrf_mpc
- dtbinding_nordic_nrf_mpu
- dtbinding_nordic_nrf_mramc
- dtbinding_nordic_nrf_mspi
- dtbinding_nordic_nrf_mutex
- dtbinding_nordic_nrf_mwu
- dtbinding_nordic_nrf_nfct
- dtbinding_nordic_nrf_nfct_v2
- dtbinding_nordic_nrf_pdm
- dtbinding_nordic_nrf_pinctrl
- dtbinding_nordic_nrf_power

- dtbinding_nordic_nrf_ppi
- dtbinding_nordic_nrf_ppib
- dtbinding_nordic_nrf_pwm
- dtbinding_nordic_nrf_pwr_antswc
- dtbinding_nordic_nrf_qdec
- dtbinding_nordic_nrf_qspi
- dtbinding_nordic_nrf_radio
- dtbinding_nordic_nrf_reset
- dtbinding_nordic_nrf_resetinfo
- dtbinding_nordic_nrf_rng
- dtbinding_nordic_nrf_rtc
- dtbinding_nordic_nrf_saadc
- dtbinding_nordic_nrf_spi
- dtbinding_nordic_nrf_spim
- dtbinding_nordic_nrf_spis
- dtbinding_nordic_nrf_spu
- dtbinding_nordic_nrf_sw_pwm
- dtbinding_nordic_nrf_swi
- dtbinding_nordic_nrf_tampc
- dtbinding_nordic_nrf_tbm
- dtbinding_nordic_nrf_tdm
- dtbinding_nordic_nrf_temp
- dtbinding_nordic_nrf_temp_nrfs
- dtbinding_nordic_nrf_timer
- dtbinding_nordic_nrf_twi
- dtbinding_nordic_nrf_twim
- dtbinding_nordic_nrf_twis
- dtbinding_nordic_nrf_uart
- dtbinding_nordic_nrf_uarte
- dtbinding_nordic_nrf_uicr
- dtbinding_nordic_nrf_usbd
- dtbinding_nordic_nrf_usbhs
- dtbinding_nordic_nrf_usbhs_bc12
- dtbinding_nordic_nrf_usbhs_phy_type1
- dtbinding_nordic_nrf_usbhs_phy_type2
- dtbinding_nordic_nrf_usbhs_wrapper
- dtbinding_nordic_nrf_usbreg
- dtbinding_nordic_nrf_vbat
- dtbinding_nordic_nrf_vevif_event_rx

- dtbinding_nordic_nrf_vevif_event_tx
- dtbinding_nordic_nrf_vevif_task_rx
- dtbinding_nordic_nrf_vevif_task_tx
- dtbinding_nordic_nrf_vmc
- dtbinding_nordic_nrf_vpr_coprocessor
- dtbinding_nordic_nrf_wdt
- dtbinding_nordic_nrf_wicr
- dtbinding_nordic_nrf21540_fem
- dtbinding_nordic_nrf21540_fem_spi
- dtbinding_nordic_nrf51_flash_controller
- dtbinding_nordic_nrf51_hfxo
- dtbinding_nordic_nrf52_flash_controller
- dtbinding_nordic_nrf52_hfxo
- dtbinding_nordic_nrf52x_regulator_hv
- dtbinding_nordic_nrf53_flash_controller
- dtbinding_nordic_nrf53_hfxo
- dtbinding_nordic_nrf53_lfxo
- dtbinding_nordic_nrf53_oscillators
- dtbinding_nordic_nrf53x_regulator_hv
- dtbinding_nordic_nrf53x_regulators
- dtbinding_nordic_nrf54h_hfxo
- dtbinding_nordic_nrf54h_lfxo
- dtbinding_nordic_nrf54l_hfxo
- dtbinding_nordic_nrf54l_lfxo
- dtbinding_nordic_nrf54l_regulators
- dtbinding_nordic_nrf5x_regulator
- dtbinding_nordic_nrf7000_coex
- dtbinding_nordic_nrf7000_qspi
- dtbinding_nordic_nrf7000_spi
- dtbinding_nordic_nrf7001_coex
- dtbinding_nordic_nrf7001_qspi
- dtbinding_nordic_nrf7001_spi
- dtbinding_nordic_nrf7002_coex
- dtbinding_nordic_nrf7002_qspi
- dtbinding_nordic_nrf7002_spi
- dtbinding_nordic_nrf71_hfxo
- dtbinding_nordic_nrf71_lfxo
- dtbinding_nordic_nrf71_p4_gpio
- dtbinding_nordic_nrf71_uicr

- dtbinding_nordic_nrf71_wifi_antsw
- dtbinding_nordic_nrf7120_wifi
- dtbinding_nordic_nrf91_flash_controller
- dtbinding_nordic_nrf91_slm
- dtbinding_nordic_nrf91_sm_v2
- dtbinding_nordic_nrf91x_regulators
- dtbinding_nordic_nrf93m1
- dtbinding_nordic_nrfs_audiopll
- dtbinding_nordic_nrfs_gdpwr
- dtbinding_nordic_nrfs_lfclk
- dtbinding_nordic_nrfs_swext
- dtbinding_nordic_owned_memory
- dtbinding_nordic_owned_partitions
- dtbinding_nordic_qspi_nor
- dtbinding_nordic_rram_controller
- dtbinding_nordic_tz_nonsecure
- dtbinding_nordic_tz_secure
- dtbinding_nordic_vpr
- dtbinding_nordic_vregusb_regulator
- dtbinding_nordic_wlan

Noritake Co., Inc. Electronics Division (noritake)

- dtbinding_noritake_itron

Nuclei System Technology (nuclei)

- dtbinding_nuclei_bumblebee
- dtbinding_nuclei_eclic
- dtbinding_nuclei_systimer

Nuvoton Technology Corporation (nuvoton)

- dtbinding_nuvoton_adc_cmp
- dtbinding_nuvoton_m4_hclk_clock
- dtbinding_nuvoton_m4_hxt_clock
- dtbinding_nuvoton_m4_lxt_clock
- dtbinding_nuvoton_m4_pll_clock
- dtbinding_nuvoton_nct38xx
- dtbinding_nuvoton_nct38xx_gpio
- dtbinding_nuvoton_nct38xx_gpio_alert
- dtbinding_nuvoton_nct38xx_gpio_port
- dtbinding_nuvoton_npck_lct

- dtbinding_nuvoton_npcm_pcc
- dtbinding_nuvoton_npcx_adc
- dtbinding_nuvoton_npcx_adc_v2t
- dtbinding_nuvoton_npcx_bbram
- dtbinding_nuvoton_npcx_booter_variant
- dtbinding_nuvoton_npcx_drbg
- dtbinding_nuvoton_npcx_espi
- dtbinding_nuvoton_npcx_espi_taf
- dtbinding_nuvoton_npcx_espi_vw_conf
- dtbinding_nuvoton_npcx_fiu_nor
- dtbinding_nuvoton_npcx_fiu_qspi
- dtbinding_nuvoton_npcx_gdma
- dtbinding_nuvoton_npcx_gpio
- dtbinding_nuvoton_npcx_host_sub
- dtbinding_nuvoton_npcx_host_uart
- dtbinding_nuvoton_npcx_i2c_ctrl
- dtbinding_nuvoton_npcx_i2c_port
- dtbinding_nuvoton_npcx_i3c
- dtbinding_nuvoton_npcx_itim_timer
- dtbinding_nuvoton_npcx_kbd
- dtbinding_nuvoton_npcx_lct_base
- dtbinding_nuvoton_npcx_lct_v1
- dtbinding_nuvoton_npcx_lct_v2
- dtbinding_nuvoton_npcx_leakage_io
- dtbinding_nuvoton_npcx_lvolctrl_conf
- dtbinding_nuvoton_npcx_miwu
- dtbinding_nuvoton_npcx_miwu_int_map
- dtbinding_nuvoton_npcx_miwu_wui_map
- dtbinding_nuvoton_npcx_pcc
- dtbinding_nuvoton_npcx_peci
- dtbinding_nuvoton_npcx_pinctrl
- dtbinding_nuvoton_npcx_pinctrl_conf
- dtbinding_nuvoton_npcx_pinctrl_def
- dtbinding_nuvoton_npcx_pinctrl_npckn
- dtbinding_nuvoton_npcx_power_psl
- dtbinding_nuvoton_npcx_power_state
- dtbinding_nuvoton_npcx_ps2_channel
- dtbinding_nuvoton_npcx_ps2_ctrl
- dtbinding_nuvoton_npcx_pwm

- dtbinding_nuvoton_npcx_rst
- dtbinding_nuvoton_npcx_scfg
- dtbinding_nuvoton_npcx_sha
- dtbinding_nuvoton_npcx_shi
- dtbinding_nuvoton_npcx_shi_enhanced
- dtbinding_nuvoton_npcx_soc_id
- dtbinding_nuvoton_npcx_spip
- dtbinding_nuvoton_npcx_tach
- dtbinding_nuvoton_npcx_uart
- dtbinding_nuvoton_npcx_uart_npckn
- dtbinding_nuvoton_npcx_watchdog
- dtbinding_nuvoton_numaker_adc
- dtbinding_nuvoton_numaker_canfd
- dtbinding_nuvoton_numaker_ethernet
- dtbinding_nuvoton_numaker_fmc
- dtbinding_nuvoton_numaker_gpio
- dtbinding_nuvoton_numaker_hsusbd
- dtbinding_nuvoton_numaker_i2c
- dtbinding_nuvoton_numaker_npu
- dtbinding_nuvoton_numaker_pcc
- dtbinding_nuvoton_numaker_pinctrl
- dtbinding_nuvoton_numaker_ppc
- dtbinding_nuvoton_numaker_pwm
- dtbinding_nuvoton_numaker_rmc
- dtbinding_nuvoton_numaker_rst
- dtbinding_nuvoton_numaker_rtc
- dtbinding_nuvoton_numaker_scc
- dtbinding_nuvoton_numaker_sdhc
- dtbinding_nuvoton_numaker_spi
- dtbinding_nuvoton_numaker_tcpc
- dtbinding_nuvoton_numaker_uart
- dtbinding_nuvoton_numaker_usbd
- dtbinding_nuvoton_numaker_usci_spi
- dtbinding_nuvoton_numaker_vbus
- dtbinding_nuvoton_numaker_wdt
- dtbinding_nuvoton_numaker_wwdt
- dtbinding_nuvoton_numicro_gpio
- dtbinding_nuvoton_numicro_m4_pcc
- dtbinding_nuvoton_numicro_m4_scc

- dtbinding_nuvoton_numicro_pinctrl
- dtbinding_nuvoton_numicro_uart

NXP Semiconductors (nxp)

- dtbinding_nxp_4ch_dma
- dtbinding_nxp_acomp
- dtbinding_nxp_adc12
- dtbinding_nxp_aon_lpadc
- dtbinding_nxp_aon_wakeup_pin
- dtbinding_nxp_bt_hci_uart
- dtbinding_nxp_c40_flash
- dtbinding_nxp_c40_flash_controller
- dtbinding_nxp_cam_44pins_connector
- dtbinding_nxp_cmc
- dtbinding_nxp_cmp
- dtbinding_nxp_cmu_fc
- dtbinding_nxp_cmu_fm
- dtbinding_nxp_cop
- dtbinding_nxp_crc
- dtbinding_nxp_ctimer_pwm
- dtbinding_nxp_dac12
- dtbinding_nxp_dai_esai
- dtbinding_nxp_dai_micfil
- dtbinding_nxp_dai_sai
- dtbinding_nxp_dcnano_lcdif
- dtbinding_nxp_dmic
- dtbinding_nxp_dspi
- dtbinding_nxp_edma
- dtbinding_nxp_ehci
- dtbinding_nxp_eim
- dtbinding_nxp_ele_trng
- dtbinding_nxp_els
- dtbinding_nxp_els_trng
- dtbinding_nxp_emios
- dtbinding_nxp_emios_pwm
- dtbinding_nxp_enet
- dtbinding_nxp_enet_mac
- dtbinding_nxp_enet_mdio
- dtbinding_nxp_enet_ptp_clock

- dtbinding_nxp_enet_qos
- dtbinding_nxp_erm
- dtbinding_nxp_espi
- dtbinding_nxp_espi_taf
- dtbinding_nxp_ewm
- dtbinding_nxp_firc
- dtbinding_nxp_flexcan
- dtbinding_nxp_flexcan_fd
- dtbinding_nxp_flexio
- dtbinding_nxp_flexio_camera
- dtbinding_nxp_flexio_pwm
- dtbinding_nxp_flexio_spi
- dtbinding_nxp_flexpwm
- dtbinding_nxp_flexram
- dtbinding_nxp_fmeas
- dtbinding_nxp_freqme
- dtbinding_nxp_fs26_wdog
- dtbinding_nxp_ftm
- dtbinding_nxp_ftm_pwm
- dtbinding_nxp_fxas21002_spi
- dtbinding_nxp_fxas21002_i2c
- dtbinding_nxp_fxls8974_i2c
- dtbinding_nxp_fxls8974_spi
- dtbinding_nxp_fxos8700_i2c
- dtbinding_nxp_fxos8700_spi
- dtbinding_nxp_fxosc
- dtbinding_nxp_gau_adc
- dtbinding_nxp_gau_dac
- dtbinding_nxp_gdma
- dtbinding_nxp_gint
- dtbinding_nxp_gpio_cluster
- dtbinding_nxp_gpt_hw_timer
- dtbinding_nxp_hci_ble
- dtbinding_nxp_hdlc_rcp_if
- dtbinding_nxp_hpdac
- dtbinding_nxp_hscmp
- dtbinding_nxp_i2c_tsc_fpc
- dtbinding_nxp_iap_fmc11
- dtbinding_nxp_iap_fmc54

- dtbinding_nxp_iap_fmc55
- dtbinding_nxp_iap_fmc553
- dtbinding_nxp_iap_fmc84x
- dtbinding_nxp_ii2c
- dtbinding_nxp_imx_anatop
- dtbinding_nxp_imx_blkctrl_ns_aon
- dtbinding_nxp_imx_blkctrl_wakeup
- dtbinding_nxp_imx_caam
- dtbinding_nxp_imx_ccm
- dtbinding_nxp_imx_ccm_fnpll
- dtbinding_nxp_imx_ccm_rev2
- dtbinding_nxp_imx_ccm_rev3
- dtbinding_nxp_imx_ccm_rev3_root
- dtbinding_nxp_imx_cguana
- dtbinding_nxp_imx_cguana_avpll
- dtbinding_nxp_imx_cguana_core_pll
- dtbinding_nxp_imx_cguana_frac_pll
- dtbinding_nxp_imx_cguana_fro12m
- dtbinding_nxp_imx_cguana_fro192m
- dtbinding_nxp_imx_cguana_sxosc
- dtbinding_nxp_imx_csi
- dtbinding_nxp_imx_dtcm
- dtbinding_nxp_imx_ecspi
- dtbinding_nxp_imx_elcdif
- dtbinding_nxp_imx_epit
- dtbinding_nxp_imx_flexspi
- dtbinding_nxp_imx_flexspi_aps6404l
- dtbinding_nxp_imx_flexspi_aps6408l
- dtbinding_nxp_imx_flexspi_hyperflash
- dtbinding_nxp_imx_flexspi_is66wvq8m4
- dtbinding_nxp_imx_flexspi_is66wvs8m8
- dtbinding_nxp_imx_flexspi_mx25um51345g
- dtbinding_nxp_imx_flexspi_nand
- dtbinding_nxp_imx_flexspi_nor
- dtbinding_nxp_imx_flexspi_s27ks0641
- dtbinding_nxp_imx_flexspi_w956a8mbya
- dtbinding_nxp_imx_gpio
- dtbinding_nxp_imx_gpio_v2
- dtbinding_nxp_imx_gpr

- dtbinding_nxp_imx_gpt
- dtbinding_nxp_imx_iomuxc
- dtbinding_nxp_imx_iomuxc_scu
- dtbinding_nxp_imx_itcm
- dtbinding_nxp_imx_iuart
- dtbinding_nxp_imx_lcdifv2
- dtbinding_nxp_imx_lcdifv3
- dtbinding_nxp_imx_mipi_dsi
- dtbinding_nxp_imx_mu
- dtbinding_nxp_imx_netc
- dtbinding_nxp_imx_netc_blk_ctrl
- dtbinding_nxp_imx_netc_emdio
- dtbinding_nxp_imx_netc_psi
- dtbinding_nxp_imx_netc_vsi
- dtbinding_nxp_imx_pwm
- dtbinding_nxp_imx_qtmr
- dtbinding_nxp_imx_rgpio
- dtbinding_nxp_imx_semc
- dtbinding_nxp_imx_snvs_gpr
- dtbinding_nxp_imx_snvs_rtc
- dtbinding_nxp_imx_src_rev2
- dtbinding_nxp_imx_tmr
- dtbinding_nxp_imx_uart
- dtbinding_nxp_imx_usdhc
- dtbinding_nxp_imx_wdog
- dtbinding_nxp_imx7d_pinctrl
- dtbinding_nxp_imx8_pinctrl
- dtbinding_nxp_imx8m_pinctrl
- dtbinding_nxp_imx8mp_pinctrl
- dtbinding_nxp_imx8ulp_pinctrl
- dtbinding_nxp_imx9_pll
- dtbinding_nxp_imx9_pstate
- dtbinding_nxp_imx93_mediamix
- dtbinding_nxp_imx93_pinctrl
- dtbinding_nxp_imx93_video_pll
- dtbinding_nxp_imxrt118x_arm_pll
- dtbinding_nxp_imxrt11xx_arm_pll
- dtbinding_nxp_inputmux
- dtbinding_nxp_ipc_rpmsg_lite

- dtbinding_nxp_ipwm
- dtbinding_nxp_irqsteer_intc
- dtbinding_nxp_irqsteer_master
- dtbinding_nxp_irtc
- dtbinding_nxp_irtc_wake_timer
- dtbinding_nxp_jpegdec
- dtbinding_nxp_kinetis_acmp
- dtbinding_nxp_kinetis_adc16
- dtbinding_nxp_kinetis_dac
- dtbinding_nxp_kinetis_dac32
- dtbinding_nxp_kinetis_ftfa
- dtbinding_nxp_kinetis_ftfc
- dtbinding_nxp_kinetis_ftfe
- dtbinding_nxp_kinetis_ftfl
- dtbinding_nxp_kinetis_gpio
- dtbinding_nxp_kinetis_i2c
- dtbinding_nxp_kinetis_ke1xf_sim
- dtbinding_nxp_kinetis_l_spi
- dtbinding_nxp_kinetis_lpsci
- dtbinding_nxp_kinetis_mcg
- dtbinding_nxp_kinetis_pcc
- dtbinding_nxp_kinetis_pwt
- dtbinding_nxp_kinetis_rnga
- dtbinding_nxp_kinetis_scg
- dtbinding_nxp_kinetis_sim
- dtbinding_nxp_kinetis_temperature
- dtbinding_nxp_kinetis_tpm
- dtbinding_nxp_kinetis_trng
- dtbinding_nxp_kinetis_uart
- dtbinding_nxp_kinetis_usbd
- dtbinding_nxp_kinetis_wdog
- dtbinding_nxp_kw41z_ieee802154
- dtbinding_nxp_lcd_8080
- dtbinding_nxp_lcd_pmod
- dtbinding_nxp_lcdic
- dtbinding_nxp_llwu
- dtbinding_nxp_lp_flexcomm
- dtbinding_nxp_lpacmp
- dtbinding_nxp_lpadc_temp40

- dtbinding_nxp_lpc_crc
- dtbinding_nxp_lpc_ctimer
- dtbinding_nxp_lpc_dma
- dtbinding_nxp_lpc_flexcomm
- dtbinding_nxp_lpc_gpio
- dtbinding_nxp_lpc_gpio_port
- dtbinding_nxp_lpc_i2c
- dtbinding_nxp_lpc_i2s
- dtbinding_nxp_lpc_iocon
- dtbinding_nxp_lpc_iocon_pinctrl
- dtbinding_nxp_lpc_iocon_pio
- dtbinding_nxp_lpc_lpadc
- dtbinding_nxp_lpc_mailbox
- dtbinding_nxp_lpc_mcan
- dtbinding_nxp_lpc_pmc_hwinfo
- dtbinding_nxp_lpc_rng
- dtbinding_nxp_lpc_rtc
- dtbinding_nxp_lpc_rtc_highres
- dtbinding_nxp_lpc_sdif
- dtbinding_nxp_lpc_spi
- dtbinding_nxp_lpc_syscon
- dtbinding_nxp_lpc_syscon_reset
- dtbinding_nxp_lpc_uid
- dtbinding_nxp_lpc_usart
- dtbinding_nxp_lpc_wwdt
- dtbinding_nxp_lpc11u6x_eeprom
- dtbinding_nxp_lpc11u6x_gpio
- dtbinding_nxp_lpc11u6x_i2c
- dtbinding_nxp_lpc11u6x_pinctrl
- dtbinding_nxp_lpc11u6x_syscon
- dtbinding_nxp_lpc11u6x_uart
- dtbinding_nxp_lpc84x_clock
- dtbinding_nxp_lpc84x_iocon
- dtbinding_nxp_lpc84x_swm
- dtbinding_nxp_lpc84x_uart
- dtbinding_nxp_lpcip3511
- dtbinding_nxp_lpcmp
- dtbinding_nxp_lpdac
- dtbinding_nxp_lpi2c

- dtbinding_nxp_lpit
- dtbinding_nxp_lpit_channel
- dtbinding_nxp_lpspi
- dtbinding_nxp_lptmr
- dtbinding_nxp_lpuart
- dtbinding_nxp_mbox_imx_mu
- dtbinding_nxp_mbox_mailbox
- dtbinding_nxp_mc_cgm
- dtbinding_nxp_mc_rgm
- dtbinding_nxp_mci_io_mux
- dtbinding_nxp_mcr20a
- dtbinding_nxp_mcux_12b1msps_sar
- dtbinding_nxp_mcux_dcp
- dtbinding_nxp_mcux_edma
- dtbinding_nxp_mcux_eqdc
- dtbinding_nxp_mcux_i2s
- dtbinding_nxp_mcux_i3c
- dtbinding_nxp_mcux_kpp
- dtbinding_nxp_mcux_qdc
- dtbinding_nxp_mcux_qdec
- dtbinding_nxp_mcux_rt_pinctrl
- dtbinding_nxp_mcux_rt11xx_pinctrl
- dtbinding_nxp_mcux_rt266x_pinctrl
- dtbinding_nxp_mcux_xbar
- dtbinding_nxp_mcxc_osc
- dtbinding_nxp_mcxe31x_siul2_pinctrl
- dtbinding_nxp_mcxn_pstate
- dtbinding_nxp_mcxw_hw_params
- dtbinding_nxp_mcxw_ieee802154
- dtbinding_nxp_mcxw_ifr
- dtbinding_nxp_mcxw7x_clock
- dtbinding_nxp_mecc
- dtbinding_nxp_micfil
- dtbinding_nxp_mipi_csi2rx
- dtbinding_nxp_mipi_dbi_dcnano_lcdif
- dtbinding_nxp_mipi_dbi_flexio_lcdif
- dtbinding_nxp_mipi_dsi_2l
- dtbinding_nxp_mipi_dsi_dwc
- dtbinding_nxp_mrcc_reset

- dtbinding_nxp_mrt
- dtbinding_nxp_mrt_channel
- dtbinding_nxp_msf1
- dtbinding_nxp_nbu
- dtbinding_nxp_netc_ptp_clock
- dtbinding_nxp_netc_switch
- dtbinding_nxp_nx20p3483
- dtbinding_nxp_ocotp
- dtbinding_nxp_opamp
- dtbinding_nxp_opamp_fast
- dtbinding_nxp_os_timer
- dtbinding_nxp_otp_uid
- dtbinding_nxp_otpc
- dtbinding_nxp_p3t1755_i3c
- dtbinding_nxp_p3t1755_i2c
- dtbinding_nxp_parallel_lcd_connector
- dtbinding_nxp_pca6408
- dtbinding_nxp_pca6416
- dtbinding_nxp_pca9420
- dtbinding_nxp_pca9422
- dtbinding_nxp_pca9422_charger
- dtbinding_nxp_pca9422_regulator
- dtbinding_nxp_pca9530
- dtbinding_nxp_pca9531
- dtbinding_nxp_pca9532
- dtbinding_nxp_pca9533
- dtbinding_nxp_pca9538
- dtbinding_nxp_pca9539
- dtbinding_nxp_pca9554
- dtbinding_nxp_pca9555
- dtbinding_nxp_pca95xx
- dtbinding_nxp_pca9633
- dtbinding_nxp_pca9685_pwm
- dtbinding_nxp_pcal6408
- dtbinding_nxp_pcal6408a
- dtbinding_nxp_pcal6416
- dtbinding_nxp_pcal6416a
- dtbinding_nxp_pcal6524
- dtbinding_nxp_pcal6534

- dtbinding_nxp_pcal9538
- dtbinding_nxp_pcal9539
- dtbinding_nxp_pcal9722
- dtbinding_nxp_pcf2123
- dtbinding_nxp_pcf85063a
- dtbinding_nxp_pcf8523
- dtbinding_nxp_pcf8563
- dtbinding_nxp_pcf857x
- dtbinding_nxp_pdcfg_power
- dtbinding_nxp_pf1550
- dtbinding_nxp_pf1550_charger
- dtbinding_nxp_pf1550_regulator
- dtbinding_nxp_pint
- dtbinding_nxp_pit
- dtbinding_nxp_pit_channel
- dtbinding_nxp_plldig
- dtbinding_nxp_pmc_tmpsns
- dtbinding_nxp_pngdec
- dtbinding_nxp_port_pinctrl
- dtbinding_nxp_port_pinmux
- dtbinding_nxp_powerquad
- dtbinding_nxp_pxp
- dtbinding_nxp_qdec_s32
- dtbinding_nxp_qspi
- dtbinding_nxp_qtmr_pwm
- dtbinding_nxp_rcm_hwinfo
- dtbinding_nxp_rdc
- dtbinding_nxp_rstctl
- dtbinding_nxp_rstctl_hwinfo
- dtbinding_nxp_rt_iocon_pinctrl
- dtbinding_nxp_rt600_dsp_ctrl
- dtbinding_nxp_rt700_dsp_ctrl_hifi4
- dtbinding_nxp_rt7xx_ocotp
- dtbinding_nxp_rtc
- dtbinding_nxp_rtc_analog
- dtbinding_nxp_rtc_jdp
- dtbinding_nxp_rtwdog
- dtbinding_nxp_rtxxx_dsp_ctrl
- dtbinding_nxp_rw_pmu

- dtbinding_nxp_rw_soc_ctrl
- dtbinding_nxp_s32_adc_sar
- dtbinding_nxp_s32_canxl
- dtbinding_nxp_s32_clock
- dtbinding_nxp_s32_crypto_hse_mu
- dtbinding_nxp_s32_emios
- dtbinding_nxp_s32_emios_pwm
- dtbinding_nxp_s32_gmac
- dtbinding_nxp_s32_lcu
- dtbinding_nxp_s32_linflexd
- dtbinding_nxp_s32_mc_me
- dtbinding_nxp_s32_mc_rgm
- dtbinding_nxp_s32_mru
- dtbinding_nxp_s32_netc_emdio
- dtbinding_nxp_s32_netc_psi
- dtbinding_nxp_s32_netc_vsi
- dtbinding_nxp_s32_psi5
- dtbinding_nxp_s32_qspi
- dtbinding_nxp_s32_qspi_device
- dtbinding_nxp_s32_qspi_hyperflash
- dtbinding_nxp_s32_qspi_nor
- dtbinding_nxp_s32_qspi_sfp_frad
- dtbinding_nxp_s32_qspi_sfp_mdad
- dtbinding_nxp_s32_sent
- dtbinding_nxp_s32_spi
- dtbinding_nxp_s32_swt
- dtbinding_nxp_s32_sys_timer
- dtbinding_nxp_s32_trgmux
- dtbinding_nxp_s32_wkpu
- dtbinding_nxp_s32_xspi
- dtbinding_nxp_s32_xspi_device
- dtbinding_nxp_s32_xspi_hyperram
- dtbinding_nxp_s32_xspi_sfp_frad
- dtbinding_nxp_s32_xspi_sfp_mdad
- dtbinding_nxp_s32k3_pmc
- dtbinding_nxp_s32k3_siul2_pinctrl
- dtbinding_nxp_s32k5_pinctrl
- dtbinding_nxp_s32ze_siul2_pinctrl
- dtbinding_nxp_sar_adc

- dtbinding_nxp_sc18im704
- dtbinding_nxp_sc18im704_gpio
- dtbinding_nxp_sc18im704_i2c
- dtbinding_nxp_sc18is606
- dtbinding_nxp_sc18is606_gpio
- dtbinding_nxp_sc18is606_spi
- dtbinding_nxp_scg_k4
- dtbinding_nxp_scmi_cpu
- dtbinding_nxp_sctimer_pwm
- dtbinding_nxp_scu_pd
- dtbinding_nxp_sdma
- dtbinding_nxp_sema42
- dtbinding_nxp_sim_uuid
- dtbinding_nxp_siul2_eirq
- dtbinding_nxp_siul2_gpio
- dtbinding_nxp_slcd
- dtbinding_nxp_sleepcon_wuc
- dtbinding_nxp_smartdma
- dtbinding_nxp_smc
- dtbinding_nxp_sof_host_dma
- dtbinding_nxp_spc
- dtbinding_nxp_spc_bandgap_buffer
- dtbinding_nxp_sramc
- dtbinding_nxp_stm
- dtbinding_nxp_sysctr
- dtbinding_nxp_sysmpu
- dtbinding_nxp_t1s_phy
- dtbinding_nxp_tempmon
- dtbinding_nxp_tempsense
- dtbinding_nxp_tja1103
- dtbinding_nxp_tja11xx
- dtbinding_nxp_tmpsns
- dtbinding_nxp_tpm_qdec
- dtbinding_nxp_tpm_timer
- dtbinding_nxp_trgmux
- dtbinding_nxp_tsi_input
- dtbinding_nxp_tstmr
- dtbinding_nxp_uhc_ehci
- dtbinding_nxp_uhc_ip3516hs

- dtbinding_nxp_uhc_khci
- dtbinding_nxp_uhc_ohci
- dtbinding_nxp_usbphy
- dtbinding_nxp_vbat
- dtbinding_nxp_vf610_adc
- dtbinding_nxp_video_smartdma
- dtbinding_nxp_vref
- dtbinding_nxp_vrefv1
- dtbinding_nxp_wake_timer
- dtbinding_nxp_wdog32
- dtbinding_nxp_wifi
- dtbinding_nxp_wuc_wuu
- dtbinding_nxp_wuu
- dtbinding_nxp_xspi
- dtbinding_nxp_xspi_nor
- dtbinding_nxp_xspi_psram

OmniVision Technologies Inc. (ovti)

- dtbinding_ovti_ov2640
- dtbinding_ovti_ov5640
- dtbinding_ovti_ov5642
- dtbinding_ovti_ov7670
- dtbinding_ovti_ov7675
- dtbinding_ovti_ov7725
- dtbinding_ovti_ov9655

OMRON Corporation (omron)

- dtbinding_omron_2smpb_02e
- dtbinding_omron_d6f_p0001
- dtbinding_omron_d6f_p0010

ON Semiconductor Corp. (onnn)

- dtbinding_onnn_fusb307_tcpc
- dtbinding_onnn_lc709203f
- dtbinding_onnn_ncp5623
- dtbinding_onnn_nct75

open-isa.org (openisa)

- dtbinding_openisa_ri5cy
- dtbinding_openisa_rv32m1_event_unit
- dtbinding_openisa_rv32m1_ftfe
- dtbinding_openisa_rv32m1_genfsk
- dtbinding_openisa_rv32m1_gpio
- dtbinding_openisa_rv32m1_intmux
- dtbinding_openisa_rv32m1_intmux_ch
- dtbinding_openisa_rv32m1_lpi2c
- dtbinding_openisa_rv32m1_lpspi
- dtbinding_openisa_rv32m1_lptmr
- dtbinding_openisa_rv32m1_lpuart
- dtbinding_openisa_rv32m1_pcc
- dtbinding_openisa_rv32m1_pinctrl
- dtbinding_openisa_rv32m1_pinmux
- dtbinding_openisa_rv32m1_tpm
- dtbinding_openisa_rv32m1_trng
- dtbinding_openisa_zero_ri5cy

OpenCores.org (opencores)

- dtbinding_opencores_or1k_pic_level
- dtbinding_opencores_spi_simple

OpenHW Group (openhwgroup)

- dtbinding_openhwgroup_cva6

OpenThread.io (openthread)

- dtbinding_openthread_config

Orise Technology (orisetech)

- dtbinding_orisetech_otm8009a

Panasonic Corporation (panasonic)

- dtbinding_panasonic_amg88xx
- dtbinding_panasonic_reduced_arduino_header

Parade Technologies Inc. (parade)

- dtbinding_parade_ps8xxx
- dtbinding_parade_tma525b

PixArt Imaging Inc. (pixart)

- dtbinding_pixart_paa3905
- dtbinding_pixart_paj7620
- dtbinding_pixart_pat912x
- dtbinding_pixart_pat9136
- dtbinding_pixart_paw32xx
- dtbinding_pixart_pmw3610

Plantower Co., Ltd (plantower)

- dtbinding_plantower_pms7003
- dtbinding_plantower_pmsa003i

PNI Sensor Corporation (pni)

- dtbinding_pni_rm3100_i2c
- dtbinding_pni_rm3100_spi

Princeton Technology Corp. (ptc)

- dtbinding_ptc_pt6314

QEMU, a generic and open source machine emulator and virtualizer (qemu)

- dtbinding_qemu_fw_cfg_ioport
- dtbinding_qemu_fw_cfg_mmio
- dtbinding_qemu_ivshmem
- dtbinding_qemu_ramfb
- dtbinding_qemu_riscv_virt

Qorvo, Inc (formerly Decawave) (decawave)

- dtbinding_decawave_dw1000

QST Corporation (qst)

- dtbinding_qst_qmi8658a

Qualcomm Atheros, Inc. (qca)

- dtbinding_qca_ar8031

Qualcomm Technologies, Inc (qcom)

- dtbinding_qcom_hexagon_v67
- dtbinding_qcom_hvm_pic
- dtbinding_qcom_hvm_timer

Quectel Wireless Solutions Co., Ltd. (quectel)

- dtbinding_quectel_bc66
- dtbinding_quectel_bc660k
- dtbinding_quectel_bc66x
- dtbinding_quectel_bg770
- dtbinding_quectel_bg95
- dtbinding_quectel_bg96
- dtbinding_quectel_bg9x
- dtbinding_quectel_eg21_g
- dtbinding_quectel_eg25_g
- dtbinding_quectel_eg800q
- dtbinding_quectel_eg915u
- dtbinding_quectel_lc26g
- dtbinding_quectel_lc76g
- dtbinding_quectel_lc86g

QuickLogic Corp. (quicklogic)

- dtbinding_quicklogic_eos_s3_gpio
- dtbinding_quicklogic_eos_s3_pinctrl
- dtbinding_quicklogic_usbserialport_s3b

Raspberry Pi Foundation (raspberrypi)

- dtbinding_raspberrypi_bcm2711_framebuffer
- dtbinding_raspberrypi_bcm283x_firmware
- dtbinding_raspberrypi_bcm283x_gpio
- dtbinding_raspberrypi_bcm283x_power
- dtbinding_raspberrypi_bcm283x_vc_thermal
- dtbinding_raspberrypi_core_supply_regulator
- dtbinding_raspberrypi_csi_connector
- dtbinding_raspberrypi_pico_adc
- dtbinding_raspberrypi_pico_clock
- dtbinding_raspberrypi_pico_clock_controller
- dtbinding_raspberrypi_pico_dma
- dtbinding_raspberrypi_pico_flash_controller
- dtbinding_raspberrypi_pico_gpio
- dtbinding_raspberrypi_pico_gpio_port
- dtbinding_raspberrypi_pico_header
- dtbinding_raspberrypi_pico_i2c
- dtbinding_raspberrypi_pico_mbox
- dtbinding_raspberrypi_pico_mipi_dbi_pio

- dtbinding_raspberrypi_pico_pinctrl
- dtbinding_raspberrypi_pico_pio
- dtbinding_raspberrypi_pico_pio_device
- dtbinding_raspberrypi_pico_pit
- dtbinding_raspberrypi_pico_pit_channel
- dtbinding_raspberrypi_pico_pll
- dtbinding_raspberrypi_pico_powman
- dtbinding_raspberrypi_pico_pwm
- dtbinding_raspberrypi_pico_reset
- dtbinding_raspberrypi_pico_rng
- dtbinding_raspberrypi_pico_rosc
- dtbinding_raspberrypi_pico_rtc
- dtbinding_raspberrypi_pico_sha256
- dtbinding_raspberrypi_pico_sio
- dtbinding_raspberrypi_pico_spi
- dtbinding_raspberrypi_pico_spi_pio
- dtbinding_raspberrypi_pico_temp
- dtbinding_raspberrypi_pico_timer
- dtbinding_raspberrypi_pico_uart
- dtbinding_raspberrypi_pico_uart_pio
- dtbinding_raspberrypi_pico_usbd
- dtbinding_raspberrypi_pico_watchdog
- dtbinding_raspberrypi_pico_xosc
- dtbinding_raspberrypi_rp1_gpio
- dtbinding_raspberrypi_rp1_pinctrl

Raydium Semiconductor Corp. (raydium)

- dtbinding_raydium_rm67162
- dtbinding_raydium_rm67199
- dtbinding_raydium_rm68200
- dtbinding_raydium_rm692c9

Realtek Semiconductor Corp. (realtek)

- dtbinding_realtek_ameba_adc
- dtbinding_realtek_ameba_bt_hci
- dtbinding_realtek_ameba_counter
- dtbinding_realtek_ameba_flash_controller
- dtbinding_realtek_ameba_gdma
- dtbinding_realtek_ameba_gpio
- dtbinding_realtek_ameba_i2c

- dtbinding_realtek_ameba_loguart
- dtbinding_realtek_ameba_pinctrl
- dtbinding_realtek_ameba_pwm
- dtbinding_realtek_ameba_rcc
- dtbinding_realtek_ameba_rtc
- dtbinding_realtek_ameba_sdhost
- dtbinding_realtek_ameba_spi
- dtbinding_realtek_ameba_trng
- dtbinding_realtek_ameba_watchdog
- dtbinding_realtek_ameba_wifi
- dtbinding_realtek_bee_adc
- dtbinding_realtek_bee_aes
- dtbinding_realtek_bee_aon_qdec
- dtbinding_realtek_bee_basic_qdec
- dtbinding_realtek_bee_basic_timer
- dtbinding_realtek_bee_bt_hci
- dtbinding_realtek_bee_can
- dtbinding_realtek_bee_cctl
- dtbinding_realtek_bee_core_wdt
- dtbinding_realtek_bee_counter_rtc
- dtbinding_realtek_bee_counter_timer
- dtbinding_realtek_bee_dma
- dtbinding_realtek_bee_enhanced_timer
- dtbinding_realtek_bee_gpio
- dtbinding_realtek_bee_i2c
- dtbinding_realtek_bee_keyscan
- dtbinding_realtek_bee_nor_flash
- dtbinding_realtek_bee_nor_flash_controller
- dtbinding_realtek_bee_pinctrl
- dtbinding_realtek_bee_pwm
- dtbinding_realtek_bee_qdec
- dtbinding_realtek_bee_sha256
- dtbinding_realtek_bee_spi
- dtbinding_realtek_bee_timer
- dtbinding_realtek_bee_trng
- dtbinding_realtek_bee_uart
- dtbinding_realtek_rtl8211f
- dtbinding_realtek_rts5817_clock
- dtbinding_realtek_rts5817_gpio

- dtbinding_realtek_rts5817_ocotp
- dtbinding_realtek_rts5817_pinctrl
- dtbinding_realtek_rts5817_regulator
- dtbinding_realtek_rts5817_reset
- dtbinding_realtek_rts5817_spi
- dtbinding_realtek_rts5817_watchdog
- dtbinding_realtek_rts5912_adc
- dtbinding_realtek_rts5912_bbram
- dtbinding_realtek_rts5912_espi
- dtbinding_realtek_rts5912_flash_controller
- dtbinding_realtek_rts5912_gpio
- dtbinding_realtek_rts5912_i2c
- dtbinding_realtek_rts5912_kbd
- dtbinding_realtek_rts5912_pinctrl
- dtbinding_realtek_rts5912_pwm
- dtbinding_realtek_rts5912_rtc
- dtbinding_realtek_rts5912_rtmr
- dtbinding_realtek_rts5912_sccon
- dtbinding_realtek_rts5912_sha
- dtbinding_realtek_rts5912_slwtimer
- dtbinding_realtek_rts5912_spi
- dtbinding_realtek_rts5912_tach
- dtbinding_realtek_rts5912_timer
- dtbinding_realtek_rts5912_uart
- dtbinding_realtek_rts5912_ulpm
- dtbinding_realtek_rts5912_watchdog

Renesas Electronics Corporation (renesas)

- dtbinding_renesas_bt_hci_da1453x
- dtbinding_renesas_bt_hci_da1469x
- dtbinding_renesas_hs300x
- dtbinding_renesas_hs400x
- dtbinding_renesas_ofs_memory
- dtbinding_renesas_pwm_rcar
- dtbinding_renesas_r8a7795_cpg_mssr
- dtbinding_renesas_r8a779f0_cpg_mssr
- dtbinding_renesas_r8a779g0_cpg_mssr
- dtbinding_renesas_ra_acmphs
- dtbinding_renesas_ra_acmphs_global

- dtbinding_renesas_ra_adc_b
- dtbinding_renesas_ra_adc_b_controller
- dtbinding_renesas_ra_adc_b_scan_group
- dtbinding_renesas_ra_adc12
- dtbinding_renesas_ra_adc16
- dtbinding_renesas_ra_agt
- dtbinding_renesas_ra_agt_counter
- dtbinding_renesas_ra_battery_backup
- dtbinding_renesas_ra_canfd
- dtbinding_renesas_ra_canfd_global
- dtbinding_renesas_ra_ceu
- dtbinding_renesas_ra_cgc_busclk
- dtbinding_renesas_ra_cgc_external_clock
- dtbinding_renesas_ra_cgc_pclk
- dtbinding_renesas_ra_cgc_pclk_block
- dtbinding_renesas_ra_cgc_pll
- dtbinding_renesas_ra_cgc_pll_out
- dtbinding_renesas_ra_cgc_subclk
- dtbinding_renesas_ra_cgc_utasel
- dtbinding_renesas_ra_crc
- dtbinding_renesas_ra_ctsu
- dtbinding_renesas_ra_ctsu_button
- dtbinding_renesas_ra_ctsu_slider
- dtbinding_renesas_ra_ctsu_wheel
- dtbinding_renesas_ra_dac
- dtbinding_renesas_ra_dac_global
- dtbinding_renesas_ra_dma
- dtbinding_renesas_ra_drw
- dtbinding_renesas_ra_elc
- dtbinding_renesas_ra_eswm
- dtbinding_renesas_ra_ethernet
- dtbinding_renesas_ra_ethernet_rmac
- dtbinding_renesas_ra_external_interrupt
- dtbinding_renesas_ra_faci
- dtbinding_renesas_ra_fcb
- dtbinding_renesas_ra_flash_hp_controller
- dtbinding_renesas_ra_flash_lp_controller
- dtbinding_renesas_ra_glcdc
- dtbinding_renesas_ra_gpio_ioport

- dtbinding_renesas_ra_gpio_mipi_header
- dtbinding_renesas_ra_i2c_sci
- dtbinding_renesas_ra_i2c_sci_b
- dtbinding_renesas_ra_i2s_ssie
- dtbinding_renesas_ra_i3c
- dtbinding_renesas_ra_iic
- dtbinding_renesas_ra_ipc_mbox
- dtbinding_renesas_ra_lin_sci_b
- dtbinding_renesas_ra_lvd
- dtbinding_renesas_ra_mdio
- dtbinding_renesas_ra_mdio_rmac
- dtbinding_renesas_ra_mipi_dsi
- dtbinding_renesas_ra_mram_controller
- dtbinding_renesas_ra_npu
- dtbinding_renesas_ra_nv_mram
- dtbinding_renesas_ra_ospi_b
- dtbinding_renesas_ra_ospi_b_nor
- dtbinding_renesas_ra_parallel_graphics_header
- dtbinding_renesas_ra_pdm
- dtbinding_renesas_ra_pinctrl_pfs
- dtbinding_renesas_ra_pwm
- dtbinding_renesas_ra_qspi
- dtbinding_renesas_ra_qspi_nor
- dtbinding_renesas_ra_rsip_e50d_trng
- dtbinding_renesas_ra_rsip_e51a_trng
- dtbinding_renesas_ra_rtc
- dtbinding_renesas_ra_sau
- dtbinding_renesas_ra_sau_channel
- dtbinding_renesas_ra_sce5_rng
- dtbinding_renesas_ra_sce7_rng
- dtbinding_renesas_ra_sce9_rng
- dtbinding_renesas_ra_sci
- dtbinding_renesas_ra_sci_uart
- dtbinding_renesas_ra_sdhc
- dtbinding_renesas_ra_sdram
- dtbinding_renesas_ra_spi
- dtbinding_renesas_ra_spi_sci
- dtbinding_renesas_ra_spi_sci_b
- dtbinding_renesas_ra_trng

- dtbinding_renesas_ra_uart_sau
- dtbinding_renesas_ra_udc
- dtbinding_renesas_ra_uhc
- dtbinding_renesas_ra_ulpt
- dtbinding_renesas_ra_ulpt_counter
- dtbinding_renesas_ra_ulpt_timer
- dtbinding_renesas_ra_usbfs
- dtbinding_renesas_ra_usbhs
- dtbinding_renesas_ra_usbphyc
- dtbinding_renesas_ra_wdt
- dtbinding_renesas_ra0_pinctrl_pfs
- dtbinding_renesas_ra8_spi_b
- dtbinding_renesas_ra8_uart_sci_b
- dtbinding_renesas_rcar_can
- dtbinding_renesas_rcar_cmt
- dtbinding_renesas_rcar_gpio
- dtbinding_renesas_rcar_hscif
- dtbinding_renesas_rcar_i2c
- dtbinding_renesas_rcar_mfis_mbox
- dtbinding_renesas_rcar_mmc
- dtbinding_renesas_rcar_pfc
- dtbinding_renesas_rcar_pfc_x5h
- dtbinding_renesas_rcar_rscanfd
- dtbinding_renesas_rcar_rscanfd_global
- dtbinding_renesas_rcar_scif
- dtbinding_renesas_rx
- dtbinding_renesas_rx_adc
- dtbinding_renesas_rx_cgc_pclk
- dtbinding_renesas_rx_cgc_pclk_block
- dtbinding_renesas_rx_cgc_pll
- dtbinding_renesas_rx_cgc_pllccr_type
- dtbinding_renesas_rx_cgc_root_clock
- dtbinding_renesas_rx_ctsu
- dtbinding_renesas_rx_ctsu_button
- dtbinding_renesas_rx_ctsu_slider
- dtbinding_renesas_rx_ctsu_wheel
- dtbinding_renesas_rx_dtc
- dtbinding_renesas_rx_external_interrupt
- dtbinding_renesas_rx_flash

- dtbinding_renesas_rx_gpio
- dtbinding_renesas_rx_gpio_ioport
- dtbinding_renesas_rx_grp_intc
- dtbinding_renesas_rx_i2c
- dtbinding_renesas_rx_icu
- dtbinding_renesas_rx_icum
- dtbinding_renesas_rx_iwdt
- dtbinding_renesas_rx_lvd
- dtbinding_renesas_rx_mtu
- dtbinding_renesas_rx_mtu_pwm
- dtbinding_renesas_rx_nv_flash
- dtbinding_renesas_rx_pinctrl
- dtbinding_renesas_rx_pinctrl_pfs
- dtbinding_renesas_rx_pinmux
- dtbinding_renesas_rx_rspi
- dtbinding_renesas_rx_sci
- dtbinding_renesas_rx_sci_b
- dtbinding_renesas_rx_swint
- dtbinding_renesas_rx_system_controller
- dtbinding_renesas_rx_timer_cmt
- dtbinding_renesas_rx_timer_cmt_start_control
- dtbinding_renesas_rx_uart_sci
- dtbinding_renesas_rx_uart_sci_b
- dtbinding_renesas_rx_uart_sci_qemu
- dtbinding_renesas_rxv1
- dtbinding_renesas_rxv2
- dtbinding_renesas_rxv3
- dtbinding_renesas_rxv3e
- dtbinding_renesas_rz_adc
- dtbinding_renesas_rz_adc_c
- dtbinding_renesas_rz_adc_e
- dtbinding_renesas_rz_canfd
- dtbinding_renesas_rz_canfd_global
- dtbinding_renesas_rz_cgc
- dtbinding_renesas_rz_cgc_pll
- dtbinding_renesas_rz_cgc_subclk
- dtbinding_renesas_rz_cgc_sys_clock
- dtbinding_renesas_rz_cmtw
- dtbinding_renesas_rz_cmtw_counter

- dtbinding_renesas_rz_cpg
- dtbinding_renesas_rz_cpg_clock
- dtbinding_renesas_rz_cpg_pll
- dtbinding_renesas_rz_dmac
- dtbinding_renesas_rz_dmac_b
- dtbinding_renesas_rz_ext_irq
- dtbinding_renesas_rz_gpio
- dtbinding_renesas_rz_gpio_common
- dtbinding_renesas_rz_gpio_common_v2
- dtbinding_renesas_rz_gpio_common_v3
- dtbinding_renesas_rz_gpio_int
- dtbinding_renesas_rz_gpio_mipi_header
- dtbinding_renesas_rz_gpt
- dtbinding_renesas_rz_gpt_pwm
- dtbinding_renesas_rz_gtm
- dtbinding_renesas_rz_gtm_counter
- dtbinding_renesas_rz_gtm_os_timer
- dtbinding_renesas_rz_icu
- dtbinding_renesas_rz_icu_v2
- dtbinding_renesas_rz_iic
- dtbinding_renesas_rz_intc
- dtbinding_renesas_rz_intc_v2
- dtbinding_renesas_rz_lcdc
- dtbinding_renesas_rz_mhu_mbox
- dtbinding_renesas_rz_mipi_dsi
- dtbinding_renesas_rz_mtu
- dtbinding_renesas_rz_mtu_pwm
- dtbinding_renesas_rz_qspi_spibsc
- dtbinding_renesas_rz_qspi_xspi
- dtbinding_renesas_rz_riic
- dtbinding_renesas_rz_rspi
- dtbinding_renesas_rz_sci
- dtbinding_renesas_rz_sci_b
- dtbinding_renesas_rz_sci_b_uart
- dtbinding_renesas_rz_sci_uart
- dtbinding_renesas_rz_scif_uart
- dtbinding_renesas_rz_spi
- dtbinding_renesas_rz_spi_b
- dtbinding_renesas_rz_spibsc

- dtbinding_renesas_rz_tint
- dtbinding_renesas_rz_wdt
- dtbinding_renesas_rz_xspi
- dtbinding_renesas_rza_pinctrl
- dtbinding_renesas_rza2m_adc
- dtbinding_renesas_rza2m_cpg
- dtbinding_renesas_rza2m_dma
- dtbinding_renesas_rza2m_gpio
- dtbinding_renesas_rza2m_gpio_int
- dtbinding_renesas_rza2m_gpt
- dtbinding_renesas_rza2m_gpt_pwm
- dtbinding_renesas_rza2m_ostm
- dtbinding_renesas_rza2m_ostm_counter
- dtbinding_renesas_rza2m_ostm_timer
- dtbinding_renesas_rza2m_pinctrl
- dtbinding_renesas_rza2m_qspi_spibsc
- dtbinding_renesas_rza2m_riic
- dtbinding_renesas_rza2m_scif_uart
- dtbinding_renesas_rzg_pinctrl
- dtbinding_renesas_rzn_pinctrl
- dtbinding_renesas_rzt_pinctrl
- dtbinding_renesas_rzt2m_gpio
- dtbinding_renesas_rzt2m_gpio_common
- dtbinding_renesas_rzt2m_pinctrl
- dtbinding_renesas_rzt2m_uart
- dtbinding_renesas_rzv_pinctrl
- dtbinding_renesas_slg47105
- dtbinding_renesas_slg47115
- dtbinding_renesas_slg471x5
- dtbinding_renesas_slg47910
- dtbinding_renesas_smartbond_adc
- dtbinding_renesas_smartbond_crypto
- dtbinding_renesas_smartbond_display
- dtbinding_renesas_smartbond_dma
- dtbinding_renesas_smartbond_flash_controller
- dtbinding_renesas_smartbond_gpio
- dtbinding_renesas_smartbond_i2c
- dtbinding_renesas_smartbond_lp_clk
- dtbinding_renesas_smartbond_lp_osc

- dtbinding_renesas_smartbond_mipi_dbi
- dtbinding_renesas_smartbond_nor_psram
- dtbinding_renesas_smartbond_pinctrl
- dtbinding_renesas_smartbond_regulator
- dtbinding_renesas_smartbond_rtc
- dtbinding_renesas_smartbond_sdadc
- dtbinding_renesas_smartbond_spi
- dtbinding_renesas_smartbond_sys_clk
- dtbinding_renesas_smartbond_timer
- dtbinding_renesas_smartbond_trng
- dtbinding_renesas_smartbond_uart
- dtbinding_renesas_smartbond_usbd
- dtbinding_renesas_smartbond_watchdog

Reyax Technology Co., Ltd. (reyax)

- dtbinding_reyax_rylrxxx

Richtek Technology Corporation (richtek)

- dtbinding_richtek_rt1715
- dtbinding_richtek_rt1718s
- dtbinding_richtek_rt1718s_gpio_port

RISC-V Foundation (riscv)

- dtbinding_riscv_aplic
- dtbinding_riscv_clic
- dtbinding_riscv_cpu_intc
- dtbinding_riscv_imsic
- dtbinding_riscv_machine_timer

ROCKTECH DISPLAYS LIMITED (rocktech)

- dtbinding_rocktech_rk043fn02h_ct

ROHM Semiconductor Co., Ltd (rohm)

- dtbinding_rohm_bd8lb600fs
- dtbinding_rohm_bd8lb600fs_diagnostics
- dtbinding_rohm_bd8lb600fs_gpio
- dtbinding_rohm_bh1730
- dtbinding_rohm_bh1750
- dtbinding_rohm_bh1790

Sciosense B.V. (sciosense)

- dtbinding_sciosense_ens160_spi
- dtbinding_sciosense_ens160_i2c

Seeed Technology Co., Ltd (seeed)

- dtbinding_seeed_grove_lcd_rgb
- dtbinding_seeed_grove_light
- dtbinding_seeed_grove_temperature
- dtbinding_seeed_hm330x
- dtbinding_seeed_xiao_expand
- dtbinding_seeed_xiao_gpio

SEGGER Microcontroller GmbH (segger)

- dtbinding_segger_rtt_uart

Seiko Epson Corp. (epson)

- dtbinding_epson_rx8130ce_rtc

Semtech Corporation (semtech)

- dtbinding_semtech_llcc68
- dtbinding_semtech_lr1121
- dtbinding_semtech_sx1261
- dtbinding_semtech_sx1262
- dtbinding_semtech_sx1268
- dtbinding_semtech_sx1272
- dtbinding_semtech_sx1276
- dtbinding_semtech_sx1278
- dtbinding_semtech_sx1509b
- dtbinding_semtech_sx9500

Sensirion AG (sensirion)

- dtbinding_sensirion_scd40
- dtbinding_sensirion_scd41
- dtbinding_sensirion_sgp40
- dtbinding_sensirion_sht21
- dtbinding_sensirion_sht3xd
- dtbinding_sensirion_sht4x
- dtbinding_sensirion_shtcx
- dtbinding_sensirion_stcc4
- dtbinding_sensirion_sts4x

sensry.io (sensry)

- dtbinding_sensry_sy1xx
- dtbinding_sensry_sy1xx_event_unit
- dtbinding_sensry_sy1xx_gpio
- dtbinding_sensry_sy1xx_i2c
- dtbinding_sensry_sy1xx_mac
- dtbinding_sensry_sy1xx_mdio
- dtbinding_sensry_sy1xx_pinctrl
- dtbinding_sensry_sy1xx_spi
- dtbinding_sensry_sy1xx_sys_timer
- dtbinding_sensry_sy1xx_trng
- dtbinding_sensry_sy1xx_uart

Sensylink Inc. (sensylink)

- dtbinding_sensylink_cht8315

Sequans Communications (sqn)

- dtbinding_sqn_gm02s
- dtbinding_sqn_hwspinlock

SHAKTI Processor Program (shakti)

- dtbinding_shakti_uart

Shanghai Awinic Technology Co., Ltd. (awinic)

- dtbinding_awinic_aw88298
- dtbinding_awinic_aw9523b
- dtbinding_awinic_aw9523b_gpio
- dtbinding_awinic_aw9523b_led

Sharp Corporation (sharp)

- dtbinding_sharp_ls0xx

Shenzhen Frida LCD Co., Ltd. (frida)

- dtbinding_frida_nt35510

Shenzhen Huiding Technology Co., Ltd. (goodix)

- dtbinding_goodix_gt911

Shenzhen Jinghua Displays Electronics Co., Ltd. (jhd)

- dtbinding_jhd_jhd1313

Shenzhen Titan Micro Electronics Co., Ltd. (titanmec)

- dtbinding_titanmec_tm1637
- dtbinding_titanmec_tm6605

Shenzhen Xptek Technology Co., Ltd (xptek)

- dtbinding_xptek_xpt2046

Siemens AG (siemens)

- dtbinding_siemens_ivshmem_eth

Sierra Wireless (swir)

- dtbinding_swir_hl7800
- dtbinding_swir_hl7812
- dtbinding_swir_hl7812_gnss
- dtbinding_swir_hl7812_offload
- dtbinding_swir_hl78xx
- dtbinding_swir_hl78xx_gnss
- dtbinding_swir_hl78xx_offload

SiFive, Inc. (sifive)

- dtbinding_sifive_clic_draft
- dtbinding_sifive_clint0
- dtbinding_sifive_dtim0
- dtbinding_sifive_e24
- dtbinding_sifive_e31
- dtbinding_sifive_e51
- dtbinding_sifive_fu740_c000_ddr
- dtbinding_sifive_gpio0
- dtbinding_sifive_i2c0
- dtbinding_sifive_pinctrl
- dtbinding_sifive_plic_1.0.0
- dtbinding_sifive_pwm0
- dtbinding_sifive_s7
- dtbinding_sifive_spi0
- dtbinding_sifive_u54
- dtbinding_sifive_uart0
- dtbinding_sifive_wdt

SiFli Technologies (sifli)

- dtbinding_sifli_sf32lb_aon
- dtbinding_sifli_sf32lb_atim
- dtbinding_sifli_sf32lb_atim_pwm
- dtbinding_sifli_sf32lb_audcodec
- dtbinding_sifli_sf32lb_cfg
- dtbinding_sifli_sf32lb_crc
- dtbinding_sifli_sf32lb_crypto
- dtbinding_sifli_sf32lb_dll
- dtbinding_sifli_sf32lb_dmac
- dtbinding_sifli_sf32lb_efuse
- dtbinding_sifli_sf32lb_gpadc
- dtbinding_sifli_sf32lb_gpio
- dtbinding_sifli_sf32lb_gpio_parent
- dtbinding_sifli_sf32lb_gpt_pwm
- dtbinding_sifli_sf32lb_gptim
- dtbinding_sifli_sf32lb_hxt48
- dtbinding_sifli_sf32lb_i2c
- dtbinding_sifli_sf32lb_lcdc
- dtbinding_sifli_sf32lb_lcdc_mipi_dbi
- dtbinding_sifli_sf32lb_mailbox
- dtbinding_sifli_sf32lb_mpi_qspi_nor
- dtbinding_sifli_sf32lb_rcc
- dtbinding_sifli_sf32lb_rcc_clk
- dtbinding_sifli_sf32lb_rcc_rctl
- dtbinding_sifli_sf32lb_rtc
- dtbinding_sifli_sf32lb_rtc_backup
- dtbinding_sifli_sf32lb_spi
- dtbinding_sifli_sf32lb_trng
- dtbinding_sifli_sf32lb_tsen
- dtbinding_sifli_sf32lb_usart
- dtbinding_sifli_sf32lb_wdt
- dtbinding_sifli_sf32lb52x_ldo
- dtbinding_sifli_sf32lb52x_pinmux
- dtbinding_sifli_sf32lb52x_pmuc

Silergy Corp. (silergy)

- dtbinding_silergy_sy24561
- dtbinding_silergy_sy6974b

Silicon Laboratories (silabs)

- dtbinding_silabs_acmp
- dtbinding_silabs_bt_hci_efr32
- dtbinding_silabs_buram
- dtbinding_silabs_burtc_counter
- dtbinding_silabs_dbus_pinctrl
- dtbinding_silabs_efr32_ieee802154
- dtbinding_silabs_emu
- dtbinding_silabs_emu_temp
- dtbinding_silabs_eusart_spi
- dtbinding_silabs_eusart_uart
- dtbinding_silabs_exp_header
- dtbinding_silabs_gecko_adc
- dtbinding_silabs_gecko_burtc
- dtbinding_silabs_gecko_ethernet
- dtbinding_silabs_gecko_flash_controller
- dtbinding_silabs_gecko_gpio
- dtbinding_silabs_gecko_gpio_port
- dtbinding_silabs_gecko_i2c
- dtbinding_silabs_gecko_leuart
- dtbinding_silabs_gecko_pinctrl
- dtbinding_silabs_gecko_pwm
- dtbinding_silabs_gecko_rtcc
- dtbinding_silabs_gecko_semailbox
- dtbinding_silabs_gecko_stimer
- dtbinding_silabs_gecko_timer
- dtbinding_silabs_gecko_trng
- dtbinding_silabs_gecko_uart
- dtbinding_silabs_gecko_usart
- dtbinding_silabs_gecko_wdog
- dtbinding_silabs_gpcrc
- dtbinding_silabs_gpdma
- dtbinding_silabs_gpio
- dtbinding_silabs_gpio_port
- dtbinding_silabs_gspi
- dtbinding_silabs_hfxo
- dtbinding_silabs_i2c
- dtbinding_silabs_iadc
- dtbinding_silabs_ldma

- dtbinding_silabs_letimer_pwm
- dtbinding_silabs_protimer
- dtbinding_silabs_pti
- dtbinding_silabs_rtcc
- dtbinding_silabs_series_clock
- dtbinding_silabs_series_clock_output
- dtbinding_silabs_series0_cmu
- dtbinding_silabs_series0_hfxo
- dtbinding_silabs_series0_lfrco
- dtbinding_silabs_series0_lfxo
- dtbinding_silabs_series2_dcdc
- dtbinding_silabs_series2_flash_controller
- dtbinding_silabs_series2_hfrcodpll
- dtbinding_silabs_series2_hfrcoem23
- dtbinding_silabs_series2_letimer
- dtbinding_silabs_series2_lfrco
- dtbinding_silabs_series2_lfxo
- dtbinding_silabs_series2_radio
- dtbinding_silabs_series2_timer
- dtbinding_silabs_series3_adc
- dtbinding_silabs_series3_crypto
- dtbinding_silabs_series3_extmem
- dtbinding_silabs_si32_aes
- dtbinding_silabs_si32_ahb
- dtbinding_silabs_si32_apb
- dtbinding_silabs_si32_dma
- dtbinding_silabs_si32_flash_controller
- dtbinding_silabs_si32_gpio
- dtbinding_silabs_si32_pinctrl
- dtbinding_silabs_si32_pll
- dtbinding_silabs_si32_usart
- dtbinding_silabs_si7006
- dtbinding_silabs_si7055
- dtbinding_silabs_si7060
- dtbinding_silabs_si7210
- dtbinding_silabs_siwx91x_adc
- dtbinding_silabs_siwx91x_bt_hci
- dtbinding_silabs_siwx91x_clock_route
- dtbinding_silabs_siwx91x_cmu_aon

- dtbinding_silabs_siwx91x_cmu_hp
- dtbinding_silabs_siwx91x_cmu_ulp
- dtbinding_silabs_siwx91x_flash_controller
- dtbinding_silabs_siwx91x_gpio
- dtbinding_silabs_siwx91x_gpio_port
- dtbinding_silabs_siwx91x_gpio_uulp
- dtbinding_silabs_siwx91x_i2s
- dtbinding_silabs_siwx91x_nwp
- dtbinding_silabs_siwx91x_physical_clock
- dtbinding_silabs_siwx91x_pinctrl
- dtbinding_silabs_siwx91x_pll
- dtbinding_silabs_siwx91x_power_domain
- dtbinding_silabs_siwx91x_pwm
- dtbinding_silabs_siwx91x_qspi_memory
- dtbinding_silabs_siwx91x_rng
- dtbinding_silabs_siwx91x_rtc
- dtbinding_silabs_siwx91x_wdt
- dtbinding_silabs_siwx91x_wifi
- dtbinding_silabs_socpll
- dtbinding_silabs_sysrtc
- dtbinding_silabs_timer_counter
- dtbinding_silabs_timer_pwm
- dtbinding_silabs_udma
- dtbinding_silabs_usart_spi
- dtbinding_silabs_usart_uart
- dtbinding_silabs_vdac
- dtbinding_silabs_wdog

SIMCom Wireless Solutions Co., LTD (simcom)

- dtbinding_simcom_a76xx
- dtbinding_simcom_sim7080

Sino Wealth Electronic Ltd (sinowealth)

- dtbinding_sinowealth_sh1106_i2c
- dtbinding_sinowealth_sh1106_spi
- dtbinding_sinowealth_sh1107_spi
- dtbinding_sinowealth_sh1107_i2c
- dtbinding_sinowealth_sh1122_i2c
- dtbinding_sinowealth_sh1122_mipi_dbi

Sitronix Technology Corporation (sitronix)

- dtbinding_sitronix_cf1133
- dtbinding_sitronix_st7305
- dtbinding_sitronix_st7306
- dtbinding_sitronix_st75256
- dtbinding_sitronix_st7567_mipi_dbi
- dtbinding_sitronix_st7567_i2c
- dtbinding_sitronix_st7586s
- dtbinding_sitronix_st7701
- dtbinding_sitronix_st7735r
- dtbinding_sitronix_st7789v
- dtbinding_sitronix_st7796s

Skyworks Solutions, Inc. (skyworks)

- dtbinding_skyworks_sky13317
- dtbinding_skyworks_sky13348
- dtbinding_skyworks_sky13351

Smart Battery System (sbs)

- dtbinding_sbs_default_sbs_gauge
- dtbinding_sbs_sbs_charger
- dtbinding_sbs_sbs_gauge
- dtbinding_sbs_sbs_gauge_new_api

Socionext Inc. (socionext)

- dtbinding_socionext_dpu

Soldered Electronics (solderedelectronics)

- dtbinding_solderedelectronics_easyc_connector

Solomon Systech Limited (solomon)

- dtbinding_solomon_ssd1305_spi
- dtbinding_solomon_ssd1305_i2c
- dtbinding_solomon_ssd1306_spi
- dtbinding_solomon_ssd1306_i2c
- dtbinding_solomon_ssd1306b_i2c
- dtbinding_solomon_ssd1306b_spi
- dtbinding_solomon_ssd1309_i2c
- dtbinding_solomon_ssd1309_spi
- dtbinding_solomon_ssd1315_i2c
- dtbinding_solomon_ssd1315_spi

- dtbinding_solomon_ssd1320_i2c
- dtbinding_solomon_ssd1320_mipi_dbi
- dtbinding_solomon_ssd1322
- dtbinding_solomon_ssd1325_mipi_dbi
- dtbinding_solomon_ssd1325_i2c
- dtbinding_solomon_ssd1327_mipi_dbi
- dtbinding_solomon_ssd1327_i2c
- dtbinding_solomon_ssd1331
- dtbinding_solomon_ssd1351
- dtbinding_solomon_ssd1357
- dtbinding_solomon_ssd1363_mipi_dbi
- dtbinding_solomon_ssd1363_i2c
- dtbinding_solomon_ssd1608
- dtbinding_solomon_ssd1673
- dtbinding_solomon_ssd1675a
- dtbinding_solomon_ssd1680
- dtbinding_solomon_ssd1681
- dtbinding_solomon_ssd1683

Sony Corporation (sony)

- dtbinding_sony_imx219
- dtbinding_sony_imx335

SparkFun Electronics (sparkfun)

- dtbinding_sparkfun_micromod_gpio
- dtbinding_sparkfun_pro_micro_gpio
- dtbinding_sparkfun_serlcd

SpinalHDL (spinalhdl)

- dtbinding_spinalhdl_vexiiriscv
- dtbinding_spinalhdl_vexriscv

Standard Microsystems Corporation (smsc)

- dtbinding_smsc_lan91c111
- dtbinding_smsc_lan91c111_mdio
- dtbinding_smsc_lan9220

StarChips Technology Inc. (sct)

- dtbinding_sct_sct2024

STMicroelectronics (st)

- dtbinding_st_dsi_lcd_qsh_030_connector
- dtbinding_st_dvp_cam_zif_30_connector
- dtbinding_st_hci_spi_v1
- dtbinding_st_hci_spi_v2
- dtbinding_st_hci_stm32wb0
- dtbinding_st_hci_stm32wba
- dtbinding_st_hts221_i2c
- dtbinding_st_hts221_spi
- dtbinding_st_i3g4250d
- dtbinding_st_iis2dh_i2c
- dtbinding_st_iis2dh_spi
- dtbinding_st_iis2dlpc_spi
- dtbinding_st_iis2dlpc_i2c
- dtbinding_st_iis2iclx_i2c
- dtbinding_st_iis2iclx_spi
- dtbinding_st_iis2mdc_i2c
- dtbinding_st_iis2mdc_spi
- dtbinding_st_iis328dq_spi
- dtbinding_st_iis328dq_i2c
- dtbinding_st_iis3dhhc_spi
- dtbinding_st_iis3dwb
- dtbinding_st_ilps22qs_spi
- dtbinding_st_ilps22qs_i3c
- dtbinding_st_ilps22qs_i2c
- dtbinding_st_ism330dhcx_spi
- dtbinding_st_ism330dhcx_i2c
- dtbinding_st_ism6hg256x_i3c
- dtbinding_st_ism6hg256x_i2c
- dtbinding_st_ism6hg256x_spi
- dtbinding_st_lis2de12_spi
- dtbinding_st_lis2de12_i2c
- dtbinding_st_lis2dh_spi
- dtbinding_st_lis2dh_i2c
- dtbinding_st_lis2dh12_i2c
- dtbinding_st_lis2ds12_i2c
- dtbinding_st_lis2ds12_spi
- dtbinding_st_lis2du12_i2c
- dtbinding_st_lis2du12_spi

- dtbinding_st_lis2dux12_i2c
- dtbinding_st_lis2dux12_spi
- dtbinding_st_lis2duxs12_i2c
- dtbinding_st_lis2duxs12_spi
- dtbinding_st_lis2dw12_i2c
- dtbinding_st_lis2dw12_spi
- dtbinding_st_lis2mdl_i2c
- dtbinding_st_lis2mdl_spi
- dtbinding_st_lis3dh_i2c
- dtbinding_st_lis3mdl_magn
- dtbinding_st_lps22df_i2c
- dtbinding_st_lps22df_i3c
- dtbinding_st_lps22df_spi
- dtbinding_st_lps22hb_press
- dtbinding_st_lps22hh_i2c
- dtbinding_st_lps22hh_i3c
- dtbinding_st_lps22hh_spi
- dtbinding_st_lps25hb_press
- dtbinding_st_lps28dfw_i3c
- dtbinding_st_lps28dfw_i2c
- dtbinding_st_lsm303agr_accel_spi
- dtbinding_st_lsm303agr_accel_i2c
- dtbinding_st_lsm303dlhc_accel
- dtbinding_st_lsm303dlhc_magn
- dtbinding_st_lsm6ds0
- dtbinding_st_lsm6dsl_spi
- dtbinding_st_lsm6dsl_i2c
- dtbinding_st_lsm6dso_i2c
- dtbinding_st_lsm6dso_spi
- dtbinding_st_lsm6dso16is_spi
- dtbinding_st_lsm6dso16is_i2c
- dtbinding_st_lsm6dso32_i2c
- dtbinding_st_lsm6dso32_spi
- dtbinding_st_lsm6dsv16x_i3c
- dtbinding_st_lsm6dsv16x_i2c
- dtbinding_st_lsm6dsv16x_spi
- dtbinding_st_lsm6dsv320x_i3c
- dtbinding_st_lsm6dsv320x_spi
- dtbinding_st_lsm6dsv320x_i2c

- dtbinding_st_lsm6dsv32x_i3c
- dtbinding_st_lsm6dsv32x_i2c
- dtbinding_st_lsm6dsv32x_spi
- dtbinding_st_lsm6dsv80x_spi
- dtbinding_st_lsm6dsv80x_i3c
- dtbinding_st_lsm6dsv80x_i2c
- dtbinding_st_lsm9ds0_gyro_i2c
- dtbinding_st_lsm9ds0_mfd_i2c
- dtbinding_st_lsm9ds1
- dtbinding_st_lsm9ds1_mag
- dtbinding_st_m2_memory_connector
- dtbinding_st_m95p32_mspi
- dtbinding_st_mbox_stm32_hsem
- dtbinding_st_mfxstm32l152
- dtbinding_st_mipid02
- dtbinding_st_mpxxdtyy_i2s
- dtbinding_st_neochrom_gpu2d
- dtbinding_st_nor_mspi
- dtbinding_st_psram_device_mspi
- dtbinding_st_st67w611m1
- dtbinding_st_st87mxx
- dtbinding_st_stm32_adc
- dtbinding_st_stm32_aes
- dtbinding_st_stm32_backup_sram
- dtbinding_st_stm32_bbram
- dtbinding_st_stm32_bdma
- dtbinding_st_stm32_bsec
- dtbinding_st_stm32_bxcan
- dtbinding_st_stm32_ccm
- dtbinding_st_stm32_clock_mco
- dtbinding_st_stm32_clock_mux
- dtbinding_st_stm32_comp
- dtbinding_st_stm32_counter
- dtbinding_st_stm32_crc
- dtbinding_st_stm32_cryp
- dtbinding_st_stm32_dac
- dtbinding_st_stm32_dcmi
- dtbinding_st_stm32_dcmipp
- dtbinding_st_stm32_dfsdm

- dtbinding_st_stm32_dfsdm_dmic
- dtbinding_st_stm32_digi_temp
- dtbinding_st_stm32_dma
- dtbinding_st_stm32_dma_v1
- dtbinding_st_stm32_dma_v2
- dtbinding_st_stm32_dma_v2bis
- dtbinding_st_stm32_dma2d
- dtbinding_st_stm32_dmamux
- dtbinding_st_stm32_dualreg_pwr
- dtbinding_st_stm32_eeprom
- dtbinding_st_stm32_ethernet
- dtbinding_st_stm32_exti
- dtbinding_st_stm32_fdcan
- dtbinding_st_stm32_flash_controller
- dtbinding_st_stm32_fmc
- dtbinding_st_stm32_fmc_mipi_dbi
- dtbinding_st_stm32_fmc_nor_psram
- dtbinding_st_stm32_fmc_sdram
- dtbinding_st_stm32_gpio
- dtbinding_st_stm32_hash
- dtbinding_st_stm32_hse_clock
- dtbinding_st_stm32_hsem_mailbox
- dtbinding_st_stm32_hsi48_clock
- dtbinding_st_stm32_i2c_v1
- dtbinding_st_stm32_i2c_v2
- dtbinding_st_stm32_i2s
- dtbinding_st_stm32_i3c
- dtbinding_st_stm32_iocell
- dtbinding_st_stm32_ipcc_mailbox
- dtbinding_st_stm32_jpeg
- dtbinding_st_stm32_lptim
- dtbinding_st_stm32_lpuart
- dtbinding_st_stm32_lse_clock
- dtbinding_st_stm32_ltdc
- dtbinding_st_stm32_mdf
- dtbinding_st_stm32_mipi_dsi
- dtbinding_st_stm32_msi_clock
- dtbinding_st_stm32_npu
- dtbinding_st_stm32_npu_cache

- dtbinding_st_stm32_nv_flash
- dtbinding_st_stm32_nvm_otp
- dtbinding_st_stm32_opamp
- dtbinding_st_stm32_ospi
- dtbinding_st_stm32_ospi_controller
- dtbinding_st_stm32_ospi_nor
- dtbinding_st_stm32_ospi_psram
- dtbinding_st_stm32_otgfs
- dtbinding_st_stm32_otghs
- dtbinding_st_stm32_pinctrl
- dtbinding_st_stm32_pwm
- dtbinding_st_stm32_pwr
- dtbinding_st_stm32_pwr_wkupctrl
- dtbinding_st_stm32_qdec
- dtbinding_st_stm32_qspi
- dtbinding_st_stm32_qspi_controller
- dtbinding_st_stm32_qspi_nor
- dtbinding_st_stm32_rcc
- dtbinding_st_stm32_rcc_rctl
- dtbinding_st_stm32_rng
- dtbinding_st_stm32_rng_noirq
- dtbinding_st_stm32_rtc
- dtbinding_st_stm32_sai
- dtbinding_st_stm32_sdmmc
- dtbinding_st_stm32_smbus
- dtbinding_st_stm32_spi
- dtbinding_st_stm32_spi_fifo
- dtbinding_st_stm32_spi_host_cmd
- dtbinding_st_stm32_spi_subghz
- dtbinding_st_stm32_temp
- dtbinding_st_stm32_temp_cal
- dtbinding_st_stm32_timers
- dtbinding_st_stm32_tsc
- dtbinding_st_stm32_uart
- dtbinding_st_stm32_ucpd
- dtbinding_st_stm32_usart
- dtbinding_st_stm32_usb
- dtbinding_st_stm32_usbphyc
- dtbinding_st_stm32_vbat

- dtbinding_st_stm32_vddcore
- dtbinding_st_stm32_venc
- dtbinding_st_stm32_vref
- dtbinding_st_stm32_vrefbuf
- dtbinding_st_stm32_watchdog
- dtbinding_st_stm32_window_watchdog
- dtbinding_st_stm32_xspi
- dtbinding_st_stm32_xspi_controller
- dtbinding_st_stm32_xspi_nor
- dtbinding_st_stm32_xspi_psram
- dtbinding_st_stm32_xspim
- dtbinding_st_stm32c0_hsi_clock
- dtbinding_st_stm32c0_temp_cal
- dtbinding_st_stm32c5_flash_controller
- dtbinding_st_stm32c5_rcc
- dtbinding_st_stm32c5_xsik_clock
- dtbinding_st_stm32f0_pll_clock
- dtbinding_st_stm32f0_rcc
- dtbinding_st_stm32f1_adc
- dtbinding_st_stm32f1_clock_mco
- dtbinding_st_stm32f1_flash_controller
- dtbinding_st_stm32f1_pinctrl
- dtbinding_st_stm32f1_pll_clock
- dtbinding_st_stm32f1_pwr_wkupctrl
- dtbinding_st_stm32f100_pll_clock
- dtbinding_st_stm32f105_pll_clock
- dtbinding_st_stm32f105_pll2_clock
- dtbinding_st_stm32f2_flash_controller
- dtbinding_st_stm32f4_adc
- dtbinding_st_stm32f4_flash_controller
- dtbinding_st_stm32f4_fsotg
- dtbinding_st_stm32f4_nv_flash
- dtbinding_st_stm32f4_rcc
- dtbinding_st_stm32f7_flash_controller
- dtbinding_st_stm32f7_pwr_wkupctrl
- dtbinding_st_stm32f7_usbphyc
- dtbinding_st_stm32fx_pll_clock
- dtbinding_st_stm32g0_exti
- dtbinding_st_stm32g0_flash_controller

- dtbinding_st_stm32g0_hsi_clock
- dtbinding_st_stm32g0_pll_clock
- dtbinding_st_stm32g4_comp
- dtbinding_st_stm32g4_flash_controller
- dtbinding_st_stm32g4_opamp
- dtbinding_st_stm32g4_pll_clock
- dtbinding_st_stm32h5_ethernet
- dtbinding_st_stm32h5_iocell
- dtbinding_st_stm32h5_pinctrl
- dtbinding_st_stm32h5_pll_clock
- dtbinding_st_stm32h5_rcc
- dtbinding_st_stm32h7_comp
- dtbinding_st_stm32h7_ethernet
- dtbinding_st_stm32h7_fdcan
- dtbinding_st_stm32h7_flash_controller
- dtbinding_st_stm32h7_fmc
- dtbinding_st_stm32h7_hsi_clock
- dtbinding_st_stm32h7_i2s
- dtbinding_st_stm32h7_pll_clock
- dtbinding_st_stm32h7_pwr
- dtbinding_st_stm32h7_rcc
- dtbinding_st_stm32h7_spi
- dtbinding_st_stm32h7rs_exti
- dtbinding_st_stm32h7rs_pll_clock
- dtbinding_st_stm32h7rs_pwr
- dtbinding_st_stm32h7rs_rcc
- dtbinding_st_stm32l0_hsi_clock
- dtbinding_st_stm32l0_msi_clock
- dtbinding_st_stm32l0_nv_flash
- dtbinding_st_stm32l0_pll_clock
- dtbinding_st_stm32l4_aes
- dtbinding_st_stm32l4_flash_controller
- dtbinding_st_stm32l4_pll_clock
- dtbinding_st_stm32l5_flash_controller
- dtbinding_st_stm32mp1_rcc
- dtbinding_st_stm32mp13_cpu_clock_mux
- dtbinding_st_stm32mp13_ethernet
- dtbinding_st_stm32mp13_pll_clock
- dtbinding_st_stm32mp2_gpio

- dtbinding_st_stm32mp2_rcc
- dtbinding_st_stm32n6_adc
- dtbinding_st_stm32n6_cpu_clock_mux
- dtbinding_st_stm32n6_ethernet
- dtbinding_st_stm32n6_hse_clock
- dtbinding_st_stm32n6_ic_clock_mux
- dtbinding_st_stm32n6_msi_clock
- dtbinding_st_stm32n6_otghs
- dtbinding_st_stm32n6_pinctrl
- dtbinding_st_stm32n6_pll_clock
- dtbinding_st_stm32n6_ramcfg
- dtbinding_st_stm32n6_rcc
- dtbinding_st_stm32n6_usbphyc
- dtbinding_st_stm32u0_pll_clock
- dtbinding_st_stm32u3_flash_controller
- dtbinding_st_stm32u3_msi_clock
- dtbinding_st_stm32u5_dma
- dtbinding_st_stm32u5_lptim
- dtbinding_st_stm32u5_mipi_dsi
- dtbinding_st_stm32u5_msi_clock
- dtbinding_st_stm32u5_otghs_phy
- dtbinding_st_stm32u5_pll_clock
- dtbinding_st_stm32u5_pwr
- dtbinding_st_stm32u5_rcc
- dtbinding_st_stm32wb_flash_controller
- dtbinding_st_stm32wb_pll_clock
- dtbinding_st_stm32wb_rcc
- dtbinding_st_stm32wb_rf
- dtbinding_st_stm32wb0_adc
- dtbinding_st_stm32wb0_flash_controller
- dtbinding_st_stm32wb0_gpio_intc
- dtbinding_st_stm32wb0_lsi_clock
- dtbinding_st_stm32wb0_pwr
- dtbinding_st_stm32wb0_radio_timer
- dtbinding_st_stm32wb0_rcc
- dtbinding_st_stm32wba_flash_controller
- dtbinding_st_stm32wba_hse_clock
- dtbinding_st_stm32wba_ieee802154
- dtbinding_st_stm32wba_pka

- dtbinding_st_stm32wba_pll_clock
- dtbinding_st_stm32wba_pwr
- dtbinding_st_stm32wba_radio
- dtbinding_st_stm32wba_rcc
- dtbinding_st_stm32wl_hse_clock
- dtbinding_st_stm32wl_rcc
- dtbinding_st_stm32wl_subghz_radio
- dtbinding_st_stm32wl3_rcc
- dtbinding_st_stmod_plus_connector
- dtbinding_st_stmpe1600
- dtbinding_st_stmpe811
- dtbinding_st_stts22h_i2c
- dtbinding_st_stts751_i2c
- dtbinding_st_vl53l0x
- dtbinding_st_vl53l1x
- dtbinding_st_vl53l5cx

Synaptics Inc. (syna)

- dtbinding_syna_sr100_clock
- dtbinding_syna_sr100_pinctrl
- dtbinding_syna_sr100_reset

Synopsys, Inc. (snps)

- dtbinding_snps_arc_iot_sysconf
- dtbinding_snps_arc_timer
- dtbinding_snps_arcem
- dtbinding_snps_archs_ici
- dtbinding_snps_archs_idu_intc
- dtbinding_snps_arcv2_intc
- dtbinding_snps_av5rhx
- dtbinding_snps_av5rmx
- dtbinding_snps_creg_gpio
- dtbinding_snps_designware_dma
- dtbinding_snps_designware_dma_axi
- dtbinding_snps_designware_ethernet
- dtbinding_snps_designware_gpio
- dtbinding_snps_designware_i2c
- dtbinding_snps_designware_i3c
- dtbinding_snps_designware_intc
- dtbinding_snps_designware_spi

- dtbinding_snps_designware_ssi
- dtbinding_snps_designware_usb
- dtbinding_snps_designware_watchdog
- dtbinding_snps_dw_apb_uart
- dtbinding_snps_dw_timers
- dtbinding_snps_dwc2
- dtbinding_snps_dwcxgmac
- dtbinding_snps_dwcxgmac_mdio
- dtbinding_snps_dwmac
- dtbinding_snps_dwmac_mdio
- dtbinding_snps_dwmac_ptp_clock
- dtbinding_snps_emsdp_pinctrl
- dtbinding_snps_ethernet_cyclonev
- dtbinding_snps_hostlink_uart
- dtbinding_snps_nsim_uart

Synopsys, Inc. (formerly ARC International PLC) (arc)

- dtbinding_arc_dccm
- dtbinding_arc_iccm
- dtbinding_arc_xccm
- dtbinding_arc_yccm

TBS Technologies (tbs)

- dtbinding_tbs_crsf

TDK Corporation. (tdk)

- dtbinding_tdk_ntcgxx3jx103x

Telink Semiconductor (telink)

- dtbinding_telink_b91
- dtbinding_telink_b91_adc
- dtbinding_telink_b91_flash_controller
- dtbinding_telink_b91_gpio
- dtbinding_telink_b91_i2c
- dtbinding_telink_b91_pinctrl
- dtbinding_telink_b91_power
- dtbinding_telink_b91_pwm
- dtbinding_telink_b91_spi
- dtbinding_telink_b91_trng
- dtbinding_telink_b91_uart

- dtbinding_telink_b91_zb

Telit Cinterion (telit)

- dtbinding_telit_le910c1tx
- dtbinding_telit_lex10q1
- dtbinding_telit_me310g1
- dtbinding_telit_me910g1

Texas Instruments (ti)

- dtbinding_ti_adc081c021
- dtbinding_ti_adc081c027
- dtbinding_ti_adc101c021
- dtbinding_ti_adc101c027
- dtbinding_ti_adc121c021
- dtbinding_ti_adc121c027
- dtbinding_ti_adc124s021
- dtbinding_ti_ads1013
- dtbinding_ti_ads1014
- dtbinding_ti_ads1015
- dtbinding_ti_ads1112
- dtbinding_ti_ads1113
- dtbinding_ti_ads1114
- dtbinding_ti_ads1115
- dtbinding_ti_ads1118
- dtbinding_ti_ads1119
- dtbinding_ti_ads114s06
- dtbinding_ti_ads114s08
- dtbinding_ti_ads1220
- dtbinding_ti_ads124s06
- dtbinding_ti_ads124s08
- dtbinding_ti_ads1262
- dtbinding_ti_ads1263
- dtbinding_ti_ads131m02
- dtbinding_ti_ads1x4s0x_gpio
- dtbinding_ti_ads7052
- dtbinding_ti_ads7828
- dtbinding_ti_ads7830
- dtbinding_ti_ads7950
- dtbinding_ti_ads7951
- dtbinding_ti_ads7952

- dtbinding_ti_ads7953
- dtbinding_ti_ads7954
- dtbinding_ti_ads7955
- dtbinding_ti_ads7956
- dtbinding_ti_ads7957
- dtbinding_ti_ads7958
- dtbinding_ti_ads7959
- dtbinding_ti_ads7960
- dtbinding_ti_ads7961
- dtbinding_ti_am26_rtitimer
- dtbinding_ti_am3352_ecap
- dtbinding_ti_am3352_ehrpwm
- dtbinding_ti_am335x_adc
- dtbinding_ti_am654_sdhci
- dtbinding_ti_am654_timer
- dtbinding_ti_boosterpack_header
- dtbinding_ti_bq24190
- dtbinding_ti_bq24295
- dtbinding_ti_bq24296
- dtbinding_ti_bq24296m
- dtbinding_ti_bq24297
- dtbinding_ti_bq24298
- dtbinding_ti_bq25180
- dtbinding_ti_bq25186
- dtbinding_ti_bq25188
- dtbinding_ti_bq25713
- dtbinding_ti_bq274xx
- dtbinding_ti_bq27z746
- dtbinding_ti_bq32002
- dtbinding_ti_bq40z50
- dtbinding_ti_cc1200
- dtbinding_ti_cc13xx_cc26xx_adc
- dtbinding_ti_cc13xx_cc26xx_flash_controller
- dtbinding_ti_cc13xx_cc26xx_gpio
- dtbinding_ti_cc13xx_cc26xx_i2c
- dtbinding_ti_cc13xx_cc26xx_ieee802154
- dtbinding_ti_cc13xx_cc26xx_ieee802154_subghz
- dtbinding_ti_cc13xx_cc26xx_pinctrl
- dtbinding_ti_cc13xx_cc26xx_radio

- dtbinding_ti_cc13xx_cc26xx_rtc_timer
- dtbinding_ti_cc13xx_cc26xx_spi
- dtbinding_ti_cc13xx_cc26xx_timer
- dtbinding_ti_cc13xx_cc26xx_timer_pwm
- dtbinding_ti_cc13xx_cc26xx_trng
- dtbinding_ti_cc13xx_cc26xx_uart
- dtbinding_ti_cc13xx_cc26xx_watchdog
- dtbinding_ti_cc23x0_adc
- dtbinding_ti_cc23x0_aes
- dtbinding_ti_cc23x0_ccfg_flash
- dtbinding_ti_cc23x0_dma
- dtbinding_ti_cc23x0_flash_controller
- dtbinding_ti_cc23x0_gpio
- dtbinding_ti_cc23x0_i2c
- dtbinding_ti_cc23x0_lf_xosc
- dtbinding_ti_cc23x0_lgpt
- dtbinding_ti_cc23x0_lgpt_pwm
- dtbinding_ti_cc23x0_pinctrl
- dtbinding_ti_cc23x0_rtc
- dtbinding_ti_cc23x0_spi
- dtbinding_ti_cc23x0_systim_timer
- dtbinding_ti_cc23x0_uart
- dtbinding_ti_cc23x0_wdt
- dtbinding_ti_cc2520
- dtbinding_ti_cc32xx_adc
- dtbinding_ti_cc32xx_gpio
- dtbinding_ti_cc32xx_i2c
- dtbinding_ti_cc32xx_pinctrl
- dtbinding_ti_cc32xx_uart
- dtbinding_ti_cc32xx_watchdog
- dtbinding_ti_cdce913
- dtbinding_ti_cdce925
- dtbinding_ti_cdce937
- dtbinding_ti_cdce949
- dtbinding_ti_control_module
- dtbinding_ti_dac161s997
- dtbinding_ti_dac43508
- dtbinding_ti_dac43608
- dtbinding_ti_dac5311

- dtbinding_ti_dac53508
- dtbinding_ti_dac53608
- dtbinding_ti_dac60508
- dtbinding_ti_dac6311
- dtbinding_ti_dac63508
- dtbinding_ti_dac70508
- dtbinding_ti_dac7311
- dtbinding_ti_dac80508
- dtbinding_ti_dac8311
- dtbinding_ti_dac8411
- dtbinding_ti_dacx0501
- dtbinding_ti_davinci_gpio
- dtbinding_ti_davinci_gpio_nexus
- dtbinding_ti_dp83825
- dtbinding_ti_dp83867
- dtbinding_ti_drv2605
- dtbinding_ti_drv84xx
- dtbinding_ti_fdc1004
- dtbinding_ti_fdc2x1x
- dtbinding_ti_hdc
- dtbinding_ti_hdc2010
- dtbinding_ti_hdc2021
- dtbinding_ti_hdc2022
- dtbinding_ti_hdc2080
- dtbinding_ti_hdc302x
- dtbinding_ti_ina219
- dtbinding_ti_ina226
- dtbinding_ti_ina228
- dtbinding_ti_ina230
- dtbinding_ti_ina232
- dtbinding_ti_ina236
- dtbinding_ti_ina237
- dtbinding_ti_ina3221
- dtbinding_ti_ina7xx
- dtbinding_ti_j7_rti_wdt
- dtbinding_ti_k2g_sci
- dtbinding_ti_k2g_sci_clk
- dtbinding_ti_k3_pinctrl
- dtbinding_ti_k3_rtc_counter

- dtbinding_ti_lmp90077
- dtbinding_ti_lmp90078
- dtbinding_ti_lmp90079
- dtbinding_ti_lmp90080
- dtbinding_ti_lmp90097
- dtbinding_ti_lmp90098
- dtbinding_ti_lmp90099
- dtbinding_ti_lmp90100
- dtbinding_ti_lmp90xxx_gpio
- dtbinding_ti_lp3943
- dtbinding_ti_lp5009
- dtbinding_ti_lp5012
- dtbinding_ti_lp5018
- dtbinding_ti_lp5024
- dtbinding_ti_lp5030
- dtbinding_ti_lp5036
- dtbinding_ti_lp5562
- dtbinding_ti_lp5569
- dtbinding_ti_lp5860
- dtbinding_ti_lp5861
- dtbinding_ti_lp5862
- dtbinding_ti_lp5864
- dtbinding_ti_lp5866
- dtbinding_ti_lp5868
- dtbinding_ti_msp432p4xx_uart
- dtbinding_ti_mspm0_adc12
- dtbinding_ti_mspm0_aes
- dtbinding_ti_mspm0_canfd
- dtbinding_ti_mspm0_clk
- dtbinding_ti_mspm0_comparator
- dtbinding_ti_mspm0_dac
- dtbinding_ti_mspm0_dma
- dtbinding_ti_mspm0_flash_controller
- dtbinding_ti_mspm0_gpio
- dtbinding_ti_mspm0_osc
- dtbinding_ti_mspm0_pinctrl
- dtbinding_ti_mspm0_pll
- dtbinding_ti_mspm0_rtc
- dtbinding_ti_mspm0_spi

- dtbinding_ti_mspm0_timer
- dtbinding_ti_mspm0_timer_counter
- dtbinding_ti_mspm0_timer_pwm
- dtbinding_ti_mspm0_trng
- dtbinding_ti_mspm0_uart
- dtbinding_ti_mspm0_vref
- dtbinding_ti_mspm0_watchdog
- dtbinding_ti_omap_i2c
- dtbinding_ti_omap_mailbox
- dtbinding_ti_omap_mcspi
- dtbinding_ti_opt3001
- dtbinding_ti_opt3004
- dtbinding_ti_pcm1681_spi
- dtbinding_ti_pcm1681_i2c
- dtbinding_ti_sci_pm_domain
- dtbinding_ti_secure_proxy
- dtbinding_ti_sn74hc595
- dtbinding_ti_stellaris_ethernet
- dtbinding_ti_stellaris_flash_controller
- dtbinding_ti_stellaris_gpio
- dtbinding_ti_stellaris_uart
- dtbinding_ti_tas2563
- dtbinding_ti_tas6422dac
- dtbinding_ti_tca6424a
- dtbinding_ti_tca9538
- dtbinding_ti_tca9544a
- dtbinding_ti_tca9546a
- dtbinding_ti_tca9548a
- dtbinding_ti_tca9554
- dtbinding_ti_tcan4x5x
- dtbinding_ti_tdp2004
- dtbinding_ti_tiva_c_pinctrl
- dtbinding_ti_tiva_c_uart
- dtbinding_ti_tla2021
- dtbinding_ti_tla2022
- dtbinding_ti_tla2024
- dtbinding_ti_tla2528
- dtbinding_ti_tla2528_adc
- dtbinding_ti_tla2528_gpio

- dtbinding_ti_tlc59108
- dtbinding_ti_tlc5971
- dtbinding_ti_tlc59731
- dtbinding_ti_tlv320aic26
- dtbinding_ti_tlv320aic3110
- dtbinding_ti_tlv320dac
- dtbinding_ti_tmag3001
- dtbinding_ti_tmag5170
- dtbinding_ti_tmag5273
- dtbinding_ti_tmp007
- dtbinding_ti_tmp1075
- dtbinding_ti_tmp108
- dtbinding_ti_tmp112
- dtbinding_ti_tmp114
- dtbinding_ti_tmp11x
- dtbinding_ti_tmp11x_eeprom
- dtbinding_ti_tmp435
- dtbinding_ti_tmp451
- dtbinding_ti_tps382x
- dtbinding_ti_tps55287
- dtbinding_ti_tps6287x
- dtbinding_ti_vim

Trasna (trasna)

- dtbinding_trasna_lexi_r10

u-blox (u-blox)

- dtbinding_u_blox_f9p
- dtbinding_u_blox_lara_r6
- dtbinding_u_blox_m10
- dtbinding_u_blox_m8
- dtbinding_u_blox_sara_r4
- dtbinding_u_blox_sara_r5

UltraChip Inc. (ultrachip)

- dtbinding_ultrachip_uc8151d
- dtbinding_ultrachip_uc8175
- dtbinding_ultrachip_uc8176
- dtbinding_ultrachip_uc8179
- dtbinding_ultrachip_uc8253

Virtual I/O Device Specification, developed by the OASIS consortium (virtio)

- dtbinding_virtio_blk
- dtbinding_virtio_console
- dtbinding_virtio_device4
- dtbinding_virtio_gpio
- dtbinding_virtio_i2c
- dtbinding_virtio_input
- dtbinding_virtio_mmio
- dtbinding_virtio_net
- dtbinding_virtio_pci
- dtbinding_virtio_spi

Vishay Intertechnology, Inc (vishay)

- dtbinding_vishay_vcnl36825t
- dtbinding_vishay_vcnl4040
- dtbinding_vishay_veml6031
- dtbinding_vishay_veml6046
- dtbinding_vishay_veml7700
- dtbinding_vishay_vs1838b

Waveshare Electronics (waveshare)

- dtbinding_waveshare_dsi2dpi

WeAct Studio (weact)

- dtbinding_weact_dcmi_camera_connector

Winbond Electronics corp. (winbond)

- dtbinding_winbond_w25q512nw

WinChipHead (wch)

- dtbinding_wch_00x_afio
- dtbinding_wch_20x_30x_afio
- dtbinding_wch_adc
- dtbinding_wch_adtm
- dtbinding_wch_adtm_pwm
- dtbinding_wch_afio
- dtbinding_wch_ch32_rcc_rctl
- dtbinding_wch_ch32h41x_pll_clock
- dtbinding_wch_ch32v_flash_controller
- dtbinding_wch_ch32v00x_hse_clock
- dtbinding_wch_ch32v00x_hsi_clock

- dtbinding_wch_ch32v00x_pll_clock
- dtbinding_wch_ch32v20x_30x_pll_clock
- dtbinding_wch_ch570_pinctrl
- dtbinding_wch_ch5xx_comparator
- dtbinding_wch_ch5xx_gpio
- dtbinding_wch_ch5xx_uart
- dtbinding_wch_ch9120
- dtbinding_wch_ch9350l
- dtbinding_wch_esig
- dtbinding_wch_ethernet
- dtbinding_wch_exti
- dtbinding_wch_gpio
- dtbinding_wch_gptm
- dtbinding_wch_gptm_pwm
- dtbinding_wch_h41x_afio
- dtbinding_wch_i2c
- dtbinding_wch_iwdg
- dtbinding_wch_pfic
- dtbinding_wch_qingke_v2
- dtbinding_wch_qingke_v3c
- dtbinding_wch_qingke_v3f
- dtbinding_wch_qingke_v4b
- dtbinding_wch_qingke_v4c
- dtbinding_wch_qingke_v4f
- dtbinding_wch_qingke_v5f
- dtbinding_wch_rcc
- dtbinding_wch_rng
- dtbinding_wch_sdi_console
- dtbinding_wch_spi
- dtbinding_wch_systick
- dtbinding_wch_usart
- dtbinding_wch_wch_dma

Wistron NeWeb Corporation (wnc)

- dtbinding_wnc_m14a2a

WIZnet Co., Ltd. (wiznet)

- dtbinding_wiznet_w5100s
- dtbinding_wiznet_w5500
- dtbinding_wiznet_w6100
- dtbinding_wiznet_w6300

Worldsemi Co., Limited (worldsemi)

- dtbinding_worldsemi_ws2812_bflb_wo
- dtbinding_worldsemi_ws2812_gpio
- dtbinding_worldsemi_ws2812_i2s
- dtbinding_worldsemi_ws2812_pulse_io
- dtbinding_worldsemi_ws2812_rpi_pico_pio
- dtbinding_worldsemi_ws2812_spi
- dtbinding_worldsemi_ws2812_uart

Würth Elektronik GmbH. (we)

- dtbinding_we_wsen_hids_2525020210002
- dtbinding_we_wsen_isds_2536030320001_i2c
- dtbinding_we_wsen_isds_2536030320001_spi
- dtbinding_we_wsen_itds_2533020201601_i2c
- dtbinding_we_wsen_itds_2533020201601_spi
- dtbinding_we_wsen_pads_2511020213301_i2c
- dtbinding_we_wsen_pads_2511020213301_spi
- dtbinding_we_wsen_pdms_25131308XXX05_i2c
- dtbinding_we_wsen_pdms_25131308XXX05_spi
- dtbinding_we_wsen_pdus_25131308XXXXX
- dtbinding_we_wsen_tids_2521020222501

X-Powers (x-powers)

- dtbinding_x_powers_axp192
- dtbinding_x_powers_axp192_gpio
- dtbinding_x_powers_axp192_led
- dtbinding_x_powers_axp192_regulator
- dtbinding_x_powers_axp2101
- dtbinding_x_powers_axp2101_charger
- dtbinding_x_powers_axp2101_fuel_gauge
- dtbinding_x_powers_axp2101_led
- dtbinding_x_powers_axp2101_regulator

Xen Hypervisor (xen)

- dtbinding_xen_hvc_consoleio
- dtbinding_xen_hvc_uart
- dtbinding_xen_watchdog
- dtbinding_xen_xen

Xilinx (xlnx)

- dtbinding_xlnx_axi_dma_1.00.a
- dtbinding_xlnx_axi_ethernet_1.00.a
- dtbinding_xlnx_axi_ethernet_1.00.a_mdio
- dtbinding_xlnx_clkx5_wiz_1.0
- dtbinding_xlnx_eth_dma
- dtbinding_xlnx_fpga
- dtbinding_xlnx_gem
- dtbinding_xlnx_gem_mdio
- dtbinding_xlnx_mbox_versal_ipi_mailbox
- dtbinding_xlnx_pinctrl_zynq
- dtbinding_xlnx_pinctrl_zynqmp
- dtbinding_xlnx_ps_gpio
- dtbinding_xlnx_ps_gpio_bank
- dtbinding_xlnx_ttc
- dtbinding_xlnx_ttc_counter
- dtbinding_xlnx_ttcps
- dtbinding_xlnx_versal_8.9a
- dtbinding_xlnx_versal_wwdt
- dtbinding_xlnx_xps_gpio_1.00.a
- dtbinding_xlnx_xps_gpio_1.00.a_gpio2
- dtbinding_xlnx_xps_iic_2.00.a
- dtbinding_xlnx_xps_iic_2.1
- dtbinding_xlnx_xps_spi_2.00.a
- dtbinding_xlnx_xps_timebase_wdt_1.00.a
- dtbinding_xlnx_xps_timer_1.00.a
- dtbinding_xlnx_xps_timer_1.00.a_pwm
- dtbinding_xlnx_xps_uartlite_1.00.a
- dtbinding_xlnx_xuartps
- dtbinding_xlnx_zynq_ocm
- dtbinding_xlnx_zynqmp_ddrc_2.40a
- dtbinding_xlnx_zynqmp_dma_1.0
- dtbinding_xlnx_zynqmp_ipi_mailbox

- dtbinding_xlnx_zynqmp_qspi_1.0
- dtbinding_xlnx_zynqmp_rtc

Zephyr Project (zephyr)

- dtbinding_zephyr_adc_emul
- dtbinding_zephyr_bbram_emul
- dtbinding_zephyr_biometrics_emul
- dtbinding_zephyr_bt_hci_3wire_uart
- dtbinding_zephyr_bt_hci_entropy
- dtbinding_zephyr_bt_hci_ipc
- dtbinding_zephyr_bt_hci_ll_sw_split
- dtbinding_zephyr_bt_hci_spi
- dtbinding_zephyr_bt_hci_spi_peripheral
- dtbinding_zephyr_bt_hci_spi_slave
- dtbinding_zephyr_bt_hci_uart
- dtbinding_zephyr_bt_hci_userchan
- dtbinding_zephyr_can_loopback
- dtbinding_zephyr_cdc_acm_uart
- dtbinding_zephyr_cdc_ecm_ethernet
- dtbinding_zephyr_cdc_ncm_ethernet
- dtbinding_zephyr_charger_gpio
- dtbinding_zephyr_coredump
- dtbinding_zephyr_counter_watchdog
- dtbinding_zephyr_cpu_freq_thermal_cap
- dtbinding_zephyr_dac_emul
- dtbinding_zephyr_dali_pwm
- dtbinding_zephyr_devmux
- dtbinding_zephyr_displays
- dtbinding_zephyr_dma_emul
- dtbinding_zephyr_dsa_port
- dtbinding_zephyr_dummy_codec
- dtbinding_zephyr_dummy_dc
- dtbinding_zephyr_emu_eeprom
- dtbinding_zephyr_emul_device_mspi
- dtbinding_zephyr_emul_flash_mspi
- dtbinding_zephyr_espi_emul_controller
- dtbinding_zephyr_fake_can
- dtbinding_zephyr_fake_comp
- dtbinding_zephyr_fake_eeprom

- dtbinding_zephyr_fake_leds
- dtbinding_zephyr_fake_pwm
- dtbinding_zephyr_fake_regulator
- dtbinding_zephyr_fake_rtc
- dtbinding_zephyr_fake_stepper_ctrl
- dtbinding_zephyr_fake_stepper_driver
- dtbinding_zephyr_flash_disk
- dtbinding_zephyr_flow_meter
- dtbinding_zephyr_fstab
- dtbinding_zephyr_fstab_ext2
- dtbinding_zephyr_fstab_fatfs
- dtbinding_zephyr_fstab_littlefs
- dtbinding_zephyr_ftl_dhara
- dtbinding_zephyr_fuel_gauge_composite
- dtbinding_zephyr_generic_emul_sensor
- dtbinding_zephyr_generic_pstate
- dtbinding_zephyr_gnss_emul
- dtbinding_zephyr_gpio_emul
- dtbinding_zephyr_gpio_emul_sdl
- dtbinding_zephyr_gpio_step_dir_stepper_ctrl
- dtbinding_zephyr_h_bridge_stepper_ctrl
- dtbinding_zephyr_hid_device
- dtbinding_zephyr_hub12
- dtbinding_zephyr_hwinfo_nvmem
- dtbinding_zephyr_i2c_dump_allowlist
- dtbinding_zephyr_i2c_emul_controller
- dtbinding_zephyr_i2c_target_eeprom
- dtbinding_zephyr_i2c_target_tmp103
- dtbinding_zephyr_ieee802154_uart_pipe
- dtbinding_zephyr_input_double_tap
- dtbinding_zephyr_input_longpress
- dtbinding_zephyr_input_sdl_touch
- dtbinding_zephyr_ipc_icbmsg
- dtbinding_zephyr_ipc_icmsg
- dtbinding_zephyr_ipc_icmsg_me_follower
- dtbinding_zephyr_ipc_icmsg_me_initiator
- dtbinding_zephyr_ipc_openamp_static_vrings
- dtbinding_zephyr_log_uart
- dtbinding_zephyr_lvgl_button_input

- dtbinding_zephyr_lvgl_encoder_input
- dtbinding_zephyr_lvgl_keypad_input
- dtbinding_zephyr_lvgl_pointer_input
- dtbinding_zephyr_mapped_partition
- dtbinding_zephyr_mbox_ipm
- dtbinding_zephyr_mctp_i2c_gpio_controller
- dtbinding_zephyr_mctp_i2c_gpio_target
- dtbinding_zephyr_mctp_i3c_controller
- dtbinding_zephyr_mctp_i3c_endpoint
- dtbinding_zephyr_mctp_i3c_target
- dtbinding_zephyr_mdio_gpio
- dtbinding_zephyr_memc_ram_disk
- dtbinding_zephyr_memory_region
- dtbinding_zephyr_midi2_device
- dtbinding_zephyr_mipi_dbi_bitbang
- dtbinding_zephyr_mipi_dbi_spi
- dtbinding_zephyr_mmc_disk
- dtbinding_zephyr_modbus_serial
- dtbinding_zephyr_mspi_emul_controller
- dtbinding_zephyr_native_linux_can
- dtbinding_zephyr_native_linux_evdev
- dtbinding_zephyr_native_linux_leds
- dtbinding_zephyr_native_linux_temp
- dtbinding_zephyr_native_posix_udc
- dtbinding_zephyr_native_ptp_clock
- dtbinding_zephyr_native_pty_uart
- dtbinding_zephyr_native_sim_counter
- dtbinding_zephyr_native_sim_cpu
- dtbinding_zephyr_native_sim_dmic
- dtbinding_zephyr_native_sim_i2s
- dtbinding_zephyr_native_sim_pstate
- dtbinding_zephyr_native_sim_rng
- dtbinding_zephyr_native_sim_video_fifo
- dtbinding_zephyr_native_sim_wifi
- dtbinding_zephyr_native_tap
- dtbinding_zephyr_native_tty_uart
- dtbinding_zephyr_nus_uart
- dtbinding_zephyr_otp_emul
- dtbinding_zephyr_panel_color_palette

- dtbinding_zephyr_panel_timing
- dtbinding_zephyr_pdm_dmic
- dtbinding_zephyr_peripheral_device_mspi
- dtbinding_zephyr_power_state
- dtbinding_zephyr_psa_crypto_rng
- dtbinding_zephyr_pstate
- dtbinding_zephyr_pulse_io_loopback
- dtbinding_zephyr_pwm_bitbang
- dtbinding_zephyr_ram_disk
- dtbinding_zephyr_retained_ram
- dtbinding_zephyr_retained_reg
- dtbinding_zephyr_retention
- dtbinding_zephyr_rtc_counter
- dtbinding_zephyr_rtc_emul
- dtbinding_zephyr_sdhc_spi_slot
- dtbinding_zephyr_sdl_dc
- dtbinding_zephyr_sdmmc_disk
- dtbinding_zephyr_semihost_flash
- dtbinding_zephyr_sensing
- dtbinding_zephyr_sensing_hinge_angle
- dtbinding_zephyr_sensing_phy_3d_sensor
- dtbinding_zephyr_sim_eeprom
- dtbinding_zephyr_sim_flash
- dtbinding_zephyr_smp_spi
- dtbinding_zephyr_spi_bitbang
- dtbinding_zephyr_spi_emul_controller
- dtbinding_zephyr_swdp_gpio
- dtbinding_zephyr_tach_gpio
- dtbinding_zephyr_uac2
- dtbinding_zephyr_uac2_audio_streaming
- dtbinding_zephyr_uac2_clock_source
- dtbinding_zephyr_uac2_feature_unit
- dtbinding_zephyr_uac2_input_terminal
- dtbinding_zephyr_uac2_output_terminal
- dtbinding_zephyr_uart_bitbang
- dtbinding_zephyr_uart_bridge
- dtbinding_zephyr_uart_emul
- dtbinding_zephyr_udc_skeleton
- dtbinding_zephyr_udc_virtual

- dtbinding_zephyr_uhc_virtual
- dtbinding_zephyr_usb_c_pwrctrl
- dtbinding_zephyr_usb_c_vbus_adc
- dtbinding_zephyr_usb_c_vbus_tcpci
- dtbinding_zephyr_uvc_device
- dtbinding_zephyr_video_emul_imager
- dtbinding_zephyr_video_emul_rx
- dtbinding_zephyr_video_sw_generator
- dtbinding_zephyr_w1_gpio
- dtbinding_zephyr_w1_serial
- dtbinding_zephyr_wifi_hwsim

Zhengzhou Winsen Electronics Technology Co., Ltd. (winsen)

- dtbinding_winsen_mhz19b

Unknown vendor

- dtbinding_ericsson_f5521gw
- dtbinding_spi_hdlc_rcp_if
- dtbinding_swerv_pic
- dtbinding_uart_hdlc_rcp_if

5.4 Configuration System (Kconfig)

The Zephyr kernel and subsystems can be configured at build time to adapt them for specific application and platform needs. Configuration is handled through Kconfig, which is the same configuration system used by the Linux kernel. The goal is to support configuration without having to change any source code.

Configuration options (often called *symbols*) are defined in `Kconfig` files, which also specify dependencies between symbols that determine what configurations are valid. Symbols can be grouped into menus and sub-menus to keep the interactive configuration interfaces organized.

The output from Kconfig is a header file `autoconf.h` with macros that can be tested at build time. Code for unused features can be compiled out to save space.

The following sections explain how to set Kconfig configuration options, go into detail on how Kconfig is used within the Zephyr project, and have some tips and best practices for writing `Kconfig` files.

5.4.1 Interactive Kconfig interfaces

There are two interactive configuration interfaces available for exploring the available Kconfig options and making temporary changes: `menuconfig` and `guiconfig`. `menuconfig` is a curses-based interface that runs in the terminal, while `guiconfig` is a graphical configuration interface.

Note

The configuration can also be changed by editing `zephyr/.config` in the application build directory by hand. Using one of the configuration interfaces is often handier, as they correctly handle dependencies between configuration symbols.

If you try to enable a symbol with unsatisfied dependencies in `zephyr/.config`, the assignment will be ignored and overwritten when re-configuring.

To make a setting permanent, you should set it in a `*.conf` file, as described in *Setting Kconfig configuration values*.

Tip

Saving a minimal configuration file (with e.g. `D` in menuconfig) and inspecting it can be handy when making settings permanent. The minimal configuration file only lists symbols that differ from their default value.

To run one of the configuration interfaces, do this:

1. Build your application as usual using either `west` or `cmake`:

   Using west:

   ```
   west build -b <board>
   ```

   Using CMake and ninja:

   ```
   mkdir build && cd build
   cmake -GNinja -DBOARD=<board> ..
   ninja
   ```

2. To run the terminal-based `menuconfig` interface, use either of these commands:

   ```
   west build -t menuconfig
   ```

   ```
   ninja menuconfig
   ```

   To run the graphical `guiconfig`, use either of these commands:

   ```
   west build -t guiconfig
   ```

   ```
   ninja guiconfig
   ```

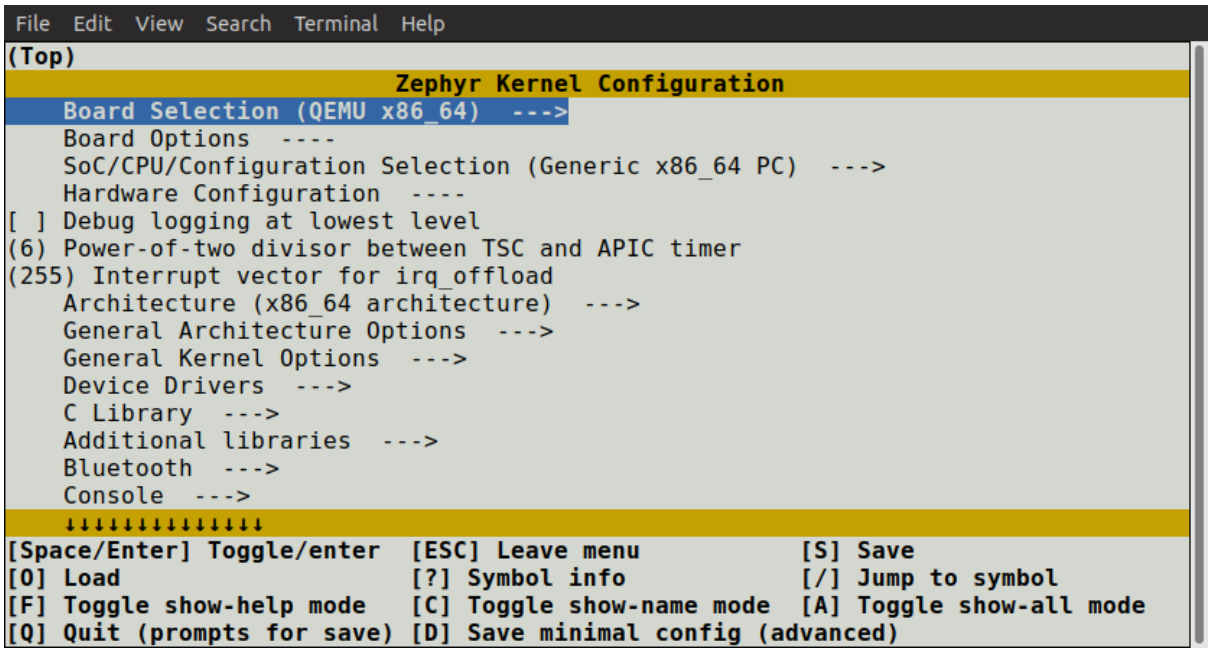
   Note

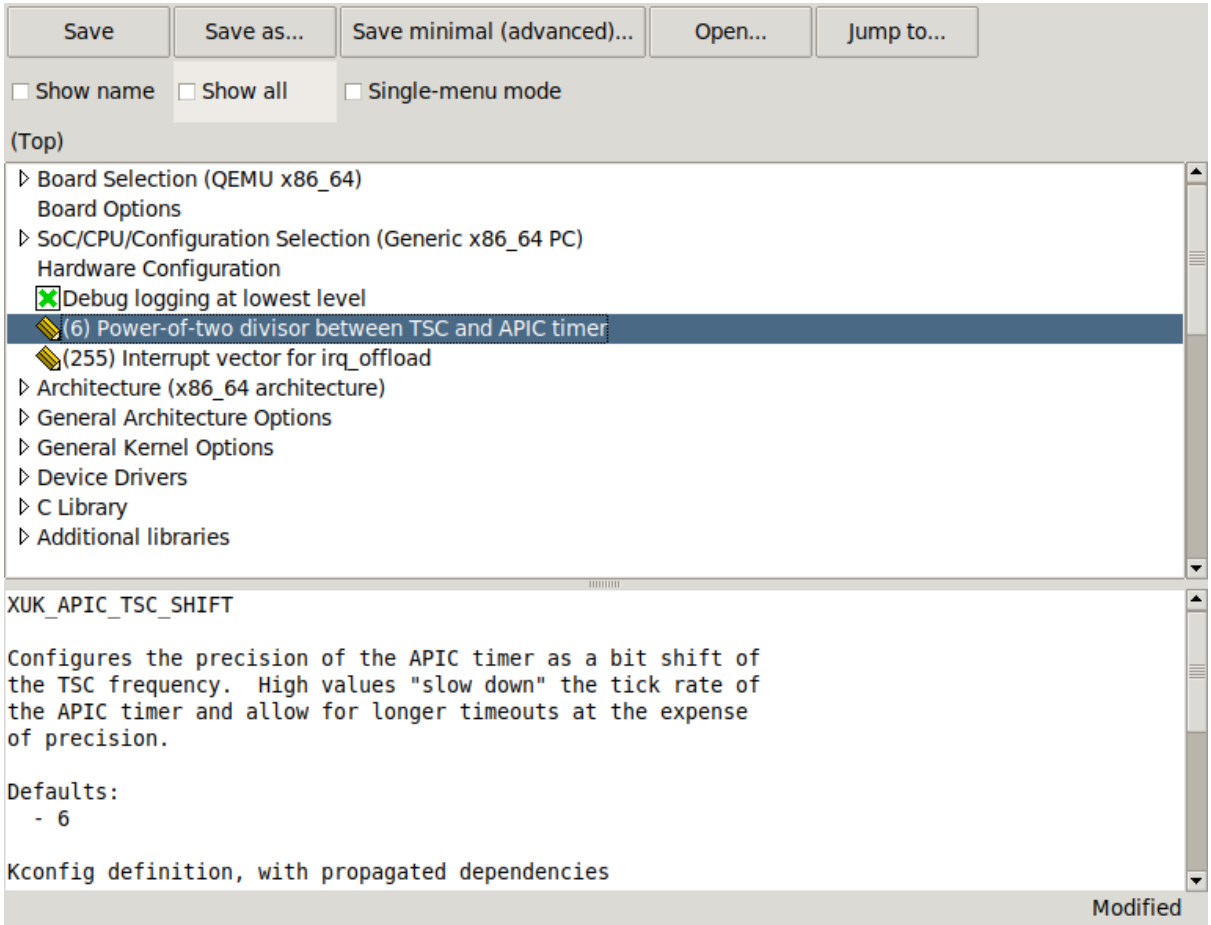
   If you get an import error for `tkinter` when trying to run `guiconfig`, you are missing required packages. See *Install Linux Host Dependencies*. The package you need is usually called something like `python3-tk/python3-tkinter`.

   `tkinter` is not included by default in many Python installations, despite being part of the standard library.

   The two interfaces are shown below:

   `guiconfig` always shows the help text and other information related to the currently selected item in the bottom window pane. In the terminal interface, press ? to view the same


File Edit View Search Terminal Help
(Top)
Zephyr Kernel Configuration
Board Selection (QEMU x86_64) --->
Board Options ----
SoC/CPU/Configuration Selection (Generic x86_64 PC) --->
Hardware Configuration ----
[ ] Debug logging at lowest level
(6) Power-of-two divisor between TSC and APIC timer
(255) Interrupt vector for irq_offload
Architecture (x86_64 architecture) --->
General Architecture Options --->
General Kernel Options --->
Device Drivers --->
C Library --->
Additional libraries --->
Bluetooth --->
Console --->
[Space/Enter] Toggle/enter [ESC] Leave menu [S] Save
[O] Load [?] Symbol info [/] Jump to symbol
[F] Toggle show-help mode [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)


Save
Save as...
Save minimal (advanced)...
Open...
Jump to...
Show name
Show all
Single-menu mode
(Top)
Board Selection (QEMU x86_64)
Board Options
SoC/CPU/Configuration Selection (Generic x86_64 PC)
Hardware Configuration
Debug logging at lowest level
(6) Power-of-two divisor between TSC and APIC timer
(255) Interrupt vector for irq_offload
Architecture (x86_64 architecture)
General Architecture Options
General Kernel Options
Device Drivers
C Library
Additional libraries
XUK_APIC_TSC_SHIFT
Configures the precision of the APIC timer as a bit shift of
the TSC frequency. High values "slow down" the tick rate of
the APIC timer and allow for longer timeouts at the expense
of precision.
Defaults:
- 6
Kconfig definition, with propagated dependencies
Modified

information.

> **Note**
>
> If you prefer to work in the `guiconfig` interface, then it's a good idea to check any changes to Kconfig files you make in *single-menu mode*, which is toggled via a checkbox at the top. Unlike full-tree mode, single-menu mode will distinguish between symbols defined with `config` and symbols defined with `menuconfig`, showing you what things would look like in the `menuconfig` interface.

3. Change configuration values in the `menuconfig` interface as follows:
   - Navigate the menu with the arrow keys. Common Vim key bindings are supported as well.
   - Use `Space` and `Enter` to enter menus and toggle values. Menus appear with `--->` next to them. Press `ESC` to return to the parent menu.

     Boolean configuration options are shown with *[ ]* brackets, while numeric and string-valued configuration symbols are shown with *( )* brackets. Symbol values that can't be changed are shown as - - or -*-.

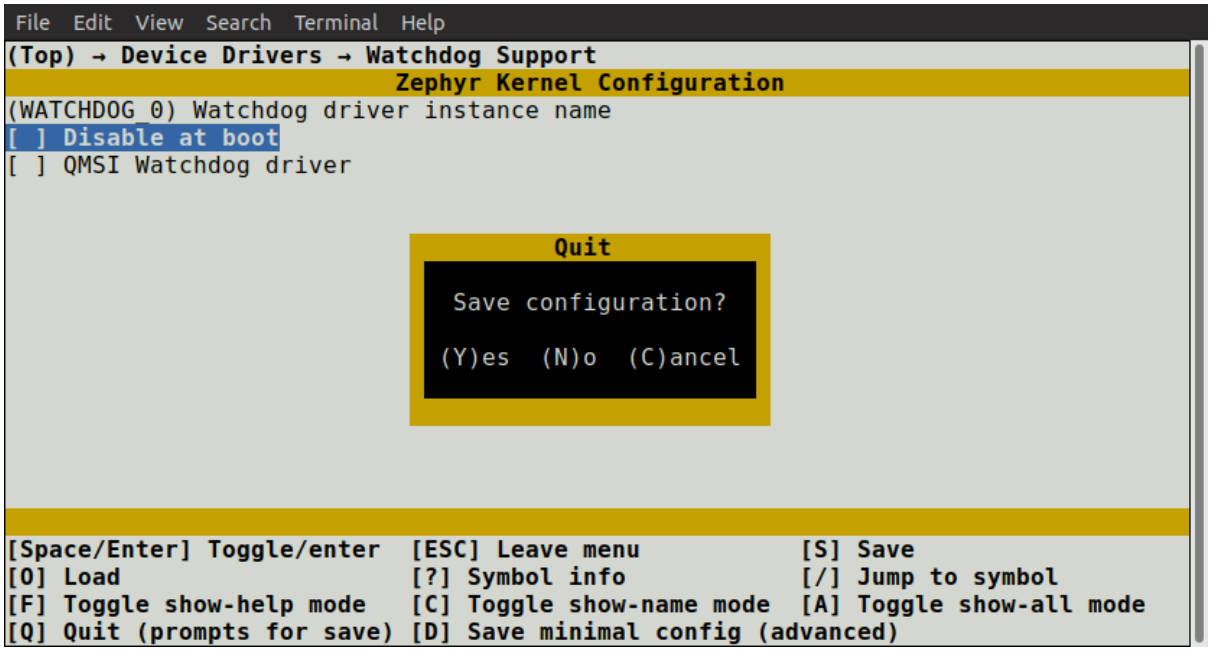
     > **Note**
     >
     > You can also press `Y` or `N` to set a boolean configuration symbol to the corresponding value.

   - Press `?` to display information about the currently selected symbol, including its help text. Press `ESC` or `Q` to return from the information display to the menu.

   In the `guiconfig` interface, either click on the image next to the symbol to change its value, or double-click on the row with the symbol (this only works if the symbol has no children, as double-clicking a symbol with children open/closes its menu instead).

   `guiconfig` also supports keyboard controls, which are similar to `menuconfig`.

4. Pressing `Q` in the `menuconfig` interface will bring up the save-and-quit dialog (if there are changes to save):

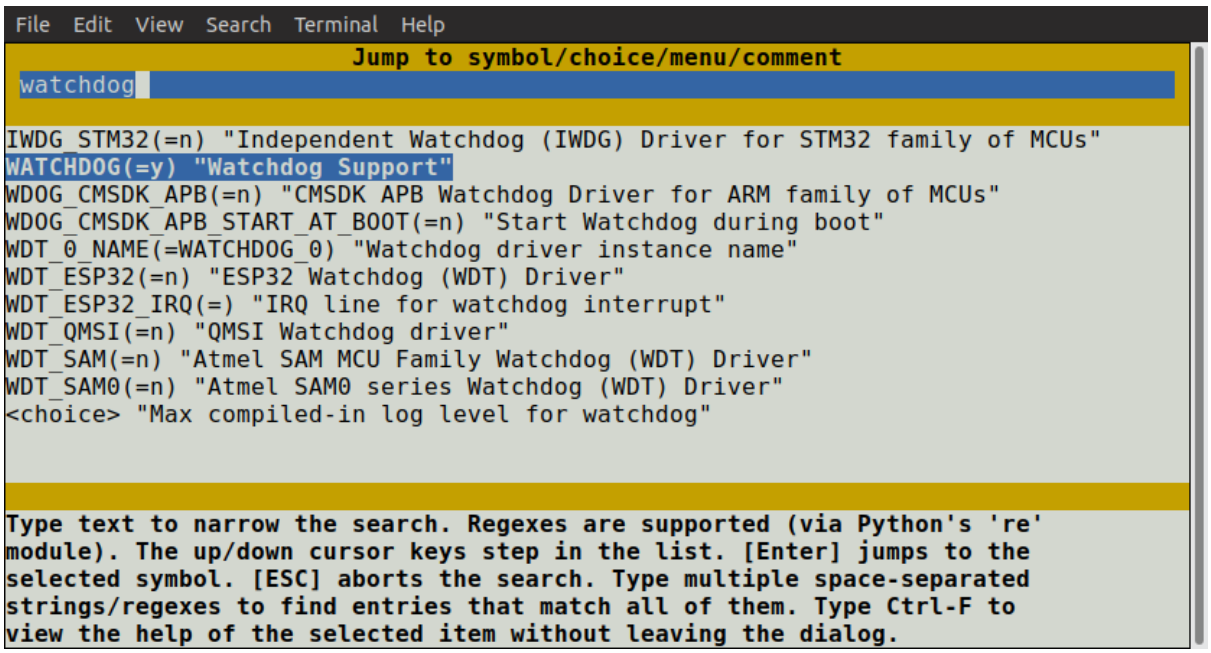


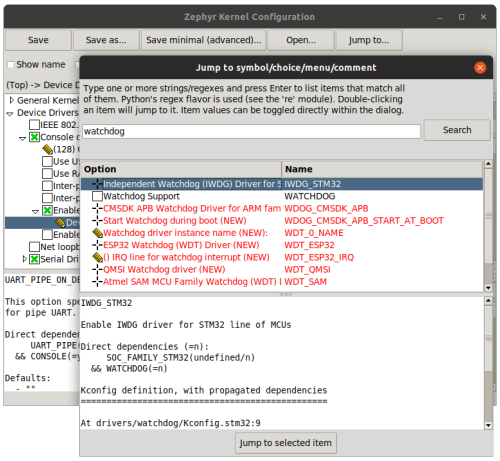
Press `Y` to save the kernel configuration options to the default filename (`zephyr/.config`). You will typically save to the default filename unless you are experimenting with different configurations.

The `guiconfig` interface will also prompt for saving the configuration on exit if it has been modified.

> **Note**
>
> The configuration file used during the build is always `zephyr/.config`. If you have another saved configuration that you want to build with, copy it to `zephyr/.config`. Make sure to back up your original configuration file.
>
> Also note that filenames starting with `.` are not listed by `ls` by default on Linux and macOS. Use the `-a` flag to see them.

Finding a symbol in the menu tree and navigating to it can be tedious. To jump directly to a symbol, press the / key (this also works in `guiconfig`). This brings up the following dialog, where you can search for symbols by name and jump to them. In `guiconfig`, you can also change symbol values directly within the dialog.





If you jump to a symbol that isn't currently visible (e.g., due to having unsatisfied dependencies),

then *show-all mode* will be enabled. In show-all mode, all symbols are displayed, including currently invisible symbols. To turn off show-all mode, press `A` in `menuconfig` or `Ctrl-A` in `guiconfig`.

> **Note**
>
> Show-all mode can't be turned off if there are no visible items in the current menu.

To figure out why a symbol you jumped to isn't visible, inspect its dependencies, either by pressing ? in `menuconfig` or in the information pane at the bottom in `guiconfig`. If you discover that the symbol depends on another symbol that isn't enabled, you can jump to that symbol in turn to see if it can be enabled.

> **Note**
>
> In `menuconfig`, you can press `Ctrl-F` to view the help of the currently selected item in the jump-to dialog without leaving the dialog.

For more information on `menuconfig` and `guiconfig`, see the Python docstrings at the top of menuconfig.py and guiconfig.py.

5.4.2 Tracing values to their source

The merged configuration saved to `zephyr/.config` contains the result of applying the user configuration file to the whole tree of Kconfig files. This process can be hard to follow, especially when invisible symbols are being implicitly selected from anywhere in the tree. To help with this, the `traceconfig` target can be used to generate a file in the build directory that details how each symbol got its final value.

After building the Zephyr project as usual, the Kconfig trace can be generated with either of these commands:

```
west build -t traceconfig
```

```
ninja traceconfig
```

> **Note**
>
> The generated information is only useful on a clean build, because otherwise the `.config` file "pins" all settings to a specific value (as described in *Stuck symbols*). Therefore, it is recommended to run a *pristine build* before generating the trace.

The output will be in the `zephyr/kconfig-trace.md` file in the build directory. This file is best viewed within IDEs to take advantage of Markdown elements (tables, highlighting, clickable links), but can easily be understood even when opened directly with any text editor.

The report is divided in three sections:

1. Visible symbols
2. Invisible symbols
3. Unset symbols

For sections 1 and 2, a table is presented where each symbol is shown with its type, name, and current value. The fourth column details the kind of statement that resulted in the value being applied, and can be one of the following:

- *assigned*, when an explicit assignment, of the form `CONFIG_xxx=y`, is read from a config file;

- *default*, when there was no user assignment, but an applicable default value was read from Kconfig tree;
- *selected* or *implied*, when a `select` or `imply` statement from a separate symbol caused the symbol to be set.

The fifth column details the location for the source statement. For the first two kinds of statements, the exact location (file name and line number) is provided; in the latter cases, it contains the expressions that resulted in the symbol being set.

Finally, section 3 simply lists all undefined symbols. These have no additional information since they never received a value by any of the above means.

5.4.3 Setting Kconfig configuration values

The *menuconfig and guiconfig interfaces* can be used to test out configurations during application development. This page explains how to make settings permanent.

All Kconfig options can be searched in the Kconfig search page.

> **Note**
>
> Before making changes to Kconfig files, it's a good idea to also go through the *Kconfig - Tips and Best Practices* page.

Visible and invisible Kconfig symbols

When making Kconfig changes, it's important to understand the difference between *visible* and *invisible* symbols.

- A visible symbol is a symbol defined with a prompt. Visible symbols show up in the interactive configuration interfaces (hence *visible*), and can be set in configuration files.

  Here's an example of a visible symbol:

  ```
  config FPU
     bool "Support floating point operations"
     depends on HAS_FPU
  ```

  The symbol is shown like this in `menuconfig`, where it can be toggled:

  ```
  [ ] Support floating point operations
  ```

- An *invisible* symbol is a symbol without a prompt. Invisible symbols are not shown in the interactive configuration interfaces, and users have no direct control over their value. They instead get their value from defaults or from other symbols.

  Here's an example of an invisible symbol:

  ```
  config CPU_HAS_FPU
     bool
     help
       This symbol is y if the CPU has a hardware floating point unit.
  ```

  In this case, `CPU_HAS_FPU` is enabled through other symbols having `select CPU_HAS_FPU`.

Setting symbols in configuration files

Visible symbols can be configured by setting them in configuration files. The initial configuration is produced by merging a `*_defconfig` file for the board with application settings, usually from `prj.conf`. See *The Initial Configuration* below for more details.

Assignments in configuration files use this syntax:

```
CONFIG_<symbol name>=<value>
```

There should be no spaces around the equals sign.

bool symbols can be enabled or disabled by setting them to y or n, respectively. The FPU symbol from the example above could be enabled like this:

```
CONFIG_FPU=y
```

Note

A boolean symbol can also be set to n with a comment formatted like this:

```
# CONFIG_SOME_OTHER_BOOL is not set
```

This is the format you will see in the merged configuration saved to zephyr/.config in the build directory.

This style is accepted for historical reasons: Kconfig configuration files can be parsed as makefiles (though Zephyr doesn't use this). Having n-valued symbols correspond to unset variables simplifies tests in Make.

Other symbol types are assigned like this:

```
CONFIG_SOME_STRING="cool value"
CONFIG_SOME_INT=123
```

Comments use a #:

```
# This is a comment
```

Assignments in configuration files are only respected if the dependencies for the symbol are satisfied. A warning is printed otherwise. To figure out what the dependencies of a symbol are, use one of the *interactive configuration interfaces* (you can jump directly to a symbol with /), or look up the symbol in the Kconfig search page.

The Initial Configuration

The initial configuration for an application comes from merging configuration settings from three sources:

1. A BOARD-specific configuration file stored in boards/<VENDOR>/<BOARD>/<BOARD>_defconfig
2. Any CMake cache entries prefix with CONFIG_
3. The application configuration

The application configuration can come from the sources below (each file is known as a Kconfig fragment, which are then merged to get the final configuration used for a particular build). By default, prj.conf is used.

1. If CONF_FILE is set, the configuration file(s) specified in it are merged and used as the application configuration. CONF_FILE can be set in various ways:
    1. In CMakeLists.txt, before calling find_package(Zephyr)
    2. By passing -DCONF_FILE=<conf file(s)>, either directly or via west
    3. From the CMake variable cache
2. Otherwise, if boards/<BOARD>.conf exists in the application configuration directory, the result of merging it with prj.conf is used.

3. Otherwise, if board revisions are used and boards/<BOARD>_<revision>.conf exists in the application configuration directory, the result of merging it with prj.conf and boards/<BOARD>.conf is used.
4. Otherwise, prj.conf is used from the application configuration directory. If it does not exist then a fatal error will be emitted.

Furthermore, applications can have SoC Kconfig fragments added to the configuration, the file socs/<SOC>_<BOARD_QUALIFIERS>.conf will be applied if it exists, after the main project configuration has been applied and before any board Kconfig fragments files have been applied.

All configuration files will be taken from the application's configuration directory except for files with an absolute path that are given with the CONF_FILE, EXTRA_CONF_FILE, DTC_OVERLAY_FILE, and EXTRA_DTC_OVERLAY_FILE arguments. For these, a file in a Zephyr module can be referred by escaping the Zephyr module dir variable like this \${ZEPHYR_<module>_MODULE_DIR}/<path-to>/<file> when setting any of said variables in the application's CMakeLists.txt.

See *Application Configuration Directory* on how the application configuration directory is defined.

If a symbol is assigned both in <BOARD>_defconfig and in the application configuration, the value set in the application configuration takes precedence.

The merged configuration is saved to zephyr/.config in the build directory.

As long as zephyr/.config exists and is up-to-date (is newer than any BOARD and application configuration files), it will be used in preference to producing a new merged configuration. zephyr/.config is also the configuration that gets modified when making changes in the *interactive configuration interfaces*.

Treating Kconfig warnings as errors

Some Kconfig warnings abort the build by default, for example assignments to undefined symbols. Others are only printed, for example when a symbol is set more than once, or when an assignment is ignored because the dependencies of the symbol are not satisfied.

Set the **KCONFIG_WARNING_AS_ERROR** CMake variable to treat *every* Kconfig warning as an error:

```
west build -b <board> <app> -- -DKCONFIG_WARNING_AS_ERROR=y
```

This is useful in CI, where a silently ignored configuration setting is likely to be a mistake. It is read with zephyr_get(), so it can also be given as an *environment variable* or through a *Zephyr Build Configuration CMake packages*.

Tracking Kconfig symbols

It is possible to create Kconfig symbols which takes the default value of another Kconfig symbol.

This is valuable when you want a symbol specific to an application or subsystem but do not want to rely directly on the common symbol. For example, you might want to decouple the settings so they can be independently configured, or to ensure you always have a locally named setting, even if the external setting name changes. is later changed.

For example, consider the common FOO_STRING setting where a subsystem wants to have a SUB_FOO_STRING but still allow for customization.

This can be done like this:

```
config FOO_STRING
        string "Foo"
        default "foo"

config SUB_FOO_STRING
```

(continues on next page)

(continued from previous page)

```
        string "Sub-foo"
        default FOO_STRING
```

This ensures that the default value of `SUB_FOO_STRING` is identical to `FOO_STRING` while still allows users to configure both settings independently.

It is also possible to make `SUB_FOO_STRING` invisible and thereby keep the two symbols in sync, unless the value of the tracking symbol is changed in a `defconfig` file.

```
config FOO_STRING
        string "Foo"
        default "foo"

config SUB_FOO_STRING
        string
        default FOO_STRING
        help
          Hidden symbol which follows FOO_STRING
          Can be changed through *.defconfig files.
```

Configuring invisible Kconfig symbols

When making changes to the default configuration for a board, you might have to configure invisible symbols. This is done in `boards/<VENDOR>/<BOARD>/Kconfig.defconfig`, which is a regular `Kconfig` file.

> **Note**
>
> Assignments in `.config` files have no effect on invisible symbols, so this scheme is not just an organizational issue.

Assigning values in `Kconfig.defconfig` relies on defining a Kconfig symbol in multiple locations. As an example, say we want to set `FOO_WIDTH` below to 32:

```
config FOO_WIDTH
    int
```

To do this, we extend the definition of `FOO_WIDTH` as follows, in `Kconfig.defconfig`:

```
if BOARD_MY_BOARD

config FOO_WIDTH
    default 32

endif
```

> **Note**
>
> Since the type of the symbol (`int`) has already been given at the first definition location, it does not need to be repeated here. Only giving the type once at the "base" definition of the symbol is a good idea for reasons explained in *Common Kconfig shorthands*.

`default` values in `Kconfig.defconfig` files have priority over `default` values given on the "base" definition of a symbol. Internally, this is implemented by including the `Kconfig.defconfig` files

first. Kconfig uses the first `default` with a satisfied condition, where an empty condition corresponds to `if y` (is always satisfied).

Note that conditions from surrounding top-level `if`s are propagated to symbol properties, so the above `default` is equivalent to `default 32 if BOARD_MY_BOARD`.

Multiple symbol definitions When a symbol is defined in multiple locations, each definition acts as an independent symbol that happens to share the same name. This means that properties are not appended to previous definitions. If the conditions for **ANY** definition result in the symbol resolving to `y`, the symbol will be `y`. It is therefore not possible to make the dependencies of a symbol more restrictive by defining it in multiple locations.

For example, the dependencies of the symbol `FOO` below are satisfied if either `DEP1` **OR** `DEP2` are true, it does not require both:

```
config FOO
   ...
   depends on DEP1

config FOO
   ...
   depends on DEP2
```

Warning

Symbols without explicit dependencies still follow the above rule. A symbol without any dependencies will result in the symbol always being assignable. The definition below will result in `FOO` always being enabled by default, regardless of the value of `DEP1`.

```
config FOO
   bool "FOO"
   depends on DEP1

config FOO
   default y
```

This dependency weakening can be avoided with the *configdefault* extension if the desire is only to add a new default without modifying any other behaviour of the symbol.

Note

When making changes to `Kconfig.defconfig` files, always check the symbol's direct dependencies in one of the *interactive configuration interfaces* afterwards. It is often necessary to repeat dependencies from the base definition of the symbol to avoid weakening a symbol's dependencies.

Motivation for Kconfig.defconfig files One motivation for this configuration scheme is to avoid making fixed `BOARD`-specific settings configurable in the interactive configuration interfaces. If all board configuration were done via `<BOARD>_defconfig`, all symbols would have to be visible, as values given in `<BOARD>_defconfig` have no effect on invisible symbols.

Having fixed settings be user-configurable would clutter up the configuration interfaces and make them harder to understand, and would make it easier to accidentally create broken configurations.

When dealing with fixed board-specific settings, also consider whether they should be handled via *devicetree* instead.

Configuring choices There are two ways to configure a Kconfig `choice`:

1. By setting one of the choice symbols to `y` in a configuration file.

   Setting one choice symbol to `y` automatically gives all other choice symbols the value `n`.

   If multiple choice symbols are set to `y`, only the last one set to `y` will be honored (the rest will get the value `n`). This allows a choice selection from a board `defconfig` file to be overridden from an application `prj.conf` file.

2. By changing the `default` of the choice in `Kconfig.defconfig`.

   As with symbols, changing the default for a choice is done by defining the choice in multiple locations. For this to work, the choice must have a name.

   As an example, assume that a choice has the following base definition (here, the name of the choice is `FOO`):

```
choice FOO
    bool "Foo choice"
    default B

config A
    bool "A"

config B
    bool "B"

endchoice
```

   To change the default symbol of `FOO` to `A`, you would add the following definition to `Kconfig.defconfig`:

```
choice FOO
    default A
endchoice
```

The `Kconfig.defconfig` method should be used when the dependencies of the choice might not be satisfied. In that case, you're setting the default selection whenever the user makes the choice visible.

More Kconfig resources The *Kconfig - Tips and Best Practices* page has some tips for writing Kconfig files.

The kconfiglib.py docstring (at the top of the file) goes over how symbol values are calculated in detail.

5.4.4 Kconfig - Tips and Best Practices

This page covers some Kconfig best practices and explains some Kconfig behaviors and features that might be cryptic or that are easily overlooked.

> **Note**
>
> The official Kconfig documentation is kconfig-language.rst and kconfig-macro-language.rst.

- *What to turn into Kconfig options*

- *What not to turn into Kconfig options*
  - *Options that specify a device in the system by name*
  - *Options that specify fixed hardware configuration*
- *`select` statements*
  - *`select` pitfalls*
  - *Alternatives to `select`*
  - *Using `select` for helper symbols*
  - *`select` recommendations*
- *(Lack of) conditional includes*
- *"Stuck" symbols in menuconfig and guiconfig*
- *Assignments to promptless symbols in configuration files*
- *`depends on` and `string`/`int`/`hex` symbols*
- *`menuconfig` symbols*
- *Commas in macro arguments*
- *Checking changes in menuconfig/guiconfig*
- *Checking changes with `scripts/kconfig/lint.py`*
- *Style recommendations and shorthands*
  - *Factoring out common dependencies*
  - *Redundant defaults*
  - *Common Kconfig shorthands*
  - *Prompt strings*
  - *Style*
- *Lesser-known/used Kconfig features*
  - *The `imply` statement*
  - *Optional prompts*
  - *Optional choices*
  - *`visible if` conditions*
- *Other resources*

What to turn into Kconfig options

When deciding whether something belongs in Kconfig, it helps to distinguish between symbols that have prompts and symbols that don't.

If a symbol has a prompt (e.g. `bool "Enable foo"`), then the user can change the symbol's value in the `menuconfig` or `guiconfig` interface (see *Interactive Kconfig interfaces*), or by manually editing configuration files. Conversely, a symbol without a prompt can never be changed directly by the user, not even by manually editing configuration files.

Only put a prompt on a symbol if it makes sense for the user to change its value.

Symbols without prompts are called *hidden* or *invisible* symbols, because they don't show up in `menuconfig` and `guiconfig`. Symbols that have prompts can also be invisible, when their dependencies are not satisfied.

Symbols without prompts can't be configured directly by the user (they derive their value from other symbols), so less restrictions apply to them. If some derived setting is easier to calculate in Kconfig than e.g. during the build, then do it in Kconfig, but keep the distinction between symbols with and without prompts in mind.

See the *optional prompts* section for a way to deal with settings that are fixed on some machines and configurable on other machines.

What not to turn into Kconfig options

In Zephyr, Kconfig configuration is done after selecting a target board. In general, it does not make sense to use Kconfig for a value that corresponds to a fixed machine-specific setting. Usually, such settings should be handled via *devicetree* instead.

In particular, avoid adding new Kconfig options of the following types:

Options that specify a device in the system by name For example, if you are writing an I2C device driver, avoid creating an option named `MY_DEVICE_I2C_BUS_NAME` for specifying the bus node your device is controlled by. See *Device drivers that depend on other devices* for alternatives.

Similarly, if your application depends on a hardware-specific PWM device to control an RGB LED, avoid creating an option like `MY_PWM_DEVICE_NAME`. See *Applications that depend on board-specific devices* for alternatives.

Options that specify fixed hardware configuration For example, avoid Kconfig options specifying a GPIO pin.

An alternative applicable to device drivers is to define a GPIO specifier with type phandle-array in the device binding, and using the *GPIO* devicetree API from C. Similar advice applies to other cases where devicetree.h provides *Hardware specific APIs* for referring to other nodes in the system. Search the source code for drivers using these APIs for examples.

An application-specific devicetree *binding* to identify board specific properties may be appropriate. See tests/drivers/gpio/gpio_basic_api for an example.

For applications, see blinky for a devicetree-based alternative.

`select` statements

The `select` statement is used to force one symbol to `y` whenever another symbol is `y`. For example, the following code forces `CONSOLE` to `y` whenever `USB_CONSOLE` is `y`:

```
config CONSOLE
     bool "Console support"

...

config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
```

This section covers some pitfalls and good uses for `select`.

`select` pitfalls `select` might seem like a generally useful feature at first, but can cause configuration issues if overused.

For example, say that a new dependency is added to the `CONSOLE` symbol above, by a developer who is unaware of the `USB_CONSOLE` symbol (or simply forgot about it):

```
config CONSOLE
     bool "Console support"
     depends on STRING_ROUTINES
```

Enabling `USB_CONSOLE` now forces `CONSOLE` to `y`, even if `STRING_ROUTINES` is `n`.

To fix the problem, the `STRING_ROUTINES` dependency needs to be added to `USB_CONSOLE` as well:

```
config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
     depends on STRING_ROUTINES

...

config STRING_ROUTINES
     bool "Include string routines"
```

More insidious cases with dependencies inherited from `if` and `menu` statements are common.

An alternative attempt to solve the issue might be to turn the `depends on` into another `select`:

```
config CONSOLE
     bool "Console support"
     select STRING_ROUTINES

...

config USB_CONSOLE
     bool "USB console support"
     select CONSOLE
```

In practice, this often amplifies the problem, because any dependencies added to `STRING_ROUTINES` now need to be copied to both `CONSOLE` and `USB_CONSOLE`.

In general, whenever the dependencies of a symbol are updated, the dependencies of all symbols that (directly or indirectly) select it have to be updated as well. This is very often overlooked in practice, even for the simplest case above.

Chains of symbols selecting each other should be avoided in particular, except for simple helper symbols, as covered below in *Using select for helper symbols*.

Liberal use of `select` also tends to make Kconfig files harder to read, both due to the extra dependencies and due to the non-local nature of `select`, which hides ways in which a symbol might get enabled.

Alternatives to `select` For the example in the previous section, a better solution is usually to turn the `select` into a `depends on`:

```
config CONSOLE
     bool "Console support"

...

config USB_CONSOLE
     bool "USB console support"
     depends on CONSOLE
```

This makes it impossible to generate an invalid configuration, and means that dependencies only ever have to be updated in a single spot.

An objection to using depends on here might be that configuration files that enable USB_CONSOLE now also need to enable CONSOLE:

```
CONFIG_CONSOLE=y
CONFIG_USB_CONSOLE=y
```

This comes down to a trade-off, but if enabling CONSOLE is the norm, then a mitigation is to make CONSOLE default to y:

```
config CONSOLE
     bool "Console support"
     default y
```

This gives just a single assignment in configuration files:

```
CONFIG_USB_CONSOLE=y
```

Note that configuration files that do not want CONSOLE enabled now have to explicitly disable it:

```
CONFIG_CONSOLE=n
```

Using select for helper symbols A good and safe use of select is for setting "helper" symbols that capture some condition. Such helper symbols should preferably have no prompt or dependencies.

For example, a helper symbol for indicating that a particular CPU/SoC has an FPU could be defined as follows:

```
config CPU_HAS_FPU
     bool
     help
       If y, the CPU has an FPU

...

config SOC_FOO
    bool
    select CPU_HAS_FPU

...

config SOC_BAR
    bool
    select CPU_HAS_FPU
```

This makes it possible for other symbols to check for FPU support in a generic way, without having to look for particular architectures:

```
config FPU
     bool "Support floating point operations"
     depends on CPU_HAS_FPU
```

The alternative would be to have dependencies like the following, possibly duplicated in several spots:

```
config FPU
     bool "Support floating point operations"
     depends on SOC_FOO || SOC_BAR || ...
```

Invisible helper symbols can also be useful without `select`. For example, the following code defines a helper symbol that has the value `y` if the machine has some arbitrarily-defined "large" amount of memory:

```
config LARGE_MEM
     def_bool MEM_SIZE >= 64
```

Note

This is short for the following:

```
config LARGE_MEM
  bool
  default MEM_SIZE >= 64
```

`select` recommendations In summary, here are some recommended practices for `select`:

- Avoid selecting symbols with prompts or dependencies. Prefer `depends on`. If `depends on` causes annoying bloat in configuration files, consider adding a Kconfig default for the most common value.

  Rare exceptions might include cases where you're sure that the dependencies of the selecting and selected symbol will never drift out of sync, e.g. when dealing with two simple symbols defined close to one another within the same `if`.

  Common sense applies, but be aware that `select` often causes issues in practice. `depends on` is usually a cleaner and safer solution.

- Select simple helper symbols without prompts and dependencies however much you like. They're a great tool for simplifying Kconfig files.

- An exemption are buses like I2C and SPI, and following the same thought process things like MFD as well. Drivers on these buses should use `select` to allow the automatic activation of the necessary bus drivers when devices on the bus are enabled in the devicetree.

```
config ADC_FOO
   bool "external SPI ADC foo driver"
   select SPI
```

(Lack of) conditional includes

`if` blocks add dependencies to each item within the `if`, as if `depends on` was used.

A common misunderstanding related to `if` is to think that the following code conditionally includes the file `Kconfig.other`:

```
if DEP

source "Kconfig.other"

endif
```

In reality, there are no conditional includes in Kconfig. `if` has no special meaning around a `source`.

Note

> Conditional includes would be impossible to implement, because `if` conditions may contain (either directly or indirectly) forward references to symbols that haven't been defined yet.

Say that `Kconfig.other` above contains this definition:

```
config FOO
     bool "Support foo"
```

In this case, `FOO` will end up with this definition:

```
config FOO
     bool "Support foo"
     depends on DEP
```

Note that it is redundant to add `depends on DEP` to the definition of `FOO` in `Kconfig.other`, because the `DEP` dependency has already been added by `if DEP`.

In general, try to avoid adding redundant dependencies. They can make the structure of the Kconfig files harder to understand, and also make changes more error-prone, since it can be hard to spot that the same dependency is added twice.

"Stuck" symbols in menuconfig and guiconfig

There is a common subtle gotcha related to interdependent configuration symbols with prompts. Consider these symbols:

```
config FOO
     bool "Foo"

config STACK_SIZE
     hex "Stack size"
     default 0x200 if FOO
     default 0x100
```

Assume that the intention here is to use a larger stack whenever `FOO` is enabled, and that the configuration initially has `FOO` disabled. Also, remember that Zephyr creates an initial configuration in `zephyr/.config` in the build directory by merging configuration files (including e.g. `prj.conf`). This configuration file exists before `menuconfig` or `guiconfig` is run.

When first entering the configuration interface, the value of `STACK_SIZE` is 0x100, as expected. After enabling `FOO`, you might reasonably expect the value of `STACK_SIZE` to change to 0x200, but it stays as 0x100.

To understand what's going on, remember that `STACK_SIZE` has a prompt, meaning it is user-configurable, and consider that all Kconfig has to go on from the initial configuration is this:

```
CONFIG_STACK_SIZE=0x100
```

Since Kconfig can't know if the 0x100 value came from a `default` or was typed in by the user, it has to assume that it came from the user. Since `STACK_SIZE` is user-configurable, the value from the configuration file is respected, and any symbol defaults are ignored. This is why the value of `STACK_SIZE` appears to be "frozen" at 0x100 when toggling `FOO`.

The right fix depends on what the intention is. Here's some different scenarios with suggestions:

- If `STACK_SIZE` can always be derived automatically and does not need to be user-configurable, then just remove the prompt:

```
config STACK_SIZE
   hex
   default 0x200 if FOO
   default 0x100
```

Symbols without prompts ignore any value from the saved configuration.

- If STACK_SIZE should usually be user-configurable, but needs to be set to 0x200 when FOO is enabled, then disable its prompt when FOO is enabled, as described in *optional prompts*:

```
config STACK_SIZE
   hex "Stack size" if !FOO
   default 0x200 if FOO
   default 0x100
```

- If STACK_SIZE should usually be derived automatically, but needs to be set to a custom value in rare circumstances, then add another option for making STACK_SIZE user-configurable:

```
config CUSTOM_STACK_SIZE
   bool "Use a custom stack size"
   help
     Enable this if you need to use a custom stack size. When disabled, a
     suitable stack size is calculated automatically.

config STACK_SIZE
   hex "Stack size" if CUSTOM_STACK_SIZE
   default 0x200 if FOO
   default 0x100
```

As long as CUSTOM_STACK_SIZE is disabled, STACK_SIZE will ignore the value from the saved configuration.

It is a good idea to try out changes in the menuconfig or guiconfig interface, to make sure that things behave the way you expect. This is especially true when making moderately complex changes like these.

Assignments to promptless symbols in configuration files

Assignments to hidden (promptless, also called *invisible*) symbols in configuration files are always ignored. Hidden symbols get their value indirectly from other symbols, via e.g. default and select.

A common source of confusion is opening the output configuration file (zephyr/.config), seeing a bunch of assignments to hidden symbols, and assuming that those assignments must be respected when the configuration is read back in by Kconfig. In reality, all assignments to hidden symbols in zephyr/.config are ignored by Kconfig, like for other configuration files.

To understand why zephyr/.config still includes assignments to hidden symbols, it helps to realize that zephyr/.config serves two separate purposes:

1. It holds the saved configuration, and
2. it holds configuration output. zephyr/.config is parsed by the CMake files to let them query configuration settings, for example.

The assignments to hidden symbols in zephyr/.config are just configuration output. Kconfig itself ignores assignments to hidden symbols when calculating symbol values.

> **Note**
>
> A *minimal configuration*, which can be generated from within the *menuconfig and guiconfig interfaces*, could be considered closer to just a saved configuration, without the full configuration output.

depends on and string/int/hex symbols

depends on works not just for bool symbols, but also for string, int, and hex symbols (and for choices).

The Kconfig definitions below will hide the FOO_DEVICE_FREQUENCY symbol and disable any configuration output for it when FOO_DEVICE is disabled.

```
config FOO_DEVICE
     bool "Foo device"

config FOO_DEVICE_FREQUENCY
     int "Foo device frequency"
     depends on FOO_DEVICE
```

In general, it's a good idea to check that only relevant symbols are ever shown in the menuconfig/guiconfig interface. Having FOO_DEVICE_FREQUENCY show up when FOO_DEVICE is disabled (and possibly hidden) makes the relationship between the symbols harder to understand, even if code never looks at FOO_DEVICE_FREQUENCY when FOO_DEVICE is disabled.

menuconfig symbols

If the definition of a symbol FOO is immediately followed by other symbols that depend on FOO, then those symbols become children of FOO. If FOO is defined with config FOO, then the children are shown indented relative to FOO. Defining FOO with menuconfig FOO instead puts the children in a separate menu rooted at FOO.

menuconfig has no effect on evaluation. It's just a display option.

menuconfig can cut down on the number of menus and make the menu structure easier to navigate. For example, say you have the following definitions:

```
menu "Foo subsystem"

config FOO_SUBSYSTEM
     bool "Foo subsystem"

if FOO_SUBSYSTEM

config FOO_FEATURE_1
     bool "Foo feature 1"

config FOO_FEATURE_2
     bool "Foo feature 2"

config FOO_FREQUENCY
     int "Foo frequency"

... lots of other FOO-related symbols

endif # FOO_SUBSYSTEM
```

(continues on next page)

(continued from previous page)

```
endmenu
```

In this case, it's probably better to get rid of the `menu` and turn `FOO_SUBSYSTEM` into a `menuconfig` symbol:

```
menuconfig FOO_SUBSYSTEM
     bool "Foo subsystem"

if FOO_SUBSYSTEM

config FOO_FEATURE_1
     bool "Foo feature 1"

config FOO_FEATURE_2
     bool "Foo feature 2"

config FOO_FREQUENCY
     int "Foo frequency"

... lots of other FOO-related symbols

endif # FOO_SUBSYSTEM
```

In the `menuconfig` interface, this will be displayed as follows:

```
[*] Foo subsystem  --->
```

Note that making a symbol without children a `menuconfig` is meaningless. It should be avoided, because it looks identical to a symbol with all children invisible:

```
[*] I have no children  ----
[*] All my children are invisible  ----
```

Commas in macro arguments

Kconfig uses commas to separate macro arguments. This means a construct like this will fail:

```
config FOO
    bool
    default y if $(dt_chosen_enabled,"zephyr,bar")
```

To solve this problem, create a variable with the text and use this variable as argument, as follows:

```
DT_CHOSEN_ZEPHYR_BAR := zephyr,bar

config FOO
    bool
    default y if $(dt_chosen_enabled,$(DT_CHOSEN_ZEPHYR_BAR))
```

Note

A variable `DT_COMPAT_VND_MY_DEVICE := vnd,my-device` is automatically created by Zephyr for every `compatible` found in Devicetree bindings; there is no need to define such variables. See *Automatic Kconfig symbols from devicetree* for details.

Checking changes in menuconfig/guiconfig

When adding new symbols or making other changes to Kconfig files, it is a good idea to look up the symbols in *menuconfig or guiconfig* afterwards. To get to a symbol quickly, use the jump-to feature (press /).

Here are some things to check:

- Are the symbols placed in a good spot? Check that they appear in a menu where they make sense, close to related symbols.

  If one symbol depends on another, then it's often a good idea to place it right after the symbol it depends on. It will then be shown indented relative to the symbol it depends on in the `menuconfig` interface, and in a separate menu rooted at the symbol in `guiconfig`. This also works if several symbols are placed after the symbol they depend on.

- Is it easy to guess what the symbols do from their prompts?
- If many symbols are added, do all combinations of values they can be set to make sense?

  For example, if two symbols `FOO_SUPPORT` and `NO_FOO_SUPPORT` are added, and both can be enabled at the same time, then that makes a nonsensical configuration. In this case, it's probably better to have a single `FOO_SUPPORT` symbol.

- Are there any duplicated dependencies?

  This can be checked by selecting a symbol and pressing ? to view the symbol information. If there are duplicated dependencies, then use the `Included via ...` path shown in the symbol information to figure out where they come from.

Checking changes with `scripts/kconfig/lint.py`

After you make Kconfig changes, you can use the scripts/kconfig/lint.py script to check for some potential issues, like unused symbols and symbols that are impossible to enable. Use `--help` to see available options.

Some checks are necessarily a bit heuristic, so a symbol being flagged by a check does not necessarily mean there's a problem. If a check returns a false positive e.g. due to token pasting in C (`CONFIG_FOO_##index##_BAR`), just ignore it.

When investigating an unknown symbol `FOO_BAR`, it is a good idea to run `git grep FOO_BAR` to look for references. It is also a good idea to search for some components of the symbol name with e.g. `git grep FOO` and `git grep BAR`, as it can help uncover token pasting.

Style recommendations and shorthands

This section gives some style recommendations and explains some common Kconfig shorthands.

Factoring out common dependencies If a sequence of symbols/choices share a common dependency, the dependency can be factored out with an `if`.

As an example, consider the following code:

```
config FOO
     bool "Foo"
     depends on DEP

config BAR
     bool "Bar"
     depends on DEP

choice
     prompt "Choice"
```

(continues on next page)

(continued from previous page)

```
    depends on DEP

config BAZ
    bool "Baz"

config QAZ
    bool "Qaz"

endchoice
```

Here, the DEP dependency can be factored out like this:

```
if DEP

config FOO
    bool "Foo"

config BAR
    bool "Bar"

choice
    prompt "Choice"

config BAZ
    bool "Baz"

config QAZ
    bool "Qaz"

endchoice

endif # DEP
```

Note

Internally, the second version of the code is transformed into the first.

If a sequence of symbols/choices with shared dependencies are all in the same menu, the dependency can be put on the menu itself:

```
menu "Foo features"
    depends on FOO_SUPPORT

config FOO_FEATURE_1
    bool "Foo feature 1"

config FOO_FEATURE_2
    bool "Foo feature 2"

endmenu
```

If FOO_SUPPORT is n, the entire menu disappears.

Redundant defaults bool symbols implicitly default to n, and string symbols implicitly default to the empty string. Therefore, default n and default "" are (almost) always redundant.

The recommended style in Zephyr is to skip redundant defaults for `bool` and `string` symbols. That also generates clearer documentation: (*Implicitly defaults to n* instead of *n if <dependencies, possibly inherited>*).

Defaults *should* always be given for `int` and `hex` symbols, however, as they implicitly default to the empty string. This is partly for compatibility with the C Kconfig tools, though an implicit 0 default might be less likely to be what was intended compared to other symbol types as well.

The one case where `default n`/`default ""` is not redundant is when defining a symbol in multiple locations and wanting to override e.g. a `default y` on a later definition. Note that a `default n` does not override a previously defined `default y`.

That is, FOO will be set to `n` in the example below. If the `default n` was omitted in the first definition, FOO would have been set to `y`.

```
config FOO
   bool "foo"
   default n

config FOO
   bool "foo"
   default y
```

In the following example FOO will get the value `y`.

```
config FOO
   bool "foo"
   default y

config FOO
   bool "foo"
   default n
```

Common Kconfig shorthands Kconfig has two shorthands that deal with prompts and defaults.

- `<type> "prompt"` is a shorthand for giving a symbol/choice a type and a prompt at the same time. These two definitions are equal:

```
config FOO
   bool "foo"
```

```
config FOO
   bool
   prompt "foo"
```

  The first style, with the shorthand, is preferred in Zephyr.

- `def_<type> <value>` is a shorthand for giving a type and a value at the same time. These two definitions are equal:

```
config FOO
   def_bool BAR && BAZ
```

```
config FOO
   bool
   default BAR && BAZ
```

Using both the `<type> "prompt"` and the `def_<type> <value>` shorthand in the same definition is redundant, since it gives the type twice.

The `def_<type> <value>` shorthand is generally only useful for symbols without prompts, and somewhat obscure.

> **Note**
>
> For a symbol defined in multiple locations (e.g., in a `Kconfig.defconfig` file in Zephyr), it is best to only give the symbol type for the "base" definition of the symbol, and to use `default` (instead of `def_<type> value`) for the remaining definitions. That way, if the base definition of the symbol is removed, the symbol ends up without a type, which generates a warning that points to the other definitions. That makes the extra definitions easier to discover and remove.

Prompt strings For a Kconfig symbol that enables a driver/subsystem FOO, consider having just "Foo" as the prompt, instead of "Enable Foo support" or the like. It will usually be clear in the context of an option that can be toggled on/off, and makes things consistent.

Style See *Coding Style Guidelines* for style guidelines.

Lesser-known/used Kconfig features

This section lists some more obscure Kconfig behaviors and features that might still come in handy.

The `imply` statement The `imply` statement is similar to `select`, but respects dependencies and doesn't force a value. For example, the following code could be used to enable USB keyboard support by default on the FOO SoC, while still allowing the user to turn it off:

```
config SOC_FOO
    bool
    imply USB_KEYBOARD

...

config USB_KEYBOARD
     bool "USB keyboard support"
```

`imply` acts like a suggestion, whereas `select` forces a value.

Optional prompts A condition can be put on a symbol's prompt to make it optionally configurable by the user. For example, a value `MASK` that's hardcoded to 0xFF on some boards and configurable on others could be expressed as follows:

```
config MASK
     hex "Bitmask" if HAS_CONFIGURABLE_MASK
     default 0xFF
```

> **Note**
>
> This is short for the following:
>
> ```
> config MASK
>  hex
>  prompt "Bitmask" if HAS_CONFIGURABLE_MASK
>  default 0xFF
> ```

The HAS_CONFIGURABLE_MASK helper symbol would get selected by boards to indicate that MASK is configurable. When MASK is configurable, it will also default to 0xFF.

Optional choices Defining a choice with the optional keyword allows the whole choice to be toggled off to select none of the symbols:

```
choice
     prompt "Use legacy protocol"
     optional

config LEGACY_PROTOCOL_1
     bool "Legacy protocol 1"

config LEGACY_PROTOCOL_2
     bool "Legacy protocol 2"

endchoice
```

In the menuconfig interface, this will be displayed e.g. as [*] Use legacy protocol (Legacy protocol 1) --->, where the choice can be toggled off to enable neither of the symbols.

visible if conditions Putting a visible if condition on a menu hides the menu and all the symbols within it, while still allowing symbol default values to kick in.

As a motivating example, consider the following code:

```
menu "Foo subsystem"
     depends on HAS_CONFIGURABLE_FOO

config FOO_SETTING_1
     int "Foo setting 1"
     default 1

config FOO_SETTING_2
     int "Foo setting 2"
     default 2

endmenu
```

When HAS_CONFIGURABLE_FOO is n, no configuration output is generated for FOO_SETTING_1 and FOO_SETTING_2, as the code above is logically equivalent to the following code:

```
config FOO_SETTING_1
     int "Foo setting 1"
     default 1
     depends on HAS_CONFIGURABLE_FOO

config FOO_SETTING_2
     int "Foo setting 2"
     default 2
     depends on HAS_CONFIGURABLE_FOO
```

If we want the symbols to still get their default values even when HAS_CONFIGURABLE_FOO is n, but not be configurable by the user, then we can use visible if instead:

```
menu "Foo subsystem"
     visible if HAS_CONFIGURABLE_FOO
```

(continues on next page)

(continued from previous page)

```
config FOO_SETTING_1
     int "Foo setting 1"
     default 1

config FOO_SETTING_2
     int "Foo setting 2"
     default 2

endmenu
```

This is logically equivalent to the following:

```
config FOO_SETTING_1
     int "Foo setting 1" if HAS_CONFIGURABLE_FOO
     default 1

config FOO_SETTING_2
     int "Foo setting 2" if HAS_CONFIGURABLE_FOO
     default 2
```

> **Note**
>
> See the *optional prompts* section for the meaning of the conditions on the prompts.

When `HAS_CONFIGURABLE_FOO` is `n`, we now get the following configuration output for the symbols, instead of no output:

```
...
CONFIG_FOO_SETTING_1=1
CONFIG_FOO_SETTING_2=2
...
```

Other resources

The *Intro to symbol values* section in the Kconfiglib docstring goes over how symbols values are calculated in more detail.

5.4.5 Custom Kconfig Preprocessor Functions

Kconfiglib supports custom Kconfig preprocessor functions written in Python. These functions are defined in scripts/kconfig/kconfigfunctions.py.

> **Note**
>
> The official Kconfig preprocessor documentation can be found here.

See the Python docstrings in scripts/kconfig/kconfigfunctions.py for detailed documentation. Most of the custom preprocessor functions are used to get devicetree information into Kconfig. For example, the default value of a Kconfig symbol can be fetched from a devicetree `reg` property.

Devicetree-related Functions

The functions listed below are used to get devicetree information into Kconfig. The `*_int` version of each function returns the value as a decimal integer, while the `*_hex` version returns a hexadecimal value starting with `0x`.

```
$(dt_alias_enabled,<node alias>)
$(dt_chosen_bool_prop, <property in /chosen>, <prop>)
$(dt_chosen_enabled,<property in /chosen>)
$(dt_chosen_has_compat,<property in /chosen>,<compatible string>)
$(dt_chosen_label,<property in /chosen>)
$(dt_chosen_partition,addr_hex,<chosen>[,<index>,<unit>])
$(dt_chosen_partition,addr_int,<chosen>[,<index>,<unit>])
$(dt_chosen_path,<property in /chosen>)
$(dt_chosen_reg_addr_hex,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_addr_int,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_size_hex,<property in /chosen>[,<index>,<unit>])
$(dt_chosen_reg_size_int,<property in /chosen>[,<index>,<unit>])
$(dt_class_enabled,<class name>)
$(dt_compat_all_has_prop,<compatible string>,<prop>[,<value>])
$(dt_compat_any_has_prop,<compatible string>,<prop>[,<value>])
$(dt_compat_any_on_bus,<compatible string>,<prop>)
$(dt_compat_enabled,<compatible string>)
$(dt_compat_enabled_num,<compatible string>)
$(dt_compat_on_bus,<compatible string>,<bus>)
$(dt_gpio_hogs_enabled)
$(dt_has_compat,<compatible string>)
$(dt_highest_controller_irq_number,<node path>,<cell>)
$(dt_node_array_prop_has_val,<node path>,<prop>,<value>)
$(dt_node_array_prop_hex,<node path>,<prop>,<index>[,<unit>])
$(dt_node_array_prop_int,<node path>,<prop>,<index>[,<unit>])
$(dt_node_bool_prop,<node path>,<prop>)
$(dt_node_has_compat,<node path>,<compatible string>)
$(dt_node_has_prop,<node path>,<prop>)
$(dt_node_int_prop_hex,<node path>,<prop>[,<unit>])
$(dt_node_int_prop_int,<node path>,<prop>[,<unit>])
$(dt_node_parent,<node path>)
$(dt_node_ph_array_prop_hex,<node path>,<prop>,<index>,<cell>[,<unit>])
$(dt_node_ph_array_prop_int,<node path>,<prop>,<index>,<cell>[,<unit>])
$(dt_node_ph_prop_path,<node path>,<prop>)
$(dt_node_reg_addr_hex,<node path>[,<index>,<unit>])
$(dt_node_reg_addr_int,<node path>[,<index>,<unit>])
$(dt_node_reg_size_hex,<node path>[,<index>,<unit>])
$(dt_node_reg_size_int,<node path>[,<index>,<unit>])
$(dt_node_str_prop_equals,<node path>,<prop>,<value>)
$(dt_nodelabel_array_prop_has_val, <node label>, <prop>, <value>)
$(dt_nodelabel_bool_prop,<node label>,<prop>)
$(dt_nodelabel_enabled,<node label>)
$(dt_nodelabel_enabled_with_compat,<node label>,<compatible string>)
$(dt_nodelabel_exists,<node label>)
$(dt_nodelabel_has_compat,<node label>,<compatible string>)
$(dt_nodelabel_has_prop,<node label>,<prop>)
$(dt_nodelabel_path,<node label>)
$(dt_nodelabel_reg_addr_hex,<node label>[,<index>,<unit>])
$(dt_nodelabel_reg_addr_int,<node label>[,<index>,<unit>])
$(dt_nodelabel_reg_size_hex,<node label>[,<index>,<unit>])
$(dt_nodelabel_reg_size_int,<node label>[,<index>,<unit>])
$(dt_path_enabled,<node path>)
```

(continues on next page)

(continued from previous page)

```
$(dt_partition_mtd,<node path>)
```

Integer functions

The functions listed below can be used to do arithmetic operations on integer variables, such as addition, subtraction and more. Functions with and without the `_hex` suffix in their names return hexadecimal and decimal values respectively.

```
$(add,<value>[,value]...)
$(add_hex,<value>[,value]...)
$(dec,<value>[,value]...)
$(dec_hex,<value>[,value]...)
$(div,<value>[,value]...)
$(div_hex,<value>[,value]...)
$(inc,<value>[,value]...)
$(inc_hex,<value>[,value]...)
$(max,<value>[,value]...)
$(max_hex,<value>[,value]...)
$(min,<value>[,value]...)
$(min_hex,<value>[,value]...)
$(mod,<value>[,value]...)
$(mod_hex,<value>[,value]...)
$(mul,<value>[,value]...)
$(mul_hex,<value>[,value]...)
$(sub,<value>[,value]...)
$(sub_hex,<value>[,value]...)
```

String functions

The functions listed below can be used to modify string variables.

```
$(normalize_upper,<string>)
$(substring,<string>,<start>[,<stop>])
```

Other functions

Functions to perform specific operations, currently only a check if a shield name is specified.

```
$(shields_list_contains,<shield name>)
```

Shield names cannot contain whitespace. A space after the comma, as in `$(shields_list_contains, foo)`, is stripped and a warning is printed so the lookup still matches `foo`.

Example Usage Assume that the devicetree for some board looks like this:

```
{
    soc {
        #address-cells = <1>;
        #size-cells = <1>;

        spi0: spi@10014000 {
            compatible = "sifive,spi0";
            reg = <0x10014000 0x1000 0x20010000 0x3c0900>;
            reg-names = "control", "mem";
```

(continues on next page)

(continued from previous page)

```
                    ...
            };
};
```

The second entry in `reg` in `spi@1001400` (`<0x20010000 0x3c0900>`) corresponds to `mem`, and has the address `0x20010000`. This address can be inserted into Kconfig as follows:

```
config FLASH_BASE_ADDRESS
     default $(dt_node_reg_addr_hex,/soc/spi@1001400,1)
```

After preprocessor expansion, this turns into the definition below:

```
config FLASH_BASE_ADDRESS
     default 0x20010000
```

5.4.6 Kconfig extensions

Zephyr uses the Kconfiglib implementation of Kconfig, which includes some Kconfig extensions:

- Default values can be applied to existing symbols without *weakening* the symbols dependencies through the use of `configdefault`.

  ```
  config FOO
      bool "FOO"
      depends on BAR

  configdefault FOO
      default y if FIZZ
  ```

  The statement above is equivalent to:

  ```
  config FOO
      bool "Foo"
      default y if FIZZ
      depends on BAR
  ```

  `configdefault` symbols cannot contain any fields other than `default`, however they can be wrapped in `if` statements. The two statements below are equivalent:

  ```
  configdefault FOO
      default y if BAR

  if BAR
  configdefault FOO
      default y
  endif # BAR
  ```

- Environment variables in `source` statements are expanded directly, meaning no "bounce" symbols with `option env="ENV_VAR"` need to be defined.

  > **Note**
  >
  > `option env` has been removed from the C tools as of Linux 4.18 as well.

  The recommended syntax for referencing environment variables is `$(FOO)` rather than `$FOO`. This uses the new Kconfig preprocessor. The `$FOO` syntax for expanding environment variables is only supported for backwards compatibility.

- The `source` statement supports glob patterns and includes each matching file. A pattern is required to match at least one file.

  Consider the following example:

  ```
  source "foo/bar/*/Kconfig"
  ```

  If the pattern `foo/bar/*/Kconfig` matches the files `foo/bar/baz/Kconfig` and `foo/bar/qaz/Kconfig`, the statement above is equivalent to the following two `source` statements:

  ```
  source "foo/bar/baz/Kconfig"
  source "foo/bar/qaz/Kconfig"
  ```

  If no files match the pattern, an error is generated.

  The wildcard patterns accepted are the same as for the Python glob module.

  For cases where it's okay for a pattern to match no files (or for a plain filename to not exist), a separate `osource` (*optional source*) statement is available. `osource` is a no-op if no file matches.

  > **Note**
  >
  > `source` and `osource` are analogous to `include` and `-include` in Make.

- An `rsource` statement is available for including files specified with a relative path. The path is relative to the directory of the `Kconfig` file that contains the `rsource` statement.

  As an example, assume that `foo/Kconfig` is the top-level `Kconfig` file, and that `foo/bar/Kconfig` has the following statements:

  ```
  source "qaz/Kconfig1"
  rsource "qaz/Kconfig2"
  ```

  This will include the two files `foo/qaz/Kconfig1` and `foo/bar/qaz/Kconfig2`.

  `rsource` can be used to create `Kconfig` "subtrees" that can be moved around freely.

  `rsource` also supports glob patterns.

  A drawback of `rsource` is that it can make it harder to figure out where a file gets included, so only use it if you need it.

- An `orsource` statement is available that combines `osource` and `rsource`.

  For example, the following statement will include `Kconfig1` and `Kconfig2` from the current directory (if they exist):

  ```
  orsource "Kconfig[12]"
  ```

- `def_int`, `def_hex`, and `def_string` keywords are available, analogous to `def_bool`. These set the type and add a `default` at the same time.

- A symbol name can be associated to `choice` groups using the `choice <symbol>` syntax. Such choices, called *named choices*, can be modified from places other than their initial definition. For example, the following statements define the named choice `FOOBAR`:

  ```
  choice FOOBAR
          prompt "Example choice"
          default BAR

  config FOO
  ```

(continues on next page)

(continued from previous page)

```
        bool "Foo"

config BAR
        bool "Bar"

endchoice
```

The following statements could then be used, in a `Kconfig.defconfig` file for example, to override the default option of choice `FOOBAR`:

```
# Note how "prompt" is not present here
choice FOOBAR
    default FOO
endchoice
```

> **Note**
>
> The *named choices* feature originates from Linux, but it is no longer supported in Linux since kernel release 6.9, and has thus become a Kconfiglib language extension.

Users interested in optimizing their configuration for security should refer to the Zephyr Security Guide's section on the *Hardening Tool*.

5.5 Snippets

Snippets are a way to save build system settings in one place, and then use those settings when you build any Zephyr application. This lets you save common configuration separately when it applies to multiple different applications.

Some example use cases for snippets are:

- changing your board's console backend from a "real" UART to a USB CDC-ACM UART
- enabling frequently-used debugging options
- applying interrelated configuration settings to your "main" CPU and a co-processor core on an AMP SoC

The following pages document this feature.

5.5.1 Using Snippets

> **Tip**
>
> See *Built-in snippets* for a list of snippets that are provided by Zephyr.

Snippets have names. You use snippets by giving their names to the build system.

With west build

To use a snippet named `foo` when building an application `app`:

```
west build -S foo app
```

To use multiple snippets:

```
west build -S snippet1 -S snippet2 [...] app
```

With cmake

If you are running CMake directly instead of using `west build`, use the `SNIPPET` variable. This is a whitespace- or semicolon-separated list of snippet names you want to use. For example:

```
cmake -Sapp -Bbuild -DSNIPPET="snippet1;snippet2" [...]
cmake --build build
```

5.5.2 Built-in snippets

Zephyr Bluetooth LE Controller (bt-ll-sw-split)

You can build with this snippet by following the instructions in *the snippets usage page*. When building with `west`, you can do:

```
west build -S bt-ll-sw-split [...]
```

Overview This selects the Zephyr Bluetooth LE Controller.

Requirements Hardware support for:

- `CONFIG_BT`

CDC-ACM Console Snippet (cdc-acm-console)

```
west build -S cdc-acm-console [...]
```

Overview This snippet redirects serial console output to a CDC ACM UART. The USB device which should be used is configured using *Devicetree*.

Requirements Hardware support for:

- `CONFIG_USB_DEVICE_STACK`
- `CONFIG_SERIAL`
- `CONFIG_CONSOLE`
- `CONFIG_UART_CONSOLE`
- `CONFIG_UART_LINE_CTRL`

A devicetree node with node label `zephyr_udc0` that points to an enabled USB device node with driver support. This should look roughly like this in *your devicetree*:

```
zephyr_udc0: usbd@deadbeef {
     compatible = "vnd,usb-device";
     /* ... */
};
```

Espressif snippets

Espressif Flash 128MB

Overview This snippet allows users to select the 128MB SPI flash variant of the ESP32 chips.

Espressif Flash 16MB

Overview This snippet allows users to select the 16MB SPI flash variant of the ESP32 chips.

Espressif Flash 2MB

Overview This snippet allows users to select the 2MB SPI flash variant of the ESP32 chips.

Espressif Flash 32MB

Overview This snippet allows users to select the 32MB SPI flash variant of the ESP32 chips.

Espressif Flash 4MB

Overview This snippet allows users to select the 4MB SPI flash variant of the ESP32 chips.

Espressif Flash 64MB

Overview This snippet allows users to select the 64MB SPI flash variant of the ESP32 chips.

Espressif Flash 8MB

Overview This snippet allows users to select the 8MB SPI flash variant of the ESP32 chips.

Espressif PSRAM 2MB

Overview This snippet allows users to select the 2MB SPI PSRAM variant of the ESP32 chips.

Espressif PSRAM 4MB

Overview This snippet allows users to select the 4MB SPI PSRAM variant of the ESP32 chips.

Espressif PSRAM 8MB

Overview This snippet allows users to select the 8MB SPI PSRAM variant of the ESP32 chips.

Espressif PSRAM relocations

Overview This snippet allows users to enable the flash sections to be relocated to PSRAM.

Espressif PSRAM Wi-Fi buffers

Overview This snippet allows users to configure Wi-Fi buffers to use PSRAM.

Native Simulator support for hci_uart Snippet (hci-uart-native-sim)

```
west build -S hci-uart-native-sim [...]
```

Overview This snippet allows to use hci_uart connected to the host computer with the Native Simulator. It is useful for testing with a real Bluetooth controller, such as a device running Zephyr.

Nordic snippets

Nordic FLPR snippet with execution in place (nordic-flpr-xip)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic FLPR (Fast Lightweight Peripheral Processor) from application core. FLPR code is to be executed from RRAM, so the FLPR image must be built for the `xip` board variant, or with `CONFIG_XIP` enabled.

Nordic FLPR snippet with execution from SRAM (nordic-flpr)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic FLPR (Fast Lightweight Peripheral Processor) from application core. FLPR code is to be executed from SRAM, so the FLPR image must be built without the `xip` board variant, or with `CONFIG_XIP` disabled.

Nordic Dictionary-based STM logging snippet (nordic-log-stm-dict)

Overview This snippet allows users to build Zephyr with the dictionary-based logging to the Coresight STM stimulus ports. Data is collected in ETR buffer. Data from ETR buffer is output on UART.

Nordic Dictionary-based STM to TPIU logging snippet (nordic-log-stm-tpiu-dict)

Overview This snippet allows users to build Zephyr with the dictionary-based logging to the Coresight STM stimulus ports. Data is written to the TPIU interface and can be captured with nrfutil trace to translate into a human-readable format.

Nordic Standalone STM logging snippet (nordic-log-stm)

Overview This snippet allows users to build Zephyr with the logging to the Coresight STM stimulus ports. Data is collected in ETR buffer. Data from ETR buffer is decoded and human-readable data is output on the UART.

Nordic boot PPR snippet with execution in place (nordic-ppr-xip)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic PPR (Peripheral Processor) from another core. PPR code is to be executed from MRAM, so the PPR image must be built for the `xip` board variant, or with `CONFIG_XIP` enabled.

Nordic PPR snippet (nordic-ppr)

Overview This snippet allows users to build Zephyr with the capability to boot Nordic PPR (Peripheral Processor) from another core.

NUS Console Snippet (nus-console)

```
west build -S nus-console [...]
```

Overview This snippet redirects serial console output to a UART over NUS (Bluetooth LE) instance. The Bluetooth Serial device used shall be configured using *Devicetree*.

Requirements Hardware support for:

- CONFIG_BT
- CONFIG_BT_PERIPHERAL
- CONFIG_BT_ZEPHYR_NUS
- CONFIG_SERIAL
- CONFIG_CONSOLE
- CONFIG_UART_CONSOLE

RAM Console Snippet (ram-console)

```
west build -S ram-console [...]
```

Overview This snippet redirects console output to a RAM buffer. The RAM console buffer is a global array located in RAM region by default, whose address is unknown before building. The RAM console driver also supports using a dedicated section for the RAM console buffer with prefined address.

How to enable RAM console buffer section Add board dts overlay to this snippet to add property `zephyr,ram-console` in the chosen node and memory-region node with compatible string `zephyr,memory-region` as the following:

```
/ {
    chosen {
        zephyr,ram-console = &snippet_ram_console;
    };

   snippet_ram_console: memory@93d00000 {
        compatible = "zephyr,memory-region";
        reg = <0x93d00000 DT_SIZE_K(4)>;
        zephyr,memory-region = "RAM_CONSOLE";
   };
};
```

RAM Tracing Snippet (ram-tracing)

```
west build -S ram-tracing [...]
```

Overview This snippet enables CTF Tracing using the RAM backend.

After connecting to the target in GDB, you can dump the RAM tracing data to a binary file:

```
(gdb) dump binary memory ram_tracing.bin ram_tracing ram_tracing+8192
```

Using Babeltrace[1], you can view the binary file as a CTF trace:

[1] https://babeltrace.org/

```
[19:00:00.023530800] (+?.?????????) thread_create: { thread_id = 2147621216, name␣
↪= "unknown" }
[19:00:00.023567100] (+0.000036300) thread_info: { thread_id = 2147621216, name =
↪"unknown", stack_base = 2147705856, stack_size = 3072 }
[19:00:00.023673700] (+0.000106600) thread_name_set: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.023731100] (+0.000057400) thread_wakeup: { thread_id = 2147621216, name␣
↪= "test_abort_from_isr" }
[19:00:00.023827500] (+0.000096400) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.023904500] (+0.000077000) thread_switched_in: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.024807600] (+0.000903100) thread_create: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.024843800] (+0.000036200) thread_info: { thread_id = 2147619320, name =
↪"unknown", stack_base = 2147693568, stack_size = 3072 }
[19:00:00.024898700] (+0.000054900) thread_wakeup: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.025020000] (+0.000121300) thread_switched_out: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.025130700] (+0.000110700) thread_switched_in: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.025249900] (+0.000119200) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.025353500] (+0.000103600) thread_switched_in: { thread_id = 2147619320,␣
↪name = "unknown" }
[19:00:00.026049900] (+0.000696400) thread_switched_in: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.026759200] (+0.000709300) semaphore_give_enter: { id = 2147735664 }
[19:00:00.026773600] (+0.000014400) semaphore_give_exit: { id = 2147735664 }
[19:00:00.027346600] (+0.000573000) thread_switched_out: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.027457400] (+0.000110800) thread_switched_in: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.029552800] (+0.002095400) thread_create: { thread_id = 2147621216, name␣
↪= "unknown" }
[19:00:00.029589000] (+0.000036200) thread_info: { thread_id = 2147621216, name =
↪"unknown", stack_base = 2147705856, stack_size = 3072 }
[19:00:00.029697400] (+0.000108400) thread_name_set: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.029754800] (+0.000057400) thread_wakeup: { thread_id = 2147621216, name␣
↪= "test_abort_from_isr" }
[19:00:00.029851200] (+0.000096400) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.029928200] (+0.000077000) thread_switched_in: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.029968400] (+0.000040200) semaphore_init: { id = 2147623436, ret = 0 }
[19:00:00.030851900] (+0.000883500) thread_create: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.030888100] (+0.000036200) thread_info: { thread_id = 2147619320, name =
↪"unknown", stack_base = 2147693568, stack_size = 3072 }
[19:00:00.030943000] (+0.000054900) thread_wakeup: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.030989600] (+0.000046600) semaphore_take_enter: { id = 2147623436,␣
↪timeout = 4294957296 }
[19:00:00.031005800] (+0.000016200) semaphore_take_blocking: { id = 2147623436,␣
↪timeout = 4294957296 }
```

(continues on next page)

(continued from previous page)

```
[19:00:00.031099600] (+0.000093800) thread_switched_out: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.031210300] (+0.000110700) thread_switched_in: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.031329500] (+0.000119200) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.031433100] (+0.000103600) thread_switched_in: { thread_id = 2147619320,␣
↪name = "unknown" }
[19:00:00.031469100] (+0.000036000) semaphore_give_enter: { id = 2147623436 }
```

References

RP2040/RP2350 boot mode retention setup (rp2-boot-mode-retention)

Overview This snippet sets up an RP2040/RP2350 based board for retaining the set boot mode via properly configured retained mem in SRAM, enabling the necessary Kconfig symbols for the retained mem drivers and retention system.

RTT Console Snippet (rtt-console)

Overview This snippet enables console output to SEGGER RTT.

Requirements Support for:

- `CONFIG_HAS_SEGGER_RTT`

SystemView RTT Tracing Snippet (rtt-tracing)

```
west build -S rtt-tracing [...]
```

Overview This snippet enables SEGGER SystemView RTT support with the tracing subsystem.

Requirements Hardware support for:

- `CONFIG_HAS_SEGGER_RTT`

Semihost Tracing Snippet (semihost-tracing)

```
west build -S semihost-tracing [...]
```

Overview This snippet enables CTF Tracing using semihosting on a target that supports it.

Using Qemu, you can run the example application and generate a CTF trace file using semihosting. The trace file can then be viewed using Babeltrace[1].

```
(gdb) dump binary memory ram_tracing.bin ram_tracing ram_tracing+8192
```

Using Babeltrace[1], you can view the binary file as a CTF trace:

[1] https://babeltrace.org/

```
[19:00:00.023530800] (+?.?????????) thread_create: { thread_id = 2147621216, name␣
↪= "unknown" }
[19:00:00.023567100] (+0.000036300) thread_info: { thread_id = 2147621216, name =
↪"unknown", stack_base = 2147705856, stack_size = 3072 }
[19:00:00.023673700] (+0.000106600) thread_name_set: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.023731100] (+0.000057400) thread_wakeup: { thread_id = 2147621216, name␣
↪= "test_abort_from_isr" }
[19:00:00.023827500] (+0.000096400) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.023904500] (+0.000077000) thread_switched_in: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.024807600] (+0.000903100) thread_create: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.024843800] (+0.000036200) thread_info: { thread_id = 2147619320, name =
↪"unknown", stack_base = 2147693568, stack_size = 3072 }
[19:00:00.024898700] (+0.000054900) thread_wakeup: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.025020000] (+0.000121300) thread_switched_out: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.025130700] (+0.000110700) thread_switched_in: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.025249900] (+0.000119200) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.025353500] (+0.000103600) thread_switched_in: { thread_id = 2147619320,␣
↪name = "unknown" }
[19:00:00.026049900] (+0.000696400) thread_switched_in: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.026759200] (+0.000709300) semaphore_give_enter: { id = 2147735664 }
[19:00:00.026773600] (+0.000014400) semaphore_give_exit: { id = 2147735664 }
[19:00:00.027346600] (+0.000573000) thread_switched_out: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.027457400] (+0.000110800) thread_switched_in: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.029552800] (+0.002095400) thread_create: { thread_id = 2147621216, name␣
↪= "unknown" }
[19:00:00.029589000] (+0.000036200) thread_info: { thread_id = 2147621216, name =
↪"unknown", stack_base = 2147705856, stack_size = 3072 }
[19:00:00.029697400] (+0.000108400) thread_name_set: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.029754800] (+0.000057400) thread_wakeup: { thread_id = 2147621216, name␣
↪= "test_abort_from_isr" }
[19:00:00.029851200] (+0.000096400) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.029928200] (+0.000077000) thread_switched_in: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.029968400] (+0.000040200) semaphore_init: { id = 2147623436, ret = 0 }
[19:00:00.030851900] (+0.000883500) thread_create: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.030888100] (+0.000036200) thread_info: { thread_id = 2147619320, name =
↪"unknown", stack_base = 2147693568, stack_size = 3072 }
[19:00:00.030943000] (+0.000054900) thread_wakeup: { thread_id = 2147619320, name␣
↪= "unknown" }
[19:00:00.030989600] (+0.000046600) semaphore_take_enter: { id = 2147623436,␣
↪timeout = 4294957296 }
[19:00:00.031005800] (+0.000016200) semaphore_take_blocking: { id = 2147623436,␣
↪timeout = 4294957296 }
```

(continues on next page)

(continued from previous page)

```
[19:00:00.031099600] (+0.000093800) thread_switched_out: { thread_id = 2147621216,␣
↪name = "test_abort_from_isr" }
[19:00:00.031210300] (+0.000110700) thread_switched_in: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.031329500] (+0.000119200) thread_switched_out: { thread_id = 2147621968,␣
↪name = "main" }
[19:00:00.031433100] (+0.000103600) thread_switched_in: { thread_id = 2147619320,␣
↪name = "unknown" }
[19:00:00.031469100] (+0.000036000) semaphore_give_enter: { id = 2147623436 }
```

References

UART Console Snippet (uart-console)

Overview This snippet enables console output the chosen UART in `zephyr,console`.

Silicon Labs Packet Trace Interface (silabs-pti)

Overview This snippet allows users to build Zephyr applications for Silicon Labs Series 2 devices where radio packets are emitted over the Packet Trace Interface for use by debugging tools.

```
west build -S silabs-pti [...]
```

Requirements Hardware support for `silabs,pti`.

A pinctrl configuration with nodelabel `pti_default` containing PTI pinout.

Slot1 partition snippet (slot1-partition)

```
west build -S slot1-partition [...]
```

Overview This snippet changes the chosen flash partition to be the `slot1_partition` node, this can be used to build for the alternate slot in MCUboot direct-xip or firmware loader modes.

Requirements Partition mapping correct setup in devicetree, for example:

```
&flash0 {
        partitions {
                ranges;
                #address-cells = <1>;
                #size-cells = <1>;

                boot_partition: partition@0 {
                        compatible = "zephyr,mapped-partition";
                        label = "mcuboot";
                        reg = <0x00000000 0x0000c000>;
                };

                slot0_partition: partition@c000 {
                        compatible = "zephyr,mapped-partition";
                        label = "image-0";
                        reg = <0x0000c000 0x00076000>;
                };
```

(continues on next page)

(continued from previous page)

```
                slot1_partition: partition@82000 {
                        compatible = "zephyr,mapped-partition";
                        label = "image-1";
                        reg = <0x00082000 0x00076000>;
                };

                /*
                 * The flash starting at 0x000f8000 and ending at
                 * 0x000fffff is reserved for use by the application.
                 */

                /*
                 * Storage partition will be used by FCB/LittleFS/NVS
                 * if enabled.
                 */
                storage_partition: partition@f8000 {
                        compatible = "zephyr,mapped-partition";
                        label = "storage";
                        reg = <0x000f8000 0x00008000>;
                };
        };
};
```

SocketCAN on Native Simulator Snippet (socketcan-native-sim)

```
west build -S socketcan-native-sim [...]
```

Overview This snippet enables Linux SocketCAN support on native_sim.

By default, the native simulator expects a SocketCAN network device called `zcan0` (specified in boards/native/native_sim/native_sim.dts). This name can be added as an alternative name for an existing SocketCAN network device (here, a newly created virtual CAN network device `vcan0`) using a command like the following:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link property add dev vcan0 altname zcan0
```

The SocketCAN device must be configured and brought up before running the native simulator:

```
sudo ip link set vcan0 up
```

Texas Instruments snippets

TI I/O Expander

Overview This snippet enables I/O expander functionality on TI boards

Supported boards

- AM62L EVM

Usage To enable I/O expander functionality, add this snippet to your build:

```
# From the root of the zephyr repository
west build -b am62l_evm/am62l3/a53 --shield ti-io-exp samples/drivers/pwm/capture
```

USB/IP on Native Simulator Snippet (usbip-native-sim)

```
west build -S usbip-native-sim [...]
```

Overview This snippet enables USB/IP support on a native simulator. When building with this snippet, you need to provide DTC overlays using the EXTRA_DTC_OVERLAY_FILE argument.

This snippet is experimental, the behavior may change without notice.

Video Native FIFO Snippet (video-native-fifo)

```
west build -b native_sim/native/64 -S video-native-fifo [...]
```

Overview This snippet instantiates the native_sim host FIFO video source as the `zephyr,camera` *Devicetree* chosen node.

The FIFO path defaults to `/tmp/zephyr-video-fifo-<pid>.fifo`, and is printed when streaming starts. The application selects the resolution and pixel format, 320x240 RGB565 by default.

See the driver documentation for how to set the FIFO path and feed it from a webcam or a test pattern, and for the details of the driver behaviour.

Requirements A native_sim target, and a host process writing raw frames of the selected format to the FIFO. Sufficient memory for the video resolution must be declared by `CONFIG_VIDEO_BUFFER_POOL_HEAP_SIZE`.

Video Software Generator Snippet (video-sw-generator)

```
west build -S video-sw-generator [...]
```

Overview This snippet instantiate a fake video source generating a test pattern continuously for test purpose. It is selected as the `zephyr,camera` *Devicetree* chosen node.

Requirements No hardware support is required besides sufficient memory for the video resolution declared by `CONFIG_VIDEO_BUFFER_POOL_HEAP_SIZE`.

Wi-Fi snippets

Wi-Fi Credentials Snippet (wifi-credentials)

```
west build  -S wifi-credentials [...]
```

Can also be used along with the *Wi-Fi Enterprise Snippet (wifi-enterprise)* snippet.

```
west build  -S "wifi-enterprise,wifi-credentials" [...]
```

Overview This snippet enables Wi-Fi credentials support.

Requirements Hardware support for:

- CONFIG_WIFI
- CONFIG_WIFI_USE_NATIVE_NETWORKING
- CONFIG_WIFI_NM_WPA_SUPPLICANT
- CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_ENTERPRISE

Wi-Fi Enterprise Snippet (wifi-enterprise)

```
west build  -S wifi-enterprise [...]
```

Can also be used along with the *Wi-Fi IPv4 Snippet (wifi-ipv4)* snippet.

```
west build  -S "wifi-enterprise,wifi-ipv4" [...]
```

Overview This snippet enables enterprise Wi-Fi support.

See *Wi-Fi Management* for more information on the usage.

Requirements Hardware support for:

- CONFIG_WIFI
- CONFIG_WIFI_USE_NATIVE_NETWORKING
- CONFIG_WIFI_NM_WPA_SUPPLICANT
- CONFIG_WIFI_NM_WPA_SUPPLICANT_CRYPTO_ENTERPRISE

Wi-Fi IPv4 and IPv6 Snippet (wifi-ip)

```
west build -S wifi-ip [...]
```

Overview This snippet enables IPv4 and IPv6 Wi-Fi support.

Use Wi-Fi shell to connect to the Wi-Fi network:

```
wifi connect -s <SSID> -k <key_management> -p <passphrase>
```

Requirements Hardware support for:

- CONFIG_WIFI
- CONFIG_WIFI_USE_NATIVE_NETWORKING

Wi-Fi IPv4 Snippet (wifi-ipv4)

```
west build -S wifi-ipv4 [...]
```

Overview This snippet enables IPv4 Wi-Fi support.

Use Wi-Fi shell to connect to the Wi-Fi network:

```
wifi connect -s <SSID> -k <key_management> -p <passphrase>
```

Requirements Hardware support for:

- CONFIG_WIFI
- CONFIG_WIFI_USE_NATIVE_NETWORKING

Wi-Fi IPv6 Snippet (wifi-ipv6)

```
west build -S wifi-ipv6 [...]
```

Overview This snippet enables IPv6 Wi-Fi support.

Use Wi-Fi shell to connect to the Wi-Fi network:

```
wifi connect -s <SSID> -k <key_management> -p <passphrase>
```

Requirements Hardware support for:

- `CONFIG_WIFI`
- `CONFIG_WIFI_USE_NATIVE_NETWORKING`

Xen Dom0: universal snippet for XEN control domain

Overview This snippet allows user to build Zephyr as a Xen initial domain (Dom0). The feature is implemented as configuration snippet to allow support for any compatible platform.

How to add support of a new board

- add board dt overlay to this snippet which deletes UART nodes.
- in application dt overlay add correct memory and hypervisor nodes, based on regions Xen picked for Domain-0 on your setup.

Programming Correct snippet designation for Xen must be entered when you invoke `west build`. For example:

```
west build -b qemu_cortex_a53 -S xen-dom0 samples/synchronization
```

QEMU example with Xen Overlay for qemu_cortex_a53 board, that is present in snippets/xen-dom0/boards/ directory of this snippet is QEMU Xen control domain example.

To run such setup, you need to:

- fetch and build Xen (e.g. RELEASE-4.17.0) for arm64 platform
- take and compile sample device tree from `example/` directory
- build your Zephyr sample/application with `xen-dom0` snippet and start it as Xen control domain

Important notice regarding device tree Xen allocates addresses used for Dom0 image and grant tables/extended regions dynamically so they may differ on specific setups (they depend on memory banks location and size, application image size, hypervisor load address used by bootloader etc.). Since Zephyr does not parse device tree in runtime and is not a position independent executable, it needs to know exact values on build time. To make it you need to build your final binary, try to boot it with Xen and analyze the hypervisor output to get exact addresses for your setup. They will look as follows:

Grant table region (first reg in hypervisor node):

```
(XEN) Grant table range: 0x00000088080000-0x000000880c0000
```

Extended regions (2nd and further regs in hypervisor node):

```
(XEN) [    0.928173] d0: extended region 2: 0x40000000->0x47e00000
```

Zephyr Dom0 memory:

```
(XEN) Allocating 1:1 mappings for dom0:
(XEN) BANK[0] 0x00000060000000-0x00000070000000 (256MB)
```

Device tree nodes, that should be added by application overlay for provided example (based on R-Car H3ULCB with extended region):

```
hypervisor: hypervisor@88080000 {
    compatible = "xen,xen";
    reg = <0x0 0x88080000 0x0 0x40000 0x0 0x40000000 0x0 0x7e00000>;
    interrupts = <GIC_PPI 0x0 IRQ_TYPE_EDGE IRQ_DEFAULT_PRIORITY>;
    interrupt-parent = <&gic>;
    status = "okay";
};

ram: memory@60000000 {
    device_type = "mmio-sram";
    reg = <0x00 0x60000000 0x00 DT_SIZE_M(256)>;
};
```

Xen extended regions notice By default, support for Xen extended regions is disabled. Therefore, after specifying the extended regions in the hypervisor node in the device tree, it is also required to set the XEN_REGIONS config to 'yes' in the application overlay file.

Start the example For starting you can use QEMU from Zephyr SDK by following command:

```
<path to Zephyr SDK>/sysroots/x86_64-pokysdk-linux/usr/bin/qemu-system-aarch64 -
↪cpu cortex-a53 \
-m 6G -nographic -machine virt,gic-version=3,virtualization=true -chardev␣
↪stdio,id=con,mux=on \
-serial chardev:con -mon chardev=con,mode=readline -pidfile qemu.pid \
-device loader,file=<path to Zephyr app build>/zephyr.bin,addr=0x40600000 \
-dtb <path to DTB>/xen.dtb -kernel <path to Xen build>/xen
```

This will start you a Xen hypervisor with your application as Xen control domain. To make it usable, you can add zephyr-xenlib by Xen-troops library to your project. It'll provide basic domain management functionalities - domain creation and configuration.

XIAO UART/Serial Console Snippet (xiao-serial-console)

Overview This snippet enables console output over the standard XIAO UART pins, setting `zephyr,console` to `&xiao_serial`.

5.5.3 Writing Snippets

- *Basics*
- *Namespacing*
- *Where snippets are located*
- *Processing order*

- *Devicetree overlays (.overlay)*
- *.conf files*
- *Sysbuild .conf files*
- *DTS_EXTRA_CPPFLAGS*
- *Board-specific settings*
  - *By name*
  - *By regular expression*
  - *Board revisions*

Basics

Snippets are defined using YAML files named `snippet.yml`.

A `snippet.yml` file contains the name of the snippet, along with additional build system settings, like this:

```
name: snippet-name
# ... build system settings go here ...
```

Build system settings go in other keys in the file as described later on in this page.

You can combine settings whenever they appear under the same keys. For example, you can combine a snippet-specific devicetree overlay and a `.conf` file like this:

```
name: foo
append:
  EXTRA_DTC_OVERLAY_FILE: foo.overlay
  EXTRA_CONF_FILE: foo.conf
```

In addition, snippets can also be applied to sysbuild configuration like so:

```
name: foo
append:
  SB_EXTRA_CONF_FILE: sb.conf
  EXTRA_CONF_FILE: app.conf
```

Namespacing

When writing devicetree overlays in a snippet, use `snippet_<name>` or `snippet-<name>` as a namespace prefix when choosing names for node labels, node names, etc. This avoids namespace conflicts.

For example, if your snippet is named `foo-bar`, write your devicetree overlays like this:

```
chosen {
        zephyr,baz = &snippet_foo_bar_dev;
};

snippet_foo_bar_dev: device@12345678 {
        /* ... */
};
```

Where snippets are located

The build system looks for snippets in these places:

1. In directories configured by the **SNIPPET_ROOT** CMake variable. This always includes the zephyr repository (so snippets/ is always a source of snippets).

   Additional directories can be added manually at CMake time.

   The variable is a whitespace- or semicolon-separated list of directories which may contain snippet definitions.

   For each directory in the list, the build system looks for `snippet.yml` files underneath a subdirectory named `snippets/`, if one exists.

   For example, if **SNIPPET_ROOT** is set to `/foo;/bar`, the build system will look for `snippet.yml` files underneath the following subdirectories:

   - `/foo/snippets/`
   - `/bar/snippets/`

   The `snippet.yml` files can be nested anywhere underneath these locations.

2. In any *module* whose `module.yml` file provides a `snippet_root` setting.

   For example, in a zephyr module named `baz`, you can add this to your `module.yml` file:

   ```
   settings:
     snippet_root: .
   ```

   And then any `snippet.yml` files in `baz/snippets` will automatically be discovered by the build system, just as if the path to `baz` had appeared in **SNIPPET_ROOT**.

Processing order

Snippets are processed in the order they are listed in the **SNIPPET** variable, or in the order of the `-S` arguments if using west.

To apply `bar` after `foo`:

```
cmake -Sapp -Bbuild -DSNIPPET="foo;bar" [...]
cmake --build build
```

The same can be achieved with west as follows:

```
west build -S foo -S bar [...] app
```

When multiple snippets set the same configuration, the configuration value set by the last processed snippet ends up in the final configurations.

For instance, if `foo` sets `CONFIG_FOO=1` and `bar` sets `CONFIG_FOO=2` in the above example, the resulting final configuration will be `CONFIG_FOO=2` because `bar` is processed after `foo`.

This principle applies to both Kconfig fragments (`.conf` files) and devicetree overlays (`.overlay` files).

Devicetree overlays (`.overlay`)

This `snippet.yml` adds `foo.overlay` to the build:

```
name: foo
append:
  EXTRA_DTC_OVERLAY_FILE: foo.overlay
```

The path to `foo.overlay` is relative to the directory containing `snippet.yml`. Multiple `.overlay` files can also be provided as a list:

```
name: foo
append:
  EXTRA_DTC_OVERLAY_FILE:
    - foo.overlay
    - bar.overlay
```

`.conf` files

This `snippet.yml` adds `foo.conf` to the build:

```
name: foo
append:
  EXTRA_CONF_FILE: foo.conf
```

The path to `foo.conf` is relative to the directory containing `snippet.yml`. Multiple `.conf` files can also be provided as a list.

Sysbuild `.conf` files

This `snippet.yml` adds `foo.conf` to the sysbuild configuration:

```
name: foo
append:
  SB_EXTRA_CONF_FILE: foo.conf
```

The path to `foo.conf` is relative to the directory containing `snippet.yml`. Multiple sysbuild `.conf` files can also be provided as a list.

`DTS_EXTRA_CPPFLAGS`

This `snippet.yml` adds `DTS_EXTRA_CPPFLAGS` CMake Cache variables to the build:

```
name: foo
append:
  DTS_EXTRA_CPPFLAGS: -DMY_DTS_CONFIGURE
```

Adding these flags enables control over the content of a devicetree file.

Board-specific settings

You can write settings that only apply to some boards.

The settings described here are applied in **addition** to snippet settings that apply to all boards. (This is similar, for example, to the way that an application with both `prj.conf` and `boards/foo.conf` files will use both `.conf` files in the build when building for board `foo`, instead of just `boards/foo.conf`)

By name

```
name: ...
boards:
  bar: # settings for board "bar" go here
    append:
      EXTRA_DTC_OVERLAY_FILE: bar.overlay
  baz: # settings for board "baz" go here
```

(continues on next page)

(continued from previous page)

```
    append:
      EXTRA_DTC_OVERLAY_FILE: baz.overlay
```

The above example uses `bar.overlay` when building for board `bar`, and `baz.overlay` when building for `baz`.

By regular expression You can enclose the board name in slashes (/) to match the name against a regular expression in the CMake syntax. The regular expression must match the entire board name.

For example:

```
name: foo
boards:
  /my_vendor_.*/:
    append:
      EXTRA_DTC_OVERLAY_FILE: my_vendor.overlay
```

The above example uses devicetree overlay `my_vendor.overlay` when building for either board `my_vendor_board1` or `my_vendor_board2`. It would not use the overlay when building for either `another_vendor_board` or `x_my_vendor_board`.

Board revisions Specific configuration for revisions of boards is also supported which will be applied after the common files:

```
name: foo
boards:
  bar:
    append:
      # Base file will be applied first
      EXTRA_DTC_OVERLAY_FILE: first.overlay
    revisions:
      "0.7.0":
        append:
          # Will be applied on top of common board file
          EXTRA_DTC_OVERLAY_FILE: extra_0_7_0.overlay
```

The above example will use `first.overlay` for all revisions of the `bar` board, and will also include `extra_0_7_0.overlay` when building for revision `0.7.0` of the `bar` board (`bar@0.7.0`).

5.5.4 Snippets Design

This page documents design goals for the snippets feature. Further information can be found in Issue #51834.

- **extensible**: for example, it is possible to add board support for an existing built-in snippet without modifying the zephyr repository
- **composable**: it is possible to use multiple snippets at once, for example using:

  ```
  west build -S <snippet1> -S <snippet2> ...
  ```

- **able to combine multiple types of configuration**: snippets make it possible to store multiple different types of build system settings in one place, and apply them all together
- **specializable**: for example, it is possible to customize a snippet's behavior for a particular board, or board revision

- **future-proof and backwards-compatible**: arbitrary future changes to the snippets feature will be possible without breaking backwards compatibility for older snippets
- **applicable to purely "software" changes**: unlike the shields feature, snippets do not assume the presence of a "daughterboard", "shield", "hat", or any other type of external assembly which is connected to the main board
- **DRY** (don't repeat yourself): snippets allow you to skip unnecessary repetition; for example, you can apply the same board-specific configuration to boards `foo` and `bar` by specifying `/(foo|bar)/` as a regular expression for the settings, which will then apply to both boards

5.6 Zephyr CMake Package

The Zephyr CMake package is a convenient way to create a Zephyr-based application.

> **Note**
>
> The *Application types* section introduces the application types used in this page.

The Zephyr CMake package ensures that CMake can automatically select a Zephyr installation to use for building the application, whether it is a *Zephyr repository application*, a *Zephyr workspace application*, or a *Zephyr freestanding application*.

When developing a Zephyr-based application, then a developer simply needs to write `find_package(Zephyr)` in the beginning of the application `CMakeLists.txt` file.

To use the Zephyr CMake package it must first be exported to the CMake user package registry. This means creating a reference to the current Zephyr installation inside the CMake user package registry.

Ubuntu

In Linux, the CMake user package registry is found in:

`~/.cmake/packages/Zephyr`

macOS

In macOS, the CMake user package registry is found in:

`~/.cmake/packages/Zephyr`

Windows

In Windows, the CMake user package registry is found in:

`HKEY_CURRENT_USER\Software\Kitware\CMake\Packages\Zephyr`

The Zephyr CMake package allows CMake to automatically find a Zephyr base. One or more Zephyr installations must be exported. Exporting multiple Zephyr installations may be useful when developing or testing Zephyr freestanding applications, Zephyr workspace application with vendor forks, etc..

5.6.1 Zephyr CMake package export (west)

When installing Zephyr using *west* then it is recommended to export Zephyr using `west zephyr-export`.

5.6.2 Zephyr CMake package export (without west)

Zephyr CMake package is exported to the CMake user package registry using the following commands:

```
cmake -P <PATH-TO-ZEPHYR>/share/zephyr-package/cmake/zephyr_export.cmake
```

This will export the current Zephyr to the CMake user package registry.

5.6.3 Zephyr Base Environment Setting

The Zephyr CMake package search functionality allows for explicitly specifying a Zephyr base using an environment variable.

To do this, use the following `find_package()` syntax:

```
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
```

This syntax instructs CMake to first search for Zephyr using the Zephyr base environment setting *ZEPHYR_BASE* and then use the normal search paths.

5.6.4 Zephyr CMake Package Search Order

When Zephyr base environment setting is not used for searching, the Zephyr installation matching the following criteria will be used:

- A Zephyr repository application will use the Zephyr in which it is located. For example:

  ```
  <projects>/zephyr-workspace/zephyr
  └── samples
      └── hello_world
  ```

  in this example, `hello_world` will use `<projects>/zephyr-workspace/zephyr`.

- Zephyr workspace application will use the Zephyr that share the same workspace. For example:

  ```
  <projects>/zephyr-workspace
  ├── zephyr
  ├── ...
  └── my_applications
       └── my_first_app
  ```

  in this example, `my_first_app` will use `<projects>/zephyr-workspace/zephyr` as this Zephyr is located in the same workspace as the Zephyr workspace application.

> **Note**
>
> The root of a Zephyr workspace is identical to `west topdir` if the workspace was installed using `west`

- Zephyr freestanding application will use the Zephyr registered in the CMake user package registry. For example:

  ```
  <projects>/zephyr-workspace-1
  └── zephyr                        (Not exported to CMake)

  <projects>/zephyr-workspace-2
  └── zephyr                        (Exported to CMake)
  ```

(continues on next page)

(continued from previous page)

```
<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

in this example, only `<projects>/zephyr-workspace-2/zephyr` is exported to the CMake package registry and therefore this Zephyr will be used by the Zephyr freestanding application `<home>/app`.

If user wants to test the application with `<projects>/zephyr-workspace-1/zephyr`, this can be done by using the Zephyr Base environment setting, meaning set `ZEPHYR_BASE=<projects>/zephyr-workspace-1/zephyr`, before running CMake.

> **Note**
>
> The Zephyr package selected on the first CMake invocation will be used for all subsequent builds. To change the Zephyr package, for example to test the application using Zephyr base environment setting, then it is necessary to do a pristine build first (See *Rebuilding an Application*).

5.6.5 Zephyr CMake Package Version

When writing an application then it is possible to specify a Zephyr version number `x.y.z` that must be used in order to build the application.

Specifying a version is especially useful for a Zephyr freestanding application as it ensures the application is built with a minimal Zephyr version.

It also helps CMake to select the correct Zephyr to use for building, when there are multiple Zephyr installations in the system.

For example:

```
find_package(Zephyr 2.2.0)
project(app)
```

will require app to be built with Zephyr 2.2.0 as minimum. CMake will search all exported candidates to find a Zephyr installation which matches this version criteria.

Thus it is possible to have multiple Zephyr installations and have CMake automatically select between them based on the version number provided, see CMake package version for details.

For example:

```
<projects>/zephyr-workspace-2.a
└── zephyr                        (Exported to CMake)

<projects>/zephyr-workspace-2.b
└── zephyr                        (Exported to CMake)

<home>/app
├── CMakeLists.txt
├── prj.conf
└── src
    └── main.c
```

in this case, there are two released versions of Zephyr installed at their own workspaces. Workspace 2.a and 2.b, corresponding to the Zephyr version.

To ensure app is built with minimum version `2.a` the following `find_package` syntax may be used:

```
find_package(Zephyr 2.a)
project(app)
```

Note that both `2.a` and `2.b` fulfill this requirement.

CMake also supports the keyword `EXACT`, to ensure an exact version is used, if that is required. In this case, the application CMakeLists.txt could be written as:

```
find_package(Zephyr 2.a EXACT)
project(app)
```

In case no Zephyr is found which satisfies the version required, as example, the application specifies

```
find_package(Zephyr 2.z)
project(app)
```

then an error similar to below will be printed:

```
Could not find a configuration file for package "Zephyr" that is compatible
with requested version "2.z".

The following configuration files were considered but not accepted:


↪<projects>/zephyr-workspace-2.a/zephyr/share/zephyr-package/cmake/ZephyrConfig.cmake,␣
↪version: 2.a.0

↪<projects>/zephyr-workspace-2.b/zephyr/share/zephyr-package/cmake/ZephyrConfig.cmake,␣
↪version: 2.b.0
```

Note

It can also be beneficial to specify a version number for Zephyr repository applications and Zephyr workspace applications. Specifying a version in those cases ensures the application will only build if the Zephyr repository or workspace is matching. This can be useful to avoid accidental builds when only part of a workspace has been updated.

5.6.6 Multiple Zephyr Installations (Zephyr workspace)

Testing out a new Zephyr version, while at the same time keeping the existing Zephyr in the workspace untouched is sometimes beneficial.

Or having both an upstream Zephyr, Vendor specific, and a custom Zephyr in same workspace.

For example:

```
<projects>/zephyr-workspace
├── zephyr
├── zephyr-vendor
├── zephyr-custom
├── ...
```

(continues on next page)

(continued from previous page)

```
└── my_applications
    └── my_first_app
```

in this setup, `find_package(Zephyr)` has the following order of precedence for selecting which Zephyr to use:

- Project name: `zephyr`
- First project, when Zephyr projects are ordered lexicographical, in this case.
  - `zephyr-custom`
  - `zephyr-vendor`

This means that `my_first_app` will use `<projects>/zephyr-workspace/zephyr`.

It is possible to specify a Zephyr preference list in the application.

A Zephyr preference list can be specified as:

```
set(ZEPHYR_PREFER "zephyr-custom" "zephyr-vendor")
find_package(Zephyr)

project(my_first_app)
```

the `ZEPHYR_PREFER` is a list, allowing for multiple Zephyrs. If a Zephyr is specified in the list, but not found in the system, it is simply ignored and `find_package(Zephyr)` will continue to the next candidate.

This allows for temporary creation of a new Zephyr release to be tested, without touching current Zephyr. When testing is done, the `zephyr-test` folder can simply be removed. Such a CMakeLists.txt could look as:

```
set(ZEPHYR_PREFER "zephyr-test")
find_package(Zephyr)

project(my_first_app)
```

5.6.7 Zephyr Build Configuration CMake packages

There are two Zephyr Build configuration packages which provide control over the build settings in Zephyr in a more generic way. These packages are:

- **ZephyrBuildConfiguration**: Applies to all Zephyr applications in the workspace
- **ZephyrAppConfiguration**: Applies only to the application you are currently building

They are similar to the per-user `.zephyrrc` file that can be used to set *Environment Variables*, but they set CMake variables instead. They also allow you to automatically share the build configuration among all users through the project repository. They also allow more advanced use cases, such as loading of additional CMake boilerplate code.

The Zephyr Build Configuration CMake packages will be loaded in the Zephyr boilerplate code after initial properties and `ZEPHYR_BASE` has been defined, but before CMake code execution. The ZephyrBuildConfiguration is included first and ZephyrAppConfiguration afterwards. That means the application-specific package could override the workspace settings, if needed. This allows the Zephyr Build Configuration CMake packages to setup or extend properties such as: `DTS_ROOT`, `BOARD_ROOT`, `TOOLCHAIN_ROOT` / other toolchain setup, fixed overlays, and any other property that can be controlled. It also allows inclusion of additional boilerplate code.

To provide a ZephyrBuildConfiguration or ZephyrAppConfiguration, create `ZephyrBuildConfig.cmake` and/or `ZephyrAppConfig.cmake` respectively and place them in the appropriate location.

The CMake `find_package` mechanism will search for these files with the steps below. Other default CMake package search paths and hints are disabled and there is no version checking implemented for these packages. This also means that these packages cannot be installed in the CMake package registry. The search steps are:

1. If `ZephyrBuildConfiguration_ROOT`, or `ZephyrAppConfiguration_ROOT` respectively, is set, search within this prefix path. If a matching file is found, execute this file. If no matching file is found, go to step 2.
2. Search within `${ZEPHYR_BASE}/../*`, or `${APPLICATION_SOURCE_DIR}` respectively. If a matching file is found, execute this file. If no matching file is found, abort the search.

It is recommended to place the files in the default paths from step 2, but with the `<PackageName>_ROOT` variables you have the flexibility to place them anywhere. This is especially necessary for freestanding applications, for which the default path to ZephyrBuildConfiguration usually does not work. In this case the `<PackageName>_ROOT` variables can be set on the CMake command line, **before** `find_package(Zephyr ...)`, as environment variable or from a CMake cache initialization file with the `-C` command line option.

> **Note**
>
> The `<PackageName>_ROOT` variables, as well as the default paths, are just the prefixes to the search path. These prefixes get combined with additional path suffixes, which together form the actual search path. Any combination that honors the CMake package search procedure is valid and will work.

If you want to completely disable the search for these packages, you can use the special CMake `CMAKE_DISABLE_FIND_PACKAGE_<PackageName>` variable for that. Just set `CMAKE_DISABLE_FIND_PACKAGE_ZephyrBuildConfiguration` or `CMAKE_DISABLE_FIND_PACKAGE_ZephyrAppConfiguration` to `TRUE` to disable the package.

An example folder structure could look like this:

```
<projects>/zephyr-workspace
├── zephyr
├── ...
├── manifest repo (can be named anything)
│    └── cmake/ZephyrBuildConfig.cmake
├── ...
└── zephyr application
     └── share/zephyrapp-package/cmake/ZephyrAppConfig.cmake
```

A sample `ZephyrBuildConfig.cmake` can be seen below.

```
# ZephyrBuildConfig.cmake sample code

# To ensure final path is absolute and does not contain ../.. in variable.
get_filename_component(APPLICATION_PROJECT_DIR
                       ${CMAKE_CURRENT_LIST_DIR}/../../..
                       ABSOLUTE
)

# Add this project to list of board roots
list(APPEND BOARD_ROOT ${APPLICATION_PROJECT_DIR})

# Default to GNU Arm Embedded toolchain if no toolchain is set
if(NOT ENV{ZEPHYR_TOOLCHAIN_VARIANT})
    set(ZEPHYR_TOOLCHAIN_VARIANT gnuarmemb)
    find_program(GNU_ARM_GCC arm-none-eabi-gcc)
```

(continues on next page)

(continued from previous page)

```
    if(NOT ${GNU_ARM_GCC} STREQUAL GNU_ARM_GCC-NOTFOUND)
        # The toolchain root is located above the path to the compiler.
        get_filename_component(GNUARMEMB_TOOLCHAIN_PATH ${GNU_ARM_GCC}/../..␣
↪ABSOLUTE)
    endif()
endif()
```

5.6.8 Zephyr CMake package source code

The Zephyr CMake package source code in share/zephyr-package/cmake contains the CMake config package which is used by the CMake `find_package` function.

It also contains code for exporting Zephyr as a CMake config package.

The following is an overview of the files in this directory:

`ZephyrConfigVersion.cmake`
: The Zephyr package version file. This file is called by CMake to determine if this installation fulfils the requirements specified by user when calling `find_package(Zephyr ...)`. It is also responsible for detection of Zephyr repository or workspace only installations.

`ZephyrConfig.cmake`
: The Zephyr package file. This file is called by CMake to for the package meeting which fulfils the requirements specified by user when calling `find_package(Zephyr ...)`. This file is responsible for sourcing of boilerplate code.

`zephyr_package_search.cmake`
: Common file used for detection of Zephyr repository and workspace candidates. Used by `ZephyrConfigVersion.cmake` and `ZephyrConfig.cmake` for common code.

`zephyr_export.cmake`
: See *Zephyr CMake package export (without west)*.

5.7 Sysbuild (System build)

Sysbuild is a higher-level build system that can be used to combine multiple other build systems together. It is a higher-level layer that combines one or more Zephyr build systems and optional additional build systems into a hierarchical build system.

For example, you can use sysbuild to build a Zephyr application together with the MCUboot bootloader, flash them both onto your device, and debug the results.

Sysbuild works by configuring and building at least a Zephyr application and, optionally, as many additional projects as you want. The additional projects can be either Zephyr applications or other types of builds you want to run.

Like Zephyr's *build system*, sysbuild is written in CMake and uses *Kconfig*.

5.7.1 Definitions

The following are some key concepts used in this document:

Single-image build
: When sysbuild is used to create and manage just one Zephyr application's build system.

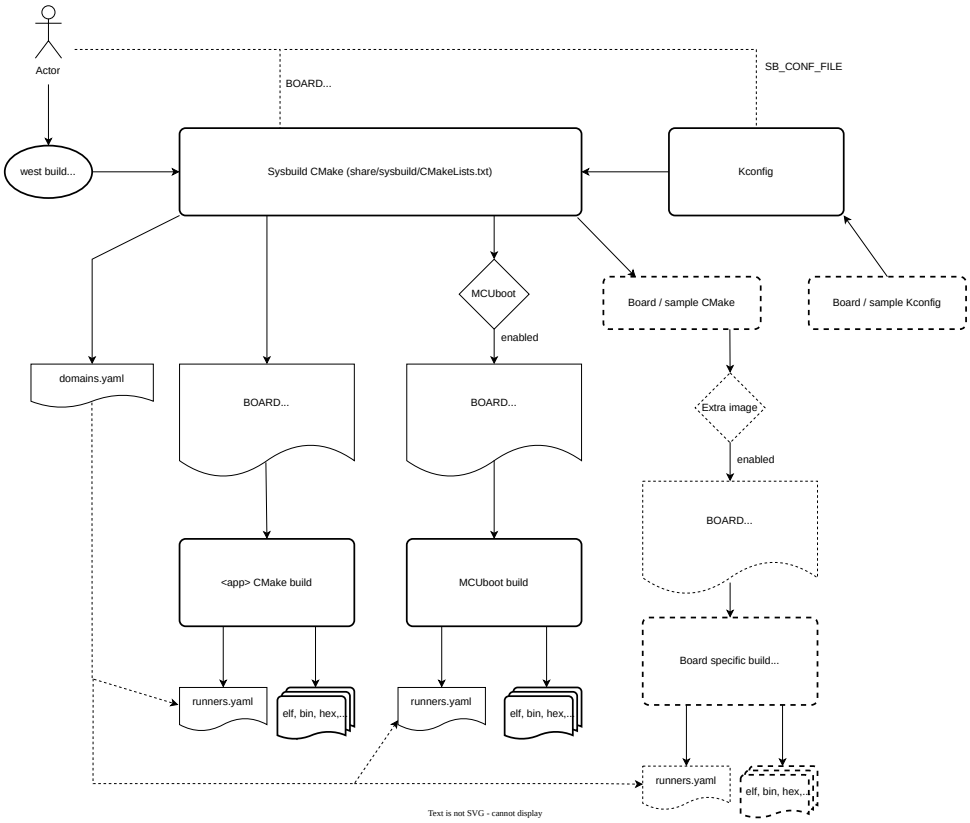
Multi-image build
: When sysbuild is used to manage multiple build systems. The word "image" is used because your main goal is usually to generate the binaries of the firmware application images from each build system.

Domain
Every Zephyr CMake build system managed by sysbuild.

Multi-domain
When more than one Zephyr CMake build system (domain) is managed by sysbuild.

5.7.2 Architectural Overview

This figure is an overview of sysbuild's inputs, outputs, and user interfaces:



The following are some key sysbuild features indicated in this figure:

- You can run sysbuild either with *west build* or directly via `cmake`.
- You can use sysbuild to generate application images from each build system, shown above as ELF, BIN, and HEX files.
- You can configure sysbuild or any of the build systems it manages using various configuration variables. These variables are namespaced so that sysbuild can direct them to the right build system. In some cases, such as the `BOARD` variable, these are shared among multiple build systems.
- Sysbuild itself is also configured using Kconfig. For example, you can instruct sysbuild to build the MCUboot bootloader, as well as to build and link your main Zephyr application as an MCUboot-bootable image, using sysbuild's Kconfig files.
- Sysbuild integrates with west's *Building, Flashing and Debugging* commands. It does this by managing the *Configuration Options*, and specifically the `runners.yaml` files that each Zephyr build system will contain. These are packaged into a global view of how to flash and debug each build system in a `domains.yaml` file generated and managed by sysbuild.
- Build names are prefixed with the target name and an underscore, for example the sysbuild target is prefixed with `sysbuild_` and if MCUboot is enabled as part of sysbuild, it will be prefixed with `mcuboot_`. This also allows for running things like menuconfig with the prefix,

for example (if using ninja) `ninja sysbuild_menuconfig` to configure sysbuild or (if using make) `make mcuboot_menuconfig`.

5.7.3 Building with sysbuild

As mentioned above, you can run sysbuild via `west build` or `cmake`.

`west build`

Here is an example. For details, see *Sysbuild (multi-domain builds)* in the `west build` documentation.

```
west build -b reel_board --sysbuild samples/hello_world
```

Tip

To configure `west build` to use `--sysbuild` by default from now on, run:

```
west config build.sysbuild True
```

Since sysbuild supports both single- and multi-image builds, this lets you use sysbuild all the time, without worrying about what type of build you are running.

To turn this off, run this before generating your build system:

```
west config build.sysbuild False
```

To turn this off for just one `west build` command, run:

```
west build --no-sysbuild ...
```

`cmake`

Here is an example using CMake and Ninja.

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world␣
↪share/sysbuild
ninja -Cbuild
```

To use sysbuild directly with CMake, you must specify the sysbuild project as the source folder, and give `-DAPP_DIR=<path-to-sample>` as an extra CMake argument or set APP_DIR as environment variable. `APP_DIR` is the path to the main Zephyr application managed by sysbuild.

Tip

The environment variables `CMAKE_BUILD_PARALLEL_LEVEL` and `VERBOSE` can be used to control the build process when using sysbuild with CMake and ninja.

To set number of jobs for ninja for all sysbuild images, set the `CMAKE_BUILD_PARALLEL_LEVEL` environment variable and invoke the build with `cmake --build`, for example:

```
CMAKE_BUILD_PARALLEL_LEVEL=<n> cmake --build .
```

For verbose output of all images, use:

```
VERBOSE=1 cmake --build .
```

5.7.4 Configuration namespacing

When building a single Zephyr application without sysbuild, all CMake cache settings and Kconfig build options given on the command line as `-D<var>=<value>` or `-DCONFIG_<var>=<value>`

are handled by the Zephyr build system.

However, when sysbuild combines multiple Zephyr build systems, there could be Kconfig settings exclusive to sysbuild (and not used by any of the applications). To handle this, sysbuild has namespaces for configuration variables. You can use these namespaces to direct settings either to sysbuild itself or to a specific Zephyr application managed by sysbuild using the information in these sections.

The following example shows how to build hello_world with MCUboot enabled, applying to both images debug optimizations:

west build

```
west build -b reel_board --sysbuild samples/hello_world -- -
↪DSB_CONFIG_BOOTLOADER_MCUBOOT=y -DCONFIG_DEBUG_OPTIMIZATIONS=y -
↪Dmcuboot_CONFIG_DEBUG_OPTIMIZATIONS=y
```

cmake

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world -
↪DSB_CONFIG_BOOTLOADER_MCUBOOT=y -DCONFIG_DEBUG_OPTIMIZATIONS=y -
↪Dmcuboot_CONFIG_DEBUG_OPTIMIZATIONS=y share/sysbuild
ninja -Cbuild
```

See the following subsections for more information.

CMake variable namespacing

CMake variable settings can be passed to CMake using `-D<var>=<value>` on the command line. You can also set Kconfig options via CMake as `-DCONFIG_<var>=<value>` or `-D<namespace>_CONFIG_<var>=<value>`.

Since sysbuild is the entry point for the build system, and sysbuild is written in CMake, all CMake variables are first processed by sysbuild.

Sysbuild creates a namespace for each domain. The namespace prefix is the domain's application name. See *Adding Zephyr applications to sysbuild* for more information.

To set the variable `<var>` in the namespace `<namespace>`, use this syntax:

```
-D<namespace>_<var>=<value>
```

For example, to set the CMake variable `FOO` in the `my_sample` application build system to the value `BAR`, run the following commands:

west build

```
west build --sysbuild ... -- -Dmy_sample_FOO=BAR
```

cmake

```
cmake -Dmy_sample_FOO=BAR ...
```

Kconfig namespacing

To set the sysbuild Kconfig option `<var>` to the value `<value>`, use this syntax:

```
-DSB_CONFIG_<var>=<value>
```

In the previous example, `SB_CONFIG` is the namespace prefix for sysbuild's Kconfig options.

To set a Zephyr application's Kconfig option instead, use this syntax:

```
-D<namespace>_CONFIG_<var>=<value>
```

In the previous example, `<namespace>` is the application name discussed above in *CMake variable namespacing*.

For example, to set the Kconfig option `FOO` in the `my_sample` application build system to the value `BAR`, run the following commands:

`west build`

```
west build --sysbuild ... -- -Dmy_sample_CONFIG_FOO=BAR
```

`cmake`

```
cmake -Dmy_sample_CONFIG_FOO=BAR ...
```

Tip

When no `<namespace>` is used, the Kconfig setting is passed to the main Zephyr application `my_sample`.

This means that passing `-DCONFIG_<var>=<value>` and `-Dmy_sample_CONFIG_<var>=<value>` are equivalent.

This allows you to build the same application with or without sysbuild using the same syntax for setting Kconfig values at CMake time. For example, the following commands will work in the same way:

```
west build -b <board> my_sample -- -DCONFIG_FOO=BAR
```

```
west build -b <board> --sysbuild my_sample -- -DCONFIG_FOO=BAR
```

5.7.5 Sysbuild flashing using `west flash`

You can use *west flash* to flash applications with sysbuild.

When invoking `west flash` on a build consisting of multiple images, each image is flashed in sequence. Extra arguments such as `--runner jlink` are passed to each invocation.

For more details, see *Multi-domain flashing*.

5.7.6 Sysbuild debugging using `west debug`

You can use `west debug` to debug the main application, whether you are using sysbuild or not. Just follow the existing *west debug* guide to debug the main sample.

To debug a different domain (Zephyr application), such as `mcuboot`, use the `--domain` argument, as follows:

```
west debug --domain mcuboot
```

For more details, see *Multi-domain debugging*.

5.7.7 Building a sample with MCUboot

Sysbuild supports MCUboot natively.

To build a sample like `hello_world` with MCUboot, enable MCUboot and build and flash the sample as follows:

`west build`

```
west build -b reel_board --sysbuild samples/hello_world -- -
↪DSB_CONFIG_BOOTLOADER_MCUBOOT=y
```

cmake

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world -
↪DSB_CONFIG_BOOTLOADER_MCUBOOT=y share/sysbuild
ninja -Cbuild
```

This builds `hello_world` and `mcuboot` for the `reel_board`, and then flashes both the `mcuboot` and `hello_world` application images to the board.

More detailed information regarding the use of MCUboot with Zephyr can be found in the MCUboot with Zephyr documentation page on the MCUboot website.

> **Note**
>
> The deprecated MCUBoot Kconfig option `CONFIG_ZEPHYR_TRY_MASS_ERASE` will perform a full chip erase when flashed. If this option is enabled, then flashing only MCUBoot, for example using `west flash --domain mcuboot`, may erase the entire flash, including the main application image.

5.7.8 Sysbuild Kconfig file

You can set sysbuild's Kconfig options for a single application using configuration files. By default, sysbuild looks for a configuration file named `sysbuild.conf` in the application top-level directory.

In the following example, there is a `sysbuild.conf` file that enables building and flashing with MCUboot whenever sysbuild is used:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
└── sysbuild.conf
```

```
SB_CONFIG_BOOTLOADER_MCUBOOT=y
```

You can set a configuration file to use with the `-DSB_CONF_FILE=<sysbuild-conf-file>` CMake build setting.

For example, you can create `sysbuild-mcuboot.conf` and then specify this file when building with sysbuild, as follows:

west build

```
west build -b reel_board --sysbuild samples/hello_world -- -
↪DSB_CONF_FILE=sysbuild-mcuboot.conf
```

cmake

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=samples/hello_world -
↪DSB_CONF_FILE=sysbuild-mcuboot.conf share/sysbuild
ninja -Cbuild
```

5.7.9 Sysbuild targets

Sysbuild creates build targets for each image (including sysbuild itself) for the following modes:

- menuconfig
- hardenconfig
- guiconfig

For the main application (as is the same without using sysbuild) these can be ran normally without any prefix. For other images (including sysbuild), these are ran with a prefix of the image name and an underscore e.g. `sysbuild_` or `mcuboot_`, using ninja or make - for details on how to run image build targets that do not have mapped build targets in sysbuild, see the *Dedicated image build targets* section.

5.7.10 Dedicated image build targets

Not all build targets for images are given equivalent prefixed build targets when sysbuild is used, for example build targets like `ram_report`, `rom_report`, `footprint`, `puncover` and `pahole` are not exposed. When using *Trusted Firmware*, the build targets prefixed with `tfm_` and `bl2_` are also not exposed, for example: `tfm_rom_report` and `bl2_ram_report`. To run these build targets, the build directory of the image can be provided to west/ninja/make along with the name of the build target to execute and it will run.

`west`

Assuming that a project has been configured and built using `west` using sysbuild with mcuboot enabled in the default `build` folder location, the `rom_report` build target for `mcuboot` can be ran with:

```
west build -d build/mcuboot -t rom_report
```

For TF-M projects using TF-M targets, the application build directory is used like so:

```
west build -d build/<app_name> -t tfm_rom_report
```

`ninja`

Assuming that a project has been configured using `cmake` and built using `ninja` using sysbuild with mcuboot enabled, the `rom_report` build target for `mcuboot` can be ran with:

```
ninja -C mcuboot rom_report
```

For TF-M projects using TF-M targets, the application build directory is used like so:

```
ninja -C <app_name> -t tfm_rom_report
```

`make`

Assuming that a project has been configured using `cmake` and built using `make` using sysbuild with mcuboot enabled, the `rom_report` build target for `mcuboot` can be ran with:

```
make -C mcuboot rom_report
```

For TF-M projects using TF-M targets, the application build directory is used like so:

```
make -C <app_name> -t tfm_rom_report
```

5.7.11 Adding Zephyr applications to sysbuild

You can use the `ExternalZephyrProject_Add()` function to add Zephyr applications as sysbuild domains. Call this CMake function from your application's `sysbuild.cmake` file, or any other CMake file you know will run as part sysbuild CMake invocation.

A variant image can also added using the *ExternalZephyrVariantProject_Add()* function which will duplicate an existing image in the sysbuild project, and allows for slight differences in configuration. An example use case for this feature is to change the chosen flash node of an image but having the rest of the configuration identical to the base image. When this is used, neither sysbuild itself nor the image will have the extra Kconfig targets made for it such as menuconfig, guiconfig, hardenconfig or traceconfig, as the base image can be used for viewing/adjusting these instead.

Targeting the same board

To include `my_sample` as another sysbuild domain, targeting the same board as the main image, use this example:

```
ExternalZephyrProject_Add(
  APPLICATION my_sample
  SOURCE_DIR <path-to>/my_sample
)
```

This could be useful, for example, if your board requires you to build and flash an SoC-specific bootloader along with your main application.

Targeting a different board

In sysbuild and Zephyr CMake build system a board may refer to:

- A physical board with a single core SoC.
- A specific core on a physical board with a multi-core SoC, such as nrf5340dk.
- A specific SoC on a physical board with multiple SoCs, such as nrf9160dk_nrf9160 and nrf9160dk_nrf52840.

If your main application, for example, is built for `mps2/an521/cpu0`, and your helper application must target the `mps2/an521/cpu1` board target, add a CMake function call that is structured as follows:

```
ExternalZephyrProject_Add(
  APPLICATION my_sample
  SOURCE_DIR <path-to>/my_sample
  BOARD mps2/an521/cpu1
)
```

This could be useful, for example, if your main application requires another helper Zephyr application to be built and flashed alongside it, but the helper runs on another core in your SoC.

Targeting conditionally using Kconfig

You can control whether extra applications are included as sysbuild domains using Kconfig.

If the extra application image is specific to the board or an application, you can create two additional files: `sysbuild.cmake` and `Kconfig.sysbuild`.

For an application, this would look like this:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
├── Kconfig.sysbuild
└── sysbuild.cmake
```

In the previous example, `sysbuild.cmake` would be structured as follows:

```
if(SB_CONFIG_SECOND_SAMPLE)
  ExternalZephyrProject_Add(
    APPLICATION second_sample
    SOURCE_DIR <path-to>/second_sample
  )
endif()
```

`Kconfig.sysbuild` would be structured as follows:

```
source "sysbuild/Kconfig"

config SECOND_SAMPLE
        bool "Second sample"
        default y
```

This will include `second_sample` by default, while still allowing you to disable it using the Kconfig option `SECOND_SAMPLE`.

For more information on setting sysbuild Kconfig options, see *Kconfig namespacing*.

Building without flashing

You can mark `my_sample` as a build-only application in this manner:

```
ExternalZephyrProject_Add(
  APPLICATION my_sample
  SOURCE_DIR <path-to>/my_sample
  BUILD_ONLY TRUE
)
```

As a result, `my_sample` will be built as part of the sysbuild build invocation, but it will be excluded from the default image sequence used by `west flash`. Instead, you may use the outputs of this domain for other purposes - for example, to produce a secondary image for DFU, or to merge multiple images together.

You can also replace `TRUE` with another boolean constant in CMake, such as a Kconfig option, which would make `my_sample` conditionally build-only.

> **Note**
>
> Applications marked as build-only can still be flashed manually, using `west flash --domain my_sample`. As such, the `BUILD_ONLY` option only controls the default behavior of `west flash`.

Zephyr application configuration

When adding a Zephyr application to sysbuild, such as MCUboot, then the configuration files from the application (MCUboot) itself will be used.

When integrating multiple applications with each other, then it is often necessary to make adjustments to the configuration of extra images.

Sysbuild gives users the ability of creating Kconfig fragments or devicetree overlays that will be used together with the application's default configuration. Sysbuild also allows users to change *Application Configuration Directory* in order to give users full control of an image's configuration.

Zephyr application Kconfig fragment and devicetree overlay In the folder of the main application, create a Kconfig fragment or a devicetree overlay under a sysbuild folder, where the

name of the file is `<image>.conf` or `<image>.overlay`, for example if your main application includes `my_sample` then create a `sysbuild/my_sample.conf` file or a devicetree overlay `sysbuild/my_sample.overlay`.

A Kconfig fragment could look as:

```
# sysbuild/my_sample.conf
CONFIG_FOO=n
```

Zephyr application configuration directory In the folder of the main application, create a new folder under `sysbuild/<image>/`. This folder will then be used as `APPLICATION_CONFIG_DIR` when building `<image>`. As an example, if your main application includes `my_sample` then create a `sysbuild/my_sample/` folder and place any configuration files in there as you would normally do:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
└── sysbuild
    └── my_sample
        ├── prj.conf
        ├── app.overlay
        └── boards
            ├── <board_A>.conf
            ├── <board_A>.overlay
            ├── <board_B>.conf
            └── <board_B>.overlay
```

All configuration files under the `sysbuild/my_sample/` folder will now be used when `my_sample` is included in the build, and the default configuration files for `my_sample` will be ignored.

This give you full control on how images are configured when integrating those with `application`.

Sysbuild file suffix support File suffix support through the `FILE_SUFFIX` is supported in sysbuild (see *File Suffixes* for details on this feature in applications). For sysbuild, a globally provided option will be passed down to all images. In addition, the image configuration file will have this value applied and used (instead of the build type) if the file exists.

Given the example project:

```
<home>/application
├── CMakeLists.txt
├── prj.conf
├── sysbuild.conf
├── sysbuild_test_key.conf
└── sysbuild
    ├── mcuboot.conf
    ├── mcuboot_max_log.conf
    └── my_sample.conf
```

- If `FILE_SUFFIX` is not defined and both `mcuboot` and `my_sample` images are included, `mcuboot` will use the `mcuboot.conf` Kconfig fragment file and `my_sample` will use the `my_sample.conf` Kconfig fragment file. Sysbuild itself will use the `sysbuild.conf` Kconfig fragment file.
- If `FILE_SUFFIX` is set to `max_log` and both `mcuboot` and `my_sample` images are included, `mcuboot` will use the `mcuboot_max_log.conf` Kconfig fragment file and `my_sample` will use the `my_sample.conf` Kconfig fragment file (as it will fallback to the file without the suffix). Sysbuild itself will use the `sysbuild.conf` Kconfig fragment file (as it will fallback to the file without the suffix).

- If FILE_SUFFIX is set to test_key and both mcuboot and my_sample images are included, mcuboot will use the mcuboot.conf Kconfig fragment file and my_sample will use the my_sample.conf Kconfig fragment file (as it will fallback to the files without the suffix). Sysbuild itself will use the sysbuild_test_key.conf Kconfig fragment file. This can be used to apply a different sysbuild configuration, for example to use a different signing key in MCUboot and when signing the main application.

The FILE_SUFFIX can also be applied only to single images by prefixing the variable with the image name:

west build

```
west build -b reel_board --sysbuild file_suffix_example -- -
↪DSB_CONFIG_BOOTLOADER_MCUBOOT=y -Dmcuboot_FILE_SUFFIX="max_log"
```

cmake

```
cmake -Bbuild -GNinja -DBOARD=reel_board -DAPP_DIR=<app_dir> -
↪DSB_CONFIG_BOOTLOADER_MCUBOOT=y -Dmcuboot_FILE_SUFFIX="max_log" share/sysbuild
ninja -Cbuild
```

Adding dependencies among Zephyr applications

Sometimes, in a multi-image build, you may want certain Zephyr applications to be configured or flashed in a specific order. For example, if you need some information from one application's build system to be available to another's, then the first thing to do is to add a configuration dependency between them. Separately, you can also add flashing dependencies to control the sequence of images used by west flash; this could be used if a specific flashing order is required by an SoC, a _runner_, or something else.

By default, sysbuild will configure and flash applications in the order that they are added, as *ExternalZephyrProject_Add()* calls are processed by CMake. You can use the *sysbuild_add_dependencies()* function to make adjustments to this order, according to your needs. Its usage is similar to the standard add_dependencies() function in CMake.

Here is an example of adding configuration dependencies for my_sample:

```
sysbuild_add_dependencies(IMAGE CONFIGURE my_sample sample_a sample_b)
```

This will ensure that sysbuild will run CMake for sample_a and sample_b (in some order) before doing the same for my_sample, when building these domains in a single invocation.

If you want to add flashing dependencies instead, then do it like this:

```
sysbuild_add_dependencies(IMAGE FLASH my_sample sample_a sample_b)
```

As a result, my_sample will be flashed after sample_a and sample_b (in some order), when flashing these domains in a single invocation.

> **Note**
>
> Adding flashing dependencies is not allowed for build-only applications. If my_sample had been created with BUILD_ONLY TRUE, then the above call to sysbuild_add_dependencies() would have produced an error.

5.7.12 Merged hex files

Sysbuild supports creating merged hex files, which will be created one per unique board target in a sysbuild project and can be enabled with SB_CONFIG_MERGED_HEX_FILES. The output filename format will be merged_<NORMALIZED_BOARD_TARGET>.hex. This is ideal for creating production

images for deployment, it requires that all images output hex files. If a sysbuild project builds multiple images for the same board target but for **different** devices (e.g. 3 images for 3 different boards as part of a test), then this option should remain disabled. Hex files will be merged as per the flashing order they have been configured in, any overlapping data from previous addresses that conflicts with later merged hex files will be overwritten with the later hex file data - see *Adding dependencies among Zephyr applications* for how to configure the flashing order of sysbuild images.

5.7.13 Adding non-Zephyr applications to sysbuild

You can include non-Zephyr applications in a multi-image build using the standard CMake module `ExternalProject`. Please refer to the CMake documentation for usage details.

When using `ExternalProject`, the non-Zephyr application will be built as part of the sysbuild build invocation, but `west flash` or `west debug` will not be aware of the application. Instead, you must manually flash and debug the application.

5.7.14 Configuring sysbuild internal state

Because sysbuild is a CMake project in it's own right, it runs and sets up itself similar to a Zephyr application but using it's own CMake code. This means that some features, for example specifying variables in an application `CMakeLists.txt` (such as `BOARD_ROOT`) will not work, instead these must be set by using a *a module* or by using a custom sysbuild project file as `<application>/sysbuild/CMakeLists.txt`, for example:

```
# Place pre-sysbuild configuration items here

# For changing configuration that sysbuild itself uses:
# set(<var> <value>)
# list(APPEND <list> <value>)

# For changing configuration of other images:
# set(<image>_<var> <value> CACHE INTERNAL "<description>")

# Finds the sysbuild project and includes it with the new configuration
find_package(Sysbuild REQUIRED HINTS $ENV{ZEPHYR_BASE})

project(sysbuild LANGUAGES)
```

An example of adding a `BOARD_ROOT`:

```
list(APPEND BOARD_ROOT ${CMAKE_CURRENT_SOURCE_DIR}/../<extra-board-root>)

find_package(Sysbuild REQUIRED HINTS $ENV{ZEPHYR_BASE})

project(sysbuild LANGUAGES)
```

This will pass the board root on to all images as part of a project, it does not need to be repeated in each image's `CMakeLists.txt` file.

5.7.15 Extending sysbuild

Sysbuild can be extended by other modules to give it additional functionality or include other configuration or images, an example could be to add support for another bootloader or external signing method.

Modules can be extended by adding custom CMake or Kconfig files as normal *modules* do, this will cause the files to be included in each image that is part of a project. Alternatively, there are *sysbuild-specific module extension* files which can be used to include CMake and Kconfig files for

the overall sysbuild image itself, this is where e.g. a custom image for a particular board or SoC can be added.

Sysbuild images

Sysbuild can be used to add additional images to builds, these can be added by projects or boards though at present must be Zephyr applications.

Methods of adding images Images can be added to project or projects in various ways, multiple ways can be used simultaneously, they can be added in the following ways:

Applications Applications can add sysbuild images using the `sysbuild.cmake` file in the application directory, the inclusion of images can be controlled with a `Kconfig.sysbuild` file in the application directory.

Boards Boards can add sysbuild images by using the `sysbuild.cmake` file in the board directory, the inclusion of images can be controlled with a `Kconfig.sysbuild` file in the board directory.

SoCs SoCs can add sysbuild images by using the `sysbuild.cmake` file in the soc directory.

Modules *Modules (External projects)* can add sysbuild images with the `sysbuild-cmake` and `sysbuild-kconfig` options in a `module.yml` file, see *Sysbuild integration* for details.

Adding images Images can be added in one of two ways:

Single unchangeable image With this setup, the image to be added is fixed to a specific application and cannot be changed (although it cannot be changed to another image, the version of the image itself can be changed by use of a west manifest bringing in a different version of the application repo or from an alternate source, assuming that the image is in it's own repository).

> **Note**
>
> This method should only be used if the image is locked to a specific interface and has no extensibility.

An example of how to create such an image, this assumes that a *Zephyr application has been created*:

`Kconfig.sysbuild`

```
config MY_IMAGE
        bool "Include my amazing image"
        help
          If enabled, will include my amazing image in the build which does...
```

> **Note**
>
> Remember to have `source "share/sysbuild/Kconfig"` in the file if this is being applied in an application `Kconfig.sysbuild` file.

`sysbuild.cmake`

```
if(SB_CONFIG_MY_IMAGE)
  ExternalZephyrProject_Add(
    APPLICATION my_image
    SOURCE_DIR ${ZEPHYR_MY_IMAGE_MODULE_DIR}/path/to/my_image
  )
endif()
```

Additional dependency ordering can be set here if needed, see *Adding dependencies among Zephyr applications* for details, image configuration can also be set here, see *Image configuration* for details.

This image can be enabled when building with west like so:

```
west build -b nrf52840dk/nrf52840 --sysbuild <app> -- -DSB_CONFIG_MY_IMAGE=y
```

Extensible changeable image With this setup, the image to be added can be one of any number of possible applications which the user has selection over, this list of selections can also be extended downstream to add additional options for out-of-tree specific applications that are not available in upstream Zephyr. This is more complex to create but is the preferred method for adding images to upstream Zephyr.

`Kconfig.sysbuild`

```
config SUPPORT_OTHER_APP
        bool
        # Conditions can be placed here if this application type is only usable on␣
↪certain platforms
        default y

config SUPPORT_OTHER_APP_MY_IMAGE
        bool
        # Conditions can be placed here if this image is only usable on certain␣
↪platforms
        default y

choice OTHER_APP
        prompt "Other app image"
        # Defaults can be specified here if a default image should be loaded if␣
↪e.g. it is supported
        default OTHER_APP_NONE
        depends on SUPPORT_OTHER_APP

config OTHER_APP_IMAGE_NONE
        bool "None"
        help
          Do not Include an other app image in the build.

config OTHER_APP_IMAGE_MY_IMAGE
        bool "my_image"
        depends on SUPPORT_OTHER_APP_MY_IMAGE
        help
          Include my amazing image as the other app image to use, which does...

endchoice

config OTHER_APP_IMAGE_NAME
        string
```

(continues on next page)

(continued from previous page)

```
        default "my_image" if OTHER_APP_IMAGE_MY_IMAGE
        help
          Name of other app image.

config OTHER_APP_IMAGE_PATH
        string
        default "$(ZEPHYR_MY_IMAGE_MODULE_DIR)/path/to/my_image" if␣
↪OTHER_APP_IMAGE_MY_IMAGE
        help
          Source directory of other app image.
```

Note

Remember to have `source "$(ZEPHYR_BASE)/share/sysbuild/Kconfig"` in the file if this is being applied in an application `Kconfig.sysbuild` file.

`sysbuild.cmake`

```
if(SB_CONFIG_OTHER_APP_IMAGE_PATH)
  ExternalZephyrProject_Add(
    APPLICATION ${SB_CONFIG_OTHER_APP_IMAGE_NAME}
    SOURCE_DIR ${SB_CONFIG_OTHER_APP_IMAGE_PATH}
  )
endif()
```

Additional dependency ordering can be set here if needed, see *Adding dependencies among Zephyr applications* for details, image configuration can also be set here, see *Image configuration* for details.

This secondary image can be enabled when building with west like so:

```
west build -b nrf52840dk/nrf52840 --sysbuild <app> -- -DSB_CONFIG_MY_IMAGE=y
```

This can then be extended by *Modules (External projects)* like so:

`Kconfig.sysbuild`

```
config SUPPORT_OTHER_APP_MY_SECOND_IMAGE
        bool
        default y

choice OTHER_APP

config OTHER_APP_IMAGE_MY_SECOND_IMAGE
        bool "my_second_image"
        depends on SUPPORT_OTHER_APP_MY_SECOND_IMAGE
        help
          Include my other amazing image as the other app image to use, which␣
↪does...

endchoice

config OTHER_APP_IMAGE_NAME
        default "my_second_image" if OTHER_APP_IMAGE_MY_SECOND_IMAGE

config OTHER_APP_IMAGE_PATH
```

(continues on next page)

(continued from previous page)

```
        default "$(ZEPHYR_MY_SECOND_IMAGE_MODULE_DIR)/path/to/my_second_image" if␣
↪OTHER_APP_IMAGE_MY_SECOND_IMAGE
```

As can be seen, no additional CMake changes are needed to add an alternative image as the base CMake code will add the replacement image instead of the original image, if selected.

This alternative secondary image can be enabled when building with west like so:

```
west build -b nrf52840dk/nrf52840 --sysbuild <app> -- -DSB_CONFIG_MY_SECOND_IMAGE=y
```

Image configuration Sysbuild supports being able to set image configuration (Kconfig options) and also supports reading the output of image configuration (Kconfig) which can be used to allow an option to be added to sysbuild itself and then to configure it globally or selectively.

Setting image configuration

Kconfig Sysbuild can be used to set image configuration **before the CMake configuration of images takes place**. An important note about setting Kconfig options in images is that these are persistent and cannot be changed by the images. The following functions can be used to set configuration on an image:

```
set_config_bool(<image> CONFIG_<setting> <value>)
set_config_string(<image> CONFIG_<setting> <value>)
set_config_int(<image> CONFIG_<setting> <value>)
```

For example, to change the default image to output a hex file:

```
set_config_bool(${DEFAULT_IMAGE} CONFIG_BUILD_OUTPUT_HEX y)
```

These can safely be used in an application, board or SoC `sysbuild.cmake` file, as that file is included before the image CMake processes are invoked. When extending *sysbuild using modules* the pre-CMake hook should be used instead, for example:

```
function(${SYSBUILD_CURRENT_MODULE_NAME}_pre_cmake)
  cmake_parse_arguments(PRE_CMAKE "" "" "IMAGES" ${ARGN})

  foreach(image ${PRE_CMAKE_IMAGES})
    set_config_bool(${image} CONFIG_BUILD_OUTPUT_HEX y)
  endforeach()
endfunction()
```

Image configuration script An image configuration script is a CMake file that can be used to configure an image with common configuration values, multiple can be used per image, the configuration should be transferable to different images to correctly configure them based upon options set in sysbuild. MCUboot configuration options are configured in both the application the MCUboot image using this method which allows sysbuild to be the central location for things like the signing key which are then kept in-sync in the main application bootloader images. See *Signing key files* for setting the key to an absolute path or a CMake variable such as `${APP_DIR}`.

Inside image configuration scripts, the `ZCMAKE_APPLICATION` variable is set to the name of the application being configured, the `set_config_*` sysbuild CMake functions can be used to set configuration and the sysbuild Kconfig can be read, for example:

```
if(SB_CONFIG_BOOTLOADER_MCUBOOT AND "${SB_CONFIG_SIGNATURE_TYPE}" STREQUAL "NONE")
  set_config_bool(${ZCMAKE_APPLICATION} CONFIG_MCUBOOT_GENERATE_UNSIGNED_IMAGE y)
endif()
```

Image configuration script (module/application) Module/application image configuration scripts can be set from module or application code, this must be done in a `sysbuild.cmake` file for an application. This can be used to add an image configuration script as follows:

`sysbuild.cmake`

```
# This applies the image configuration script to the default image only
get_property(tmp_conf_scripts TARGET ${DEFAULT_IMAGE} PROPERTY IMAGE_CONF_SCRIPT)
list(APPEND tmp_conf_scripts "$
↪{CMAKE_SOURCE_DIR}/image_configurations/MY_CUSTOM_TYPE_image_default.cmake")
set_target_properties(${DEFAULT_IMAGE} PROPERTIES IMAGE_CONF_SCRIPT "$
↪{tmp_conf_scripts}")
```

Module CMake

```
function(${SYSBUILD_CURRENT_MODULE_NAME}_pre_cmake)
  cmake_parse_arguments(PRE_CMAKE "" "" "IMAGES" ${ARGN})

  # This applies the image configuration script to all images
  foreach(image ${PRE_CMAKE_IMAGES})
    get_property(tmp_conf_scripts TARGET ${image} PROPERTY IMAGE_CONF_SCRIPT)
    list(APPEND tmp_conf_scripts "$
↪{CMAKE_SOURCE_DIR}/image_configurations/MY_CUSTOM_TYPE_image_default.cmake")
    set_target_properties(${image} PROPERTIES IMAGE_CONF_SCRIPT "$
↪{tmp_conf_scripts}")
  endforeach()
endfunction(${SYSBUILD_CURRENT_MODULE_NAME}_pre_cmake)
```

Image configuration script (Zephyr-wide) Global Zephyr provided image configuration scripts, which allow specifying the type when using *ExternalZephyrProject_Add()* require changes to sysbuild code in Zephyr. This should only be added to when adding a new type that any project should be able to select, generally this should only be needed for upstream Zephyr though forked versions of Zephyr might use this to add additional types without restriction.

Image configuration has an allow-list of names which must be set in the Zephyr file share/sysbuild/cmake/modules/sysbuild_extensions.cmake in the *ExternalZephyrProject_Add()* function. After adding a new type, it can be used when adding a sysbuild image, for example:

`sysbuild_extensions.cmake`

Full file path: `share/sysbuild/cmake/modules/sysbuild_extensions.cmake`

```
# ...
# Usage:
#   ExternalZephyrProject_Add(APPLICATION <name>
#                             SOURCE_DIR <dir>
#                             [BOARD <board> [BOARD_REVISION <revision>]]
#                             [APP_TYPE <MAIN|BOOTLOADER|MY_CUSTOM_TYPE>]
#   )
# ...
# APP_TYPE <MAIN|BOOTLOADER|MY_CUSTOM_TYPE>: Application type.
#                                            MAIN indicates this application is␣
↪the main application
#                                            and where user defined settings␣
↪should be passed on as-is
#                                            except for multi image build flags.
#                                            For example, -DCONF_FILES=<files>␣
↪will be passed on to the
```

(continues on next page)

(continued from previous page)

```
#                                                MAIN_APP unmodified.
#                                                BOOTLOADER indicates this app is a␣
↪bootloader
#                                                MY_CUSTOM_TYPE indicates this app␣
↪is...
# ...
function(ExternalZephyrProject_Add)
  set(app_types MAIN BOOTLOADER MY_CUSTOM_TYPE)
# ...
```

sysbuild.cmake

```
if(SB_CONFIG_OTHER_APP_IMAGE_PATH)
  ExternalZephyrProject_Add(
    APPLICATION ${SB_CONFIG_OTHER_APP_IMAGE_NAME}
    SOURCE_DIR ${SB_CONFIG_OTHER_APP_IMAGE_PATH}
    APP_TYPE MY_CUSTOM_TYPE
  )
endif()
```

MY_CUSTOM_TYPE_image_default.cmake

Full file path: share/sysbuild/image_configurations/MY_CUSTOM_TYPE_image_default.cmake

```
# Here, the ZCMAKE_APPLICATION variable will be replaced with the image being␣
↪configured
set_config_bool(${ZCMAKE_APPLICATION} CONFIG_BUILD_OUTPUT_HEX y)
```

Reading image configuration

Kconfig Kconfig values from images can be read by sysbuild **after the CMake configuration of images has taken place**. This can be used to check configuration or to adjust additional sysbuild tasks depending upon configuration. The following function can be used for this purpose:

```
sysbuild_get(<variable> IMAGE <image> [VAR <image-variable>] KCONFIG)
```

This function can only be used when *sysbuild is extended by a module* or inside of a sysbuild/CMakeLists.txt file after find_package(Sysbuild) has been used. An example of outputting values from all images is shown:

Module CMake

```
function(${SYSBUILD_CURRENT_MODULE_NAME}_post_cmake)
  cmake_parse_arguments(POST_CMAKE "" "" "IMAGES" ${ARGN})

  foreach(image ${POST_CMAKE_IMAGES})
    # Note that the variable to read in to must not be set before using the␣
↪sysbuild_get() function
    set(tmp_val)
    sysbuild_get(tmp_val IMAGE ${image} VAR CONFIG_BUILD_OUTPUT_HEX KCONFIG)
    message(STATUS "Image ${image} build hex: ${tmp_val}")
  endforeach()
endfunction()
```

sysbuild/CMakeLists.txt

```
find_package(Sysbuild REQUIRED HINTS $ENV{ZEPHYR_BASE})

project(sysbuild LANGUAGES)

sysbuild_get(tmp_val IMAGE ${DEFAULT_IMAGE} VAR CONFIG_BUILD_OUTPUT_HEX KCONFIG)
message(STATUS "Image ${DEFAULT_IMAGE} build hex: ${tmp_val}")
```

5.7.16 Sysbuild and CMake presets

`CMake presets` can be used with Sysbuild but not all preset macros will work as expected.

> **Note**
>
> Using CMake presets with sysbuild requires CMake version 3.27 or higher.

As described in *Sysbuild (System build)* then sysbuild is a higher-level build system which means that when CMake presets are used together with sysbuild, then the preset is consumed and processed by sysbuild itself and result is passed to the application.

Running sysbuild with preset.

`west build`

Here is an example where preset `release` should be used. For details, see *Sysbuild (multi-domain builds)* in the `west build documentation`.

```
west build -b reel_board --sysbuild -- --preset=release samples/hello_world
```

`cmake`

Here is an example using CMake and Ninja.

```
APP_DIR=samples/hello_world cmake -Bbuild -GNinja -DBOARD=reel_board --
↪preset=release share/sysbuild
ninja -Cbuild
```

When using CMake presets with sysbuild then `APP_DIR` must be set in environment in order for Sysbuild CMake to be able to include the `CMakePresets.json` from the main Zephyr application's source directory.

> **Note**
>
> As sysbuild changes the top-level cmake project to its own directory, the cmake presets are parsed from there, the application's presets are included from this file verbatim. Therefore relative paths, and macros resolving relative to the source directory will not work as expected, but as relative to share/sysbuild, for example `${sourceDir}`.
>
> The `${fileDir}` macro can be used to create portable paths relative to the application's directory.

5.8 Application version management

Zephyr supports an application version management system for applications which is built around the system that Zephyr uses for its own version system management. This allows applications to define a version file and have application (or module) code include the auto-generated file and be able to access it, just as they can with the kernel version. This version information is available from multiple scopes, including:

- Code (C/C++)
- Kconfig
- CMake
- Shell

which makes it a very versatile system for lifecycle management of applications. In addition, it can be used when building applications which target supported bootloaders (e.g. MCUboot) allowing images to be signed with correct version of the application automatically - no manual signing steps are required.

5.8.1 VERSION file

Application version information is set on a per-application basis in a file named `VERSION`, which must be placed at the base directory of the application, where the CMakeLists.txt file is located. This is a simple text file which contains the various version information fields, each on a newline. The basic `VERSION` file has the following structure:

```
VERSION_MAJOR =
VERSION_MINOR =
PATCHLEVEL =
VERSION_TWEAK =
EXTRAVERSION =
```

Each field and the values it supports is described below. Zephyr limits the value of each numeric field to a single byte (note that there may be further restrictions depending upon what the version is used for, e.g. bootloaders might only support some of these fields or might place limits on the maximum values of fields):

Field	Data type	Required
VERSION_MAJOR	Numerical (0-255)	X
VERSION_MINOR	Numerical (0-255)	X
PATCHLEVEL	Numerical (0-255)	X
VERSION_TWEAK	Numerical (0-255)	X
EXTRAVERSION	Alphanumerical (Lowercase a-z and 0-9) and "." or "-"	

When an application is configured using CMake, the version file will be automatically processed, and will be checked automatically each time the version is changed, so CMake does not need to be manually re-ran for changes to this file.

For the sections below, examples are provided for the following `VERSION` file:

```
VERSION_MAJOR = 1
VERSION_MINOR = 2
PATCHLEVEL = 3
VERSION_TWEAK = 4
EXTRAVERSION = unstable.5
```

5.8.2 Application build version

The *application build version* is a separate, more dynamic concept from the one described in the `VERSION` file: it is intended to represent the current state of the application source code at the time of build, and is extracted from source versioning metadata.

By default, if the application is in a Git repository, its contents are generated from the output of the command `git describe --abbrev=12 --always` run in the application folder. This results in a string which contains the most recent tag, the number of commits since that tag, and the abbreviated commit hash.

Setting *APP_BUILD_VERSION* to the desired string value directly overrides this default behavior. This can either be done from the application `CMakeLists.txt` file or on the command line when invoking CMake, e.g. via `cmake -DAPP_BUILD_VERSION="v3.3.0-18-g2c85d92" ...`

5.8.3 Use in code

To use the version information in application code, the version file must be included, then the fields can be freely used. The include file name is `app_version.h` (use `#include <zephyr/app_version.h>`), the following defines are available:

Define	Type	Field(s)	Example
APPVERSION	Numerical	`VERSION_MAJOR` (left shifted by 24 bits), `VERSION_MINOR` (left shifted by 16 bits), `PATCHLEVEL` (left shifted by 8 bits), `VERSION_TWEAK`	0x1020304
APP_VERSION_N	Numerical	`VERSION_MAJOR` (left shifted by 16 bits), `VERSION_MINOR` (left shifted by 8 bits), `PATCHLEVEL`	0x10203
APP_VERSION_M	Numerical	`VERSION_MAJOR`	1
APP_VERSION_M	Numerical	`VERSION_MINOR`	2
APP_PATCHLEVI	Numerical	`PATCHLEVEL`	3
APP_TWEAK	Numerical	`VERSION_TWEAK`	4
APP_EXTRAVERS	String (unquoted)	`EXTRAVERSION`	unstable.5
APP_VERSION_S	String (quoted)	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `EXTRAVERSION`	"1.2.3-unstable.5"
APP_VERSION_E	String (quoted)	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `EXTRAVERSION` `VERSION_TWEAK`	"1.2.3-unstable.5+4"
APP_VERSION_T	String (quoted)	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `VERSION_TWEAK`	"1.2.3+4"
APP_BUILD_VER	String (unquoted)	None (see *Application build version* above)	v3.3.0-18-g2c85d92

5.8.4 Use in Kconfig

The following variables are available for usage in Kconfig files:

Variable	Type	Field(s)	Example
$(VERSION_MAJOR)	Numerical	VERSION_MAJOR	1
$(VERSION_MINOR)	Numerical	VERSION_MINOR	2
$(PATCHLEVEL)	Numerical	PATCHLEVEL	3
$(VERSION_TWEAK)	Numerical	VERSION_TWEAK	4
$(EXTRAVERSION)	String	EXTRAVERSION	unstable.5
$(APPVERSION)	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, EXTRAVERSION	1.2.3-unstable.5
$(APP_VERSION_EXTEN	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, EXTRAVERSION, VERSION_TWEAK	1.2.3-unstable.5+4
$(APP_VERSION_TWEAI	String	VERSION_MAJOR, VERSION_MINOR, PATCHLEVEL, VERSION_TWEAK	1.2.3+4

5.8.5 Use in CMake

The following variable are available for usage in CMake files:

Variable	Type	Field(s)	Example
APPVERSION	Numerical (hex)	`VERSION_MAJOR` (left shifted by 24 bits), `VERSION_MINOR` (left shifted by 16 bits), `PATCHLEVEL` (left shifted by 8 bits), `VERSION_TWEAK`	0x1020304
APP_VERSION_N	Numerical (hex)	`VERSION_MAJOR` (left shifted by 16 bits), `VERSION_MINOR` (left shifted by 8 bits), `PATCHLEVEL`	0x10203
APP_VERSION_M	Numerical	`VERSION_MAJOR`	1
APP_VERSION_M	Numerical	`VERSION_MINOR`	2
APP_PATCHLEVI	Numerical	`PATCHLEVEL`	3
APP_VERSION_T	Numerical	`VERSION_TWEAK`	4
APP_VERSION_E	String	`EXTRAVERSION`	unstable.5
APP_VERSION_S	String	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `EXTRAVERSION`	1.2.3-unstable.5
APP_VERSION_E	String	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `EXTRAVERSION`, `VERSION_TWEAK`	1.2.3-unstable.5+4
APP_VERSION_T	String	`VERSION_MAJOR`, `VERSION_MINOR`, `PATCHLEVEL`, `VERSION_TWEAK`	1.2.3+4
APP_BUILD_VER	String	None (see *Application build version* above)	v3.3.0-18-g2c85d92

5.8.6 Use in MCUboot-supported applications

No additional configuration needs to be done to the target application so long as it is configured to support MCUboot and a signed image is generated, the version information will be automatically included in the image data.

5.8.7 Use in Shell

When a shell interface is configured, the following commands are available to retrieve application version information:

Command	Variable	Example
app version	APP_VERSION_STRING	1.2.3-unstable.5
app version-extended	APP_VERSION_EXTENDED_STRING	1.2.3-unstable.5+4
app build-version	APP_BUILD_VERSION	v3.3.0-18-g2c85d92

5.9 Flashing

5.9.1 Flashing configuration

Zephyr supports setting up configuration for flash runners (invoked from *west flash*) which allows for customising how commands are used when programming boards. This configuration is used for *Sysbuild (System build)* projects and allows for configuring when commands are ran for groups of board targets. As an example: a multi-core SoC might want to only allow the `--erase` argument to be used once for all of the cores in the SoC which would prevent multiple erase tasks running in a single `west flash` invocation, which could wrongly clear the memory which is used by the other images being programmed.

Priority

Flashing configuration is singular, it will only be read from a single location, this configuration can reside in the following files starting with the highest priority:

- `soc.yml` (in soc folder)
- `board.yml` (in board folder)

Configuration

Configuration is applied in the yml file by using a `runners` map with a single `run_once` child, this then contains a map of commands as they would be provided to the flash runner e.g. `--reset` followed by a list which specifies the settings for each of these commands (these are grouped by flash runner, and by qualifiers/boards). Commands have associated runners that they apply to using a `runners` list value, this can contain `all` if it applies to all runners, otherwise must contain each runner that it applies to using the runner-specific name. Groups of board targets can be specified using the `groups` key which has a list of board target sets. Board targets are regular expression matches, for `soc.yml` files each set of board targets must be in a `qualifiers` key (only regular expression matches for board qualifiers are allowed, board names must be omitted from these entries). For `board.yml` files each set of board targets must be in a `boards` key, these are lists containing the matches which form a singular group. A final parameter `run` can be set to `first` which means that the command will be ran once with the first image flashing process per set of board targets, or to `last` which will be ran once for the final image flash per set of board targets.

An example flashing configuration for a `soc.yml` is shown below in which the `--recover` command will only be used once for any board targets which used the nRF5340 SoC application or network CPU cores, and will only reset the network or application core after all images for the respective core have been flashed.

```
runners:
  run_once:
    '--recover':
      - run: first
        runners:
          - nrfjprog
        groups:
          - qualifiers:
              - nrf5340/cpunet
              - nrf5340/cpuapp
              - nrf5340/cpuapp/ns
    '--reset':
      - run: last
        runners:
          - nrfjprog
          - jlink
        groups:
          - qualifiers:
```

(continues on next page)

(continued from previous page)

```
            - nrf5340/cpunet
        - qualifiers:
            - nrf5340/cpuapp
            - nrf5340/cpuapp/ns
    # Made up non-real world example to show how to specify different options␣
↪for different
    # flash runners
    - run: first
      runners:
        - some_other_runner
      groups:
        - qualifiers:
            - nrf5340/cpunet
        - qualifiers:
            - nrf5340/cpuapp
            - nrf5340/cpuapp/ns
```

Usage

Commands that are supported by flash runners can be used as normal when flashing non-sysbuild applications, the run once configuration will not be used. When flashing a sysbuild project with multiple images, the flash runner run once configuration will be applied.

For example, building the smp-svr sample for the nrf5340dk which will include MCUboot as a secondary image:

```
cmake -GNinja -Sshare/sysbuild/ -Bbuild -DBOARD=nrf5340dk/nrf5340/cpuapp -
↪DAPP_DIR=samples/subsys/mgmt/mcumgr/smp_svr
cmake --build build
```

Once built with an nrf5340dk connected, the following command can be used to flash the board with both applications and will only perform a single device recovery operation when programming the first image:

```
west flash --recover
```

If the above was ran without the flashing configuration, the recovery process would be ran twice and the device would be unbootable.

5.10 Signing Binaries

Binaries can be optionally signed as part of a build automatically using CMake code, there is also the ability to use `west sign` to sign binaries too, this page describes the former, the latter is documented on *Signing Binaries*.

5.10.1 MCUboot / imgtool

The Zephyr build system has special support for signing binaries for use with the MCUboot bootloader using the imgtool program provided by its developers. You can both build and sign this type of application binary in one step by setting some Kconfig options. If you do, `west flash` will use the signed binaries.

Here is an example workflow, which builds and flashes MCUboot, as well as the hello_world application for chain-loading by MCUboot. Run these commands from the `zephyrproject` workspace you created in the *Getting Started Guide*.

```
west build -b YOUR_BOARD zephyr/samples/hello_world --sysbuild -d␣
↪build-hello-signed -- \
    -DSB_CONFIG_BOOTLOADER_MCUBOOT=y

west flash -d build-hello-signed
```

Notes on the above commands:

- `YOUR_BOARD` should be changed to match your board
- The signing key value is the insecure default provided and used by MCUboot for development and testing
- You can change the `hello_world` application directory to any other application that can be loaded by MCUboot, such as the smp-svr sample.

For more information on these and other related configuration options, see:

- `SB_CONFIG_BOOTLOADER_MCUBOOT`: build the application for loading by MCUboot
- `SB_CONFIG_BOOT_SIGNATURE_KEY_FILE`: the key file, or a comma-separated list of key files, to use when signing images. If you have your own key, change this appropriately; see *Signing key files* for using an absolute path or a CMake variable such as `${APP_DIR}`. When a list is given, MCUboot embeds the public half of every key and accepts an image signed with any of them; the first entry also signs the application, and every entry past the first must be a public-only PEM of the same signature type for `imgtool`.
- `CONFIG_MCUBOOT_EXTRA_IMGTOOL_ARGS`: optional additional command line arguments for `imgtool`
- `CONFIG_MCUBOOT_GENERATE_CONFIRMED_IMAGE`: also generate a confirmed image, which may be more useful for flashing in production environments than the OTA-able default image
- On Windows, if you get "Access denied" issues, the recommended fix is to run `pip3 install imgtool`, then retry with a pristine build directory.

For a worked example of a multi-key bootloader exercised end-to-end under QEMU, see the tests/boot/mcuboot_multiple_keys test.

If your `west flash` *runner* uses an image format supported by imgtool, you should see something like this on your device's serial console when you run `west flash -d build-hello-signed`:

```
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
[00:00:00.004,669] <inf> mcuboot: Starting bootloader
[00:00:00.011,169] <inf> mcuboot: Primary image: magic=unset, swap_type=0x1,␣
↪copy_done=0x3, image_ok=0x3
[00:00:00.021,636] <inf> mcuboot: Boot source: none
[00:00:00.027,374] <inf> mcuboot: Swap type: none
[00:00:00.115,142] <inf> mcuboot: Bootloader chainload address offset: 0xc000
[00:00:00.123,168] <inf> mcuboot: Jumping to the first image slot
*** Booting Zephyr OS build zephyr-v2.3.0-2310-gcebac69c8ae1  ***
Hello World! nrf52840dk_nrf52840
```

Whether `west flash` supports this feature depends on your runner. The `nrfjprog` and `pyocd` runners work with the above flow. If your runner does not support this flow and you would like it to, please send a patch or file an issue for adding support.

5.10.2 Signing key files

When building with sysbuild, `SB_CONFIG_BOOT_SIGNATURE_KEY_FILE` selects the key used to sign images. Its value is propagated to two images:

- the application image, as `CONFIG_MCUBOOT_SIGNATURE_KEY_FILE`, which `imgtool` uses to sign the application; and

- the MCUboot image, as CONFIG_BOOT_SIGNATURE_KEY_FILE, whose public part is built into the bootloader so it can verify that signature.

Warning

The default value points at one of the insecure development keys bundled with MCUboot (for example root-ec-p256.pem). These keys are public — they ship with every Zephyr and MCUboot checkout — so they are only suitable for development and testing. For anything else, generate your own key and keep the private key out of any repository or build directory that you do not control.

How sysbuild resolves the key file path

The value of SB_CONFIG_BOOT_SIGNATURE_KEY_FILE is processed with CMake's string(CONFIGURE) command, so any ${VARIABLE} reference it contains is expanded as a CMake variable before the path is used. After expansion, an absolute path is used as-is; a relative path is resolved by each image independently:

Image	Search order for a relative path
Application (CONFIG_MCUBOOT_SIGNATURE_KE	APPLICATION_CONFIG_DIR, then the west workspace topdir (WEST_TOPDIR)
MCUboot (CONFIG_BOOT_SIGNATURE_KEY_F	APPLICATION_CONFIG_DIR, then the MCUboot module directory

Warning

Each image resolves a relative path against a different base, so a bare relative path points at a different file for each image and usually fails to build. Do not use one: give an absolute path, or anchor the path with a CMake variable.

${APP_DIR} (the main application's source directory) is the recommended anchor: it keeps the key inside your own application, and both images resolve it to the same file. Any CMake variable available when the value is expanded may be used:

```
# sysbuild.conf
SB_CONFIG_BOOT_SIGNATURE_KEY_FILE="${APP_DIR}/keys/my-signing-key.pem"
```

The same resolution rules apply to the optional encryption key file (CONFIG_MCUBOOT_ENCRYPTION_KEY_FILE).

5.10.3 Extending signing externally

The signing script used when running west flash can be extended or replaced to change features or introduce different signing mechanisms. By default with MCUboot enabled, signing is setup by the cmake/mcuboot.cmake file in Zephyr which adds extra post build commands for generating the signed images. The file used for signing can be replaced from a sysbuild scope (if being used) or from a zephyr/zephyr module scope, the priority of which is:

- Sysbuild
- Zephyr property
- Default MCUboot script (if enabled)

From sysbuild, -D<target>_SIGNING_SCRIPT can be used to set a signing script for a specific image or -DSIGNING_SCRIPT can be used to set a signing script for all images, for example:

```
west build -b <board> <application> -DSIGNING_SCRIPT=<file>
```

The zephyr property method is achieved by adjusting the `SIGNING_SCRIPT` property on the `zephyr_property_target`, ideally from by a module by using:

```
if(CONFIG_BOOTLOADER_MCUBOOT)
  set_target_properties(zephyr_property_target PROPERTIES SIGNING_SCRIPT $
↪{CMAKE_CURRENT_LIST_DIR}/custom_signing.cmake)
endif()
```

This will include the custom signing CMake file instead of the default Zephyr one when projects are built with MCUboot signing support enabled. The base Zephyr MCUboot signing file can be used as a reference for creating a new signing system or extending the default behaviour.

Chapter 6

Hardware Support

6.1 Architecture-related Guides

6.1.1 Zephyr support status on ARC processors

Overview

This page describes current state of Zephyr for ARC processors and some future plans. Please note that

- plans are given without exact deadlines
- software features require corresponding hardware to be present and configured the proper way
- not all the features can be enabled at the same time

Support status

Legend: **Y** - yes, supported; **N** - no, not supported; **WIP** - Work In Progress; **TBD** - to be decided

	Processor families				
	EM	**HS3x/4x**	**VPX**	**HS5x**	**HS6x**
Port status	up-streamed	up-streamed	up-streamed	up-streamed	up-streamed
Features					
Closely coupled memories (ICCM, DCCM)[1]	Y	Y	Y	TBD	TBD
Execution with caches - Instruction/Data, L1/L2 caches	Y	Y	Y	Y	Y
Hardware-assisted unaligned memory access	Y[2]	Y	Y	Y	Y
Regular interrupts with multiple priority levels, direct interrupts	Y	Y	Y	Y	Y
Fast interrupts, separate register banks for fast interrupts	Y	Y	TBD	N	N
Hardware floating point unit (FPU)	Y	Y	TBD[6]	TBD	TBD
Symmetric multiprocessing (SMP) support, switch-based	N/A	Y	TBD	Y	Y
Hardware-assisted stack checking	Y	Y	Y	N	N
Hardware-assisted atomic operations	N/A	Y	Y	Y	Y
DSP ISA	Y	N[3]	TBD[6]	TBD	TBD
DSP AGU/XY extensions	Y	N[3]	N/A	TBD	TBD
Userspace	Y	Y	N	TBD	TBD
Memory protection unit (MPU)	Y	Y	TBD	N	N
Memory management unit (MMU)	N/A	N	TBD	N	N
SecureShield	Y	N/A	N/A	N/A	N/A
Single-thread kernel support[5]	Y	Y	Y	Y	Y
Toolchains					
GNU (open source GCC-based)	Y	Y	N	Y	Y
MetaWare (proprietary Clang-based)	Y	Y	Y	Y	Y
Simulators					
QEMU (open source)[4]	Y	Y	N	Y	Y
nSIM (proprietary, provided by MetaWare Development Tools)	Y	Y	Y	Y	Y

Notes

VPX Vector Registers Zephyr supports a limited form sharing of the VPX vector registers known as cooperative sharing. Threads that use these registers must bookend the relevant sections with calls to `arc_vpx_lock()` and `arc_vpx_unlock()` to control access to this resource.

> **Note**
>
> If the system has multiple CPUs, then it is the responsibility of the application developer to both pin the thread to a single CPU before it attempts to get the cooperative lock, and not modify the CPU affinity while it is waiting for or holding that cooperative lock.

[6] currently only ARC VPX scalar port is supported. The support of VPX vector pipeline, VCCM, STU is not included in this port, and require additional development and / or other runtime integration.

[1] usage of CCMs is limited on SMP systems

[2] except the systems with secure features (SecureShield) due to HW limitation

[3] We only support save/restore ACCL/ACCH registers in task's context. Rest of DSP/AGU registers save/restore isn't implemented but kernel itself does not use these registers. This allows single task per core to use DSP/AGU safely.

[5] Single-thread kernel is support only for single core targets

[4] QEMU doesn't support all the ARC processor's HW features. For the detailed info please check the ARC QEMU documentation

Configuration Options The cooperative sharing of the VPX vector registers is selected when configuration option `CONFIG_ARC_VPX_COOPERATIVE_SHARING` is enabled.

6.1.2 Arm Cortex-M Developer Guide

Overview

This page contains detailed information about the status of the Arm Cortex-M architecture porting in the Zephyr RTOS and describes key aspects when developing Zephyr applications for Arm Cortex-M-based platforms.

Key supported features

The table below summarizes the status of key OS features in the different Arm Cortex-M implementation variants.

Architecture variant		Arm v6-M		Arm v7-M
		M0/M1	**M0+**	**M3**
OS Features				
Programmable fault IRQ priorities		N	N	Y
Single-thread kernel support		Y	Y	Y
Thread local storage support		Y	Y	Y
Interrupt handling				
	Regular interrupts	Y	Y	Y
	Dynamic interrupts	Y	Y	Y

	Direct interrupts	Y	Y	Y
	Zero Latency interrupts	N	N	Y
CPU idling		Y	Y	Y
Native system timer (SysTick)		N[1]	Y	Y
Memory protection				
	User mode	N	Y	Y
	HW stack protection (MPU)	N	N	Y
	HW-assisted stack limit checking	N	N	N
	Privileged Execute Never[3]	N	N	N
HW-assisted Control Flow integrity	PACBTI	N	N	N

HW-assisted null-pointer dereference detection		N	N	Y
HW-assisted atomic operations		N	N	Y
Support for non-cacheable regions		N	N	Y
Execute SRAM functions		N	N	Y
Floating Point Services		N	N	N
DSP ISA		N	N	N
Trusted Execution				
	Native TrustZone-M support	N	N	N
	TF-M integration	N	N	N
Code relocation		Y	Y	Y
SW-based vector table relaying		Y	Y	Y

HW-assisted timing functions	N	N	Y

Notes

OS features

Threads

Thread stack alignment Each Zephyr thread is defined with its own stack memory. By default, Cortex-M enforces a double word thread stack alignment, see `CONFIG_STACK_ALIGN_DOUBLE_WORD`. If MPU-based HW-assisted stack overflow detection (`CONFIG_MPU_STACK_GUARD`) is enabled, thread stacks need to be aligned with a larger value, reflected by `CONFIG_ARM_MPU_REGION_MIN_ALIGN_AND_SIZE`. In Arm v6-M and Arm v7-M architecture variants, thread stacks are additionally required to align with a value equal to their size, in applications that need to support user mode (`CONFIG_USERSPACE`). The thread stack sizes in that case need to be a power of two. This is all reflected by `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT`, that is enforced in Arm v6-M and Arm v7-M builds with user mode support.

Stack pointers While executing in thread mode the processor is using the Process Stack Pointer (PSP). The processor uses the Main Stack Pointer (MSP) while executing in handler mode, that is, while servicing exceptions and HW interrupts. Using PSP in thread mode *facilitates thread stack pointer manipulation* during thread context switching, without affecting the current execution context flow in handler mode.

In Arm Cortex-M builds a single interrupt stack memory is shared among exceptions and interrupts. The size of the interrupt stack needs to be selected taking into consideration nested interrupts, each pushing an additional stack frame. Developers can modify the interrupt stack size using `CONFIG_ISR_STACK_SIZE`.

The interrupt stack is also used during early boot so the kernel can initialize the main thread's stack before switching to the main thread.

Thread context switching In Arm Cortex-M builds, the PendSV exception is used in order to trigger a context switch to a different thread. PendSV exception is always present in Cortex-M implementations. PendSV is configured with the lowest possible interrupt priority level, in all Cortex-M variants. The main reasons for that design are

- to utilize the tail chaining feature of Cortex-M processors, and thus limit the number of context switch operations that occur.
- to not impact the interrupt latency observed by HW interrupts.

As a result, context switch in Cortex-M is non-atomic, i.e. it may be *preempted* by HW interrupts, however, a context-switch operation must be completed before a new thread context-switch may start.

Typically a thread context-switch will perform the following operations

[1] SysTick is optional in Cortex-M1
[2] Stack limit checking only in Secure builds in Cortex-M23
[3] https://developer.arm.com/documentation/107655/100/RTOS-and-Secure-software-design-considerations/Secure-software-development-design-considerations/Security-and-privilege-combination/Using-PXN-bit?lang=en

- When switching-out the current thread, the processor stores
  - the callee-saved registers (R4 - R11) in the thread's container for callee-saved registers, which is located in kernel memory
  - the thread's current operation *mode*
    - user or privileged execution mode
    - presence of an active floating point context
    - the EXC_RETURN value of the current handler context (PendSV)
  - the floating point callee-saved registers (S16 - S31) in the thread's container for FP callee-saved registers, if the current thread has an active FP context
  - the PSP of the current thread which points to the beginning of the current thread's exception stack frame. The latter contains the caller-saved context and the return address of the switched-out thread.
- When switching-in a new thread the processor
  - restores the new thread's callee-saved registers from the thread's container for callee-saved registers
  - restores the new thread's operation *mode*
  - restores the FP callee-saved registers if the switched-in thread had an active FP context before being switched-out
  - re-programs the dynamic MPU regions to allow a user thread access its stack and application memories, and/or programs a stack-overflow MPU guard at the bottom of the thread's privileged stack
  - restores the PSP for the incoming thread and re-programs the stack pointer limit register (if applicable, see `CONFIG_BUILTIN_STACK_GUARD`)
  - optionally does a stack limit checking for the switched-in thread, if sentinel-based stack limit checking is enabled (see `CONFIG_STACK_SENTINEL`).

PendSV exception return sequence restores the new thread's caller-saved registers and the return address, as part of unstacking the exception stack frame.

The implementation of the context-switch mechanism is present in `arch/arm/core/cortex_m/swap_helper.S`.

Stack limit checking (Arm v8-M) Armv8-M and Armv8.1-M variants support stack limit checking using the MSPLIM and PSPLIM core registers. The feature is enabled when `CONFIG_BUILTIN_STACK_GUARD` is set. When stack limit checking is enabled, both the thread's privileged or user stack, as well as the interrupt stack are guarded by PSPLIM and MSPLIM registers, respectively. MSPLIM is configured *once* during kernel boot, while PSLIM is re-programmed during every thread context-switch or during system calls, when the thread switches from using its default stack to using its privileged stack, and vice versa. PSPLIM re-programming

- has a relatively low runtime overhead (programming is done with MSR instructions)
- does not impact interrupt latency
- does not require any memory areas to be reserved for stack guards
- does not make use of MPU regions

It is, therefore, considered as a lightweight but very efficient stack overflow detection mechanism in Cortex-M applications.

Stack overflows trigger the dedicated UsageFault exception provided by Arm v8-M.

Interrupt handling features This section describes certain aspects around exception and interrupt handling in Arm Cortex-M.

Interrupt priority levels The number of available (configurable) interrupt priority levels is determined by the number of implemented interrupt priority bits in NVIC; this needs to be described for each Cortex-M platform using DeviceTree:

```
&nvic {
        arm,num-irq-priority-bits = <#priority-bits>;
};
```

Reserved priority levels A number of interrupt priority levels are reserved for the OS.

By design, system fault exceptions have the highest priority level. In *Baseline* Cortex-M, this is actually enforced by hardware, as HardFault is the only available processor fault exception, and its priority is higher than any configurable exception priority.

In *Mainline* Cortex-M, the available fault exceptions (e.g. MemManageFault, UsageFault, etc.) are assigned the highest *configurable* priority level. (`CONFIG_CPU_CORTEX_M_HAS_PROGRAMMABLE_FAULT_PRIOS` signifies explicitly that the Cortex-M implementation supports configurable fault priorities.)

This priority level is never shared with HW interrupts (an exception to this rule is described below). As a result, processor faults occurring in regular ISRs will be handled by the corresponding fault handler and will not escalate to a HardFault, *similar to processor faults occurring in thread mode*.

SVC exception is normally configured with the highest configurable priority level (an exception to this rule will be described below). SVCs are used by the Zephyr kernel to dispatch system calls, trigger runtime system errors (e.g. Kernel oops or panic), or implement IRQ offloading.

In Baseline Cortex-M the priority level of SVC may be shared with other exceptions or HW interrupts that are also given the highest configurable priority level (As a result of this, kernel runtime errors during interrupt handling will escalate to HardFault. Additional logic in the fault handling routines ensures that such runtime errors are detected successfully).

In Mainline Cortex-M, however, the SVC priority level is *reserved*, thus normally it is only shared with the fault exceptions of configurable priority. This simplifies the fault handling routines in Mainline Cortex-M architecture, since runtime kernel errors are serviced by the SVC handler (i.e no HardFault escalation, even if the kernel errors occur in ISR context).

HW interrupts in Mainline Cortex-M builds are allocated a priority level lower than the SVC.

One exception to the above rules is when Zephyr applications support Zero Latency Interrupts (ZLIs). Such interrupts are designed to have a priority level higher than any HW or system interrupt. If the ZLI feature is enabled in Mainline Cortex-M builds (see `CONFIG_ZERO_LATENCY_IRQS`), then

- ZLIs are assigned the highest configurable priority level
- SVCs are assigned the second highest configurable priority level
- Regular HW interrupts are assigned priority levels lower than SVC.

The priority level configuration in Cortex-M is implemented in `include/zephyr/arch/arm/cortex_m/exception.h`.

Locking and unlocking IRQs In Baseline Cortex-M locking interrupts is implemented using the PRIMASK register.

```
arch_irq_lock()
```

will set the PRIMASK register to 1, eventually, masking all IRQs with configurable priority. While this fulfils the OS requirement of locking interrupts, the consequence is that kernel runtime errors (triggering SVCs) will escalate to HardFault.

In Mainline Cortex-M locking interrupts is implemented using the BASEPRI register (Mainline Cortex-M builds select `CONFIG_CPU_CORTEX_M_HAS_BASEPRI` to signify that BASEPRI register is implemented.). By modifying BASEPRI (or BASEPRI_MAX) arch_irq_lock() masks all system and HW interrupts with the exception of

- SVCs
- processor faults
- ZLIs

This allows zero latency interrupts to be triggered inside OS critical sections. Additionally, this allows system (processor and kernel) faults to be handled by Zephyr in *exactly the same way*, regardless of whether IRQs have been locked or not when the error occurs. It also allows for system calls to be dispatched while IRQs are locked.

> **Note**
>
> Mainline Cortex-M fault handling is designed and configured in a way that all processor and kernel faults are handled by the corresponding exception handlers and never result in HardFault escalation. In other words, a HardFault may only occur in Zephyr applications that have modified the default fault handling configurations. The main reason for this design was to reserve the HardFault exception for handling exceptional error conditions in safety critical applications.

Dynamic direct interrupts Cortex-M builds support the installation of direct interrupt service routines during runtime. Direct interrupts are designed for performance-critical interrupt handling and do not go through all of the common Zephyr interrupt handling code.

Direct dynamic interrupts are enabled via switching on `CONFIG_DYNAMIC_DIRECT_INTERRUPTS`.

Note that enabling direct dynamic interrupts requires enabling support for dynamic interrupts in the kernel, as well (see `CONFIG_DYNAMIC_INTERRUPTS`).

Zero Latency interrupts As described above, in Mainline Cortex-M applications, the Zephyr kernel reserves the highest configurable interrupt priority level for its own use (SVC). SVCs will not be masked by interrupt locking. Zero-latency interrupt can be used to set up an interrupt at the highest interrupt priority which will not be blocked by interrupt locking. To use the ZLI feature `CONFIG_ZERO_LATENCY_IRQS` needs to be enabled.

Zero latency IRQs have minimal interrupt latency, as they will always preempt regular HW or system interrupts.

Note, however, that since ZLI ISRs will run at a priority level higher than the kernel exceptions they **cannot use** any kernel functionality. Additionally, since the ZLI interrupt priority level is equal to processor fault priority level, faults occurring in ZLI ISRs will escalate to HardFault and will not be handled in the same way as regular processor faults. Developers need to be aware of this limitation.

CPU Idling The Cortex-M architecture port implements both k_cpu_idle() and k_cpu_atomic_idle(). The implementation is present in `arch/arm/core/cortex_m/cpu_idle.c`.

In both implementations, the processor will attempt to put the core to low power mode. In k_cpu_idle() the processor ends up executing WFI (Wait For Interrupt) instruction, while in k_cpu_atomic_idle() the processor will execute a WFE (Wait For Event) instruction.

When using the CPU idling API in Cortex-M it is important to note the following:

- Both k_cpu_idle() and k_cpu_atomic_idle() are *assumed* to be invoked with interrupts locked. This is taken care of by the kernel if the APIs are called by the idle thread.
- After waking up from low power mode, both functions will *restore* interrupts unconditionally, that is, regardless of the interrupt lock status before the CPU idle API was called.

The Zephyr CPU Idling mechanism is detailed in *CPU Idling*.

Memory protection features This section describes certain aspects around memory protection features in Arm Cortex-M applications.

User mode system calls User mode is supported in Cortex-M platforms that implement the standard (Arm) MPU or a similar core peripheral logic for memory access policy configuration and control, such as the NXP MPU for Kinetis platforms. (Currently, `CONFIG_ARCH_HAS_USERSPACE` is selected if `CONFIG_ARM_MPU` is enabled by the user in the board default Kconfig settings).

A thread performs a system call by triggering a (synchronous) SVC exception, where

- up to 5 arguments are placed on registers R1 - R5
- system call ID is placed on register R6.

The SVC Handler will branch to the system call preparation logic, which will perform the following operations

- switch the thread's PSP to point to the beginning of the thread's privileged stack area, optionally reprogramming the PSPLIM if stack limit checking is enabled
- modify CONTROL register to switch to privileged mode
- modify the return address in the SVC exception stack frame, so that after exception return the system call dispatcher is executed (in thread privileged mode)

Once the system call execution is completed the system call dispatcher will restore the user's original PSP and PSPLIM and switch the CONTROL register back to unprivileged mode before returning back to the caller of the system call.

System calls execute in thread mode and can be preempted by interrupts at any time. A thread may also be context-switched-out while doing a system call; the system call will resume as soon as the thread is switched-in again.

The system call dispatcher executes at SVC priority, therefore it cannot be preempted by HW interrupts (with the exception of ZLIs), which may observe some additional interrupt latency if they occur during a system call preparation.

MPU-assisted stack overflow detection Cortex-M platforms with MPU may enable `CONFIG_MPU_STACK_GUARD` to enable the MPU-based stack overflow detection mechanism. The following points need to be considered when enabling the MPU stack guards

- stack overflows are triggering processor faults as soon as they occur
- the mechanism is essential for detecting stack overflows in supervisor threads, or user threads in privileged mode; stack overflows in threads in user mode will always be detected regardless of `CONFIG_MPU_STACK_GUARD` being set.
- stack overflows are always detected, however, the mechanism does not guarantee that no memory corruption occurs when supervisor threads overflow their stack memory
- `CONFIG_MPU_STACK_GUARD` will normally reserve one MPU region for programming the stack guard (in certain Arm v8-M configurations with `CONFIG_MPU_GAP_FILLING` enabled 2 MPU regions are required to implement the guard feature)
- MPU guards are re-programmed at every context-switch, adding a small overhead to the thread swap routine. Compared, however, to the `CONFIG_BUILTIN_STACK_GUARD` feature, no re-programming occurs during system calls.

- When CONFIG_HW_STACK_PROTECTION is enabled on Arm v8-M platforms the native stack limit checking mechanism is used by default instead of the MPU-based stack overflow detection mechanism; users may override this setting by manually enabling CONFIG_MPU_STACK_GUARD in these scenarios.

Pointer Authentication and Branch Target Identification (PACBTI) The Armv8.1-M Pointer Authentication and Branch Target Identification (PACBTI) extension is an optional extension for the Armv8.1-M architecture profile and consists of the implementation of the following control-flow integrity approaches:

- Return address signing and authentication (PAC-RET) as a mitigation for Return Oriented Programming (ROP) style attack.
- BTI instruction placement (BTI) as a mitigation for Jump Oriented Programming (JOP) style attacks.

When hardware support is present (e.g., Cortex-M85) and compiler support is available, PACBTI can be enabled at build time in Zephyr by selecting one of the below configs:

- CONFIG_ARM_PACBTI_STANDARD
- CONFIG_ARM_PACBTI_PACRET
- CONFIG_ARM_PACBTI_PACRET_LEAF
- CONFIG_ARM_PACBTI_BTI
- CONFIG_ARM_PACBTI_PACRET_BTI
- CONFIG_ARM_PACBTI_PACRET_LEAF_BTI
- CONFIG_ARM_PACBTI_NONE

The config options ensures that compiler flags enabling PACBTI instructions are added to the build, specifically:

- -mbranch-protection= for GCC toolchains.

Further, CONFIG_ARM_PAC and CONFIG_ARM_BTI are automatically selected based on the branch protection option chosen for CONFIG_ARM_PACBTI. These configuration options enforce PACBTI by enabling corresponding PACBTI bits in CONTROL register and in the FVP.

To further enhance pointer authentication, Zephyr supports using cryptographically secure, per-thread PAC keys by enabling CONFIG_ARM_PAC_PER_THREAD. For more details on key generation sources and configuration, refer to the Kconfig help for CONFIG_ARM_PAC_PER_THREAD.

Limitations:

- Only builds targeting Armv8.1-M Mainline processors with PACBTI hardware support (e.g., Cortex-M85) are able to fully use this feature.
- Zephyr's integrated SDK currently includes GCC 12.2 which does not support PACBTI so external GCC toolchains (14.3 or later recommended) must be used for PACBTI support. Refer to this document on how to set up toolchains.

For more information about PACBTI, refer to the official Arm documentation and also Arm community blog

Memory map and MPU considerations

Fixed MPU regions By default, when CONFIG_ARM_MPU is enabled a set of *fixed* MPU regions are programmed during system boot.

- One MPU region programs the entire flash area as read-execute. User can override this setting by enabling CONFIG_MPU_ALLOW_FLASH_WRITE, which programs the flash with RWX

permissions. If CONFIG_USERSPACE is enabled unprivileged access on the entire flash area is allowed.

- One MPU region programs the entire SRAM area with privileged-only RW permissions. That is, an MPU region is utilized to disallow execute permissions on SRAM. (An exception to this setting is when CONFIG_MPU_GAP_FILLING is disabled (Arm v8-M only); in that case no SRAM MPU programming is done so the access is determined by the default Arm memory map policies, allowing for privileged-only RWX permissions on SRAM).
- All the memory regions defined in the devicetree with the property zephyr,memory-attr defining the MPU permissions for the memory region. See the next section for more details.

The above MPU regions are defined in arch/arm/core/mpu/arm_mpu_regions.c. Alternative MPU configurations are allowed by enabling CONFIG_CPU_HAS_CUSTOM_FIXED_SOC_MPU_REGIONS. When enabled, this option signifies that the Cortex-M SoC will define and configure its own fixed MPU regions in the SoC definition.

Fixed MPU regions defined in devicetree When the property zephyr,memory-attr is present in a memory node, a new MPU region will be allocated and programmed during system boot. When used with the zephyr,memory-region devicetree compatible, it will result in a linker section being generated associated to that MPU region.

For example, to define a new non-cacheable memory region in devicetree:

```
sram_no_cache: memory@20300000 {
     compatible = "zephyr,memory-region", "mmio-sram";
     reg = <0x20300000 0x100000>;
     zephyr,memory-region = "SRAM_NO_CACHE";
     zephyr,memory-attr = <DT_MEM_ARM_MPU_RAM_NOCACHE>;
};
```

This will automatically create a new MPU entry in with the correct name, base, size and attributes gathered directly from the devicetree. See *Caching Basics* and *Memory Attributes* for more details.

Static MPU regions Additional *static* MPU regions may be programmed once during system boot. These regions are required to enable certain features. See *Caching Basics* for more details.

- a RX region to allow execution from SRAM, when CONFIG_ARCH_HAS_RAMFUNC_SUPPORT is enabled and users have defined functions to execute from SRAM.
- a RX region for relocating text sections to SRAM, when CONFIG_CODE_DATA_RELOCATION_SRAM is enabled
- a nocache region to allow for a non-cacheable SRAM area, when CONFIG_NOCACHE_MEMORY is enabled
- a possibly unprivileged RW region for GCOV code coverage accounting area, when CONFIG_COVERAGE_GCOV is enabled
- a no-access region to implement null pointer dereference detection, when CONFIG_NULL_POINTER_EXCEPTION_DETECTION_MPU is enabled

The boundaries of these static MPU regions are derived from symbols exposed by the linker, in include/linker/linker-defs.h.

Dynamic MPU regions Certain thread-specific MPU regions may be re-programmed dynamically, at each thread context switch:

- an unprivileged RW region for the current thread's stack area (for user threads)
- a read-only region for the MPU stack guard

- unprivileged RW regions for the partitions of the current thread's application memory domain.

Considerations The number of available MPU regions for a Cortex-M platform is a limited resource. Most platforms have 8 MPU regions, while some Cortex-M33 or Cortex-M7 platforms may have up to 16 MPU regions. Therefore there is a relatively strict limitation on how many fixed, static and dynamic MPU regions may be programmed simultaneously. For platforms with 8 available MPU regions it might not be possible to enable all the aforementioned features that require MPU region programming. In most practical applications, however, only a certain set of features is required and 8 MPU regions are, in many cases, sufficient.

In Arm v8-M processors the MPU architecture does not allow programmed MPU regions to overlap. `CONFIG_MPU_GAP_FILLING` controls whether the fixed MPU region covering the entire SRAM is programmed. When it does, a full SRAM area partitioning is required, in order to program the static and the dynamic MPU regions. This increases the total number of required MPU regions. When `CONFIG_MPU_GAP_FILLING` is not enabled the fixed MPU region covering the entire SRAM is not programmed, thus, the static and dynamic regions are simply programmed on top of the always-existing background region (full-SRAM partitioning is not required). Note, however, that the background SRAM region allows execution from SRAM, so when `CONFIG_MPU_GAP_FILLING` is not set Zephyr is not protected against attacks that attempt to execute malicious code from SRAM.

Floating point Services Both unshared and shared FP registers mode are supported in Cortex-M (see *Floating Point Services* for more details).

When FPU support is enabled in the build (`CONFIG_FPU` is enabled), the sharing FP registers mode (`CONFIG_FPU_SHARING`) is enabled by default. This is done as some compiler configurations may activate a floating point context by generating FP instructions for any thread, regardless of whether floating point calculations are performed, and that context must be preserved when switching such threads in and out.

The developers can still disable the FP sharing mode in their application projects, and switch to Unshared FP registers mode, if it is guaranteed that the image code does not generate FP instructions outside the single thread context that is allowed (and supposed) to do so.

Under FPU sharing mode, the callee-saved FPU registers are saved and restored in context-switch, if the corresponding threads have an active FP context. This adds some runtime overhead on the swap routine. In addition to the runtime overhead, the sharing FPU mode

- requires additional memory for each thread to save the callee-saved FP registers
- requires additional stack memory for each thread, to stack the caller-saved FP registers, upon exception entry, if an FP context is active. Note, however, that since lazy stacking is enabled, there is no runtime overhead of FP context stacking in regular interrupts (FP state preservation is only activated in the swap routine in PendSV interrupt).

Misc

Chain-loadable images Cortex-M applications may either be standalone images or chain-loadable, for instance, by a bootloader. Application images chain-loadable by bootloaders (or other applications) normally occupy a specific area in the flash denoted as their *code partition*. `CONFIG_USE_DT_CODE_PARTITION` will ensure that a Zephyr chain-loadable image will be linked into its code partition, specified in DeviceTree.

HW initialization at boot In order to boot properly, chain-loaded applications may require that the core Arm hardware registers and peripherals are initialized in their reset values. Enabling `CONFIG_INIT_ARCH_HW_AT_BOOT` Zephyr to force the initialization of the internal Cortex-M architectural state during boot to the reset values as specified by the corresponding Arm architecture manual.

Software vector relaying In Cortex-M platforms that implement the VTOR register (see CONFIG_CPU_CORTEX_M_HAS_VTOR), chain-loadable images relocate the Cortex-M vector table by updating the VTOR register with the offset of the image vector table.

Baseline Cortex-M platforms without VTOR register might not be able to relocate their vector table which remains at a fixed location. Therefore, a chain-loadable image will require an alternative way to route HW interrupts and system exceptions to its own vector table; this is achieved with software vector relaying.

When a bootloader image enables CONFIG_SW_VECTOR_RELAY it is able to relay exceptions and interrupts based on a vector table pointer that is set by the chain-loadable application. The latter sets the CONFIG_SW_VECTOR_RELAY_CLIENT option to instruct the boot sequence to set the vector table pointer in SRAM so that the bootloader can forward the exceptions and interrupts to the chain-loadable image's software vector table.

While this feature is intended for processors without VTOR register, it may also be used in Mainline Cortex-M platforms.

Code relocation Cortex-M support the code relocation feature. When CONFIG_CODE_DATA_RELOCATION_SRAM is selected, Zephyr will relocate .text, data and .bss sections from the specified files and place it in SRAM. It is possible to relocate only parts of the code sections into SRAM, without relocating the whole image text and data sections. More details on the code relocation feature can be found in *Code And Data Relocation*.

Linking Cortex-M applications

Most Cortex-M platforms make use of the default Cortex-M GCC linker script in `include/zephyr/arch/arm/cortex_m/scripts/linker.ld`, although it is possible for platforms to use a custom linker script as well.

CMSIS

Cortex-M CMSIS headers are provided through standalone module repositories:

- **CMSIS 5**: zephyrproject-rtos/cmsis
- **CMSIS 6**: zephyrproject-rtos/CMSIS_6

Zephyr has begun transitioning to **CMSIS 6** as the default source for Cortex-M core headers. However, at present, Zephyr includes headers from **both** the CMSIS 6 and legacy CMSIS 5 modules.

The legacy CMSIS 5 headers remain available primarily for compatibility with vendor HALs, while all new architecture-level development should use **CMSIS 6** headers whenever possible.

CONFIG_CPU_CORTEX_M selects CONFIG_HAS_CMSIS_CORE to signify that CMSIS headers are available for all supported Cortex-M variants.

Testing

A list of unit tests for the Cortex-M porting and miscellaneous features is present in `tests/arch/arm/`. The tests suites are continuously extended and new test suites are added, in an effort to increase the coverage of the Cortex-M architecture support in Zephyr.

QEMU

We use QEMU to verify the implemented features of the Cortex-M architecture port in Zephyr. Adequate coverage is achieved by defining and utilizing a list of QEMU targets, each with a specific architecture variant and Arm peripheral support list.

The table below lists the QEMU platform targets defined in Zephyr along with the corresponding Cortex-M implementation variant and the peripherals these targets emulate.

	QEMU target				
Architecture variant	Arm v6-M	Arm v7-M		Arm v8-M	Arm v8.1-M
	qemu_cortex	**qemu_cortex**	**mps2/an3**	**mps2/an521/c**	**mps3/corstone300/an547**
Emulated features					
NVIC	Y	Y	Y	Y	Y
BASEPRI	N	Y	Y	Y	Y
SysTick	N	Y	Y	Y	Y
MPU	N	N	Y	Y	Y
FPU	N	N	N	Y	N
SPLIM	N	N	N	Y	Y
TrustZone-M	N	N	N	Y	N

Maintainers & Collaborators

The status of the Arm Cortex-M architecture port in Zephyr is: *maintained*. The updated list of maintainers and collaborators for Cortex-M can be found in `MAINTAINERS.yml`.

6.1.3 ARM64 Developer Guide

Wait with timeout (WFxT)

The Arm WFxT extension provides the WFET and WFIT instructions, which take an absolute virtual counter value as a timeout. Zephyr detects the extension at runtime from the `WFxT` field in `ID_AA64ISAR2_EL1`. Arm64 architecture code can query this support with `is_wfxt_implemented()`, declared in include/zephyr/arch/arm64/lib_helpers.h.

When the Arm architectural timer implements `arch_busy_wait()`, the Arm64 path uses WFET for `k_busy_wait()` if WFxT is implemented. It computes a deadline from `CNTVCT_EL0` and repeats WFET until the counter reaches that deadline. The loop is required because WFET is permitted to return before its timeout. CPUs without WFxT use counter polling instead. See drivers/timer/arm_arch_timer.c for the implementation.

The default Arm64 `arch_cpu_idle()` implementation continues to use WFI. With the Arm architectural timer, the kernel's next timeout is programmed in the system timer and its interrupt wakes the CPU. Replacing WFI with WFIT while retaining that interrupt does not remove the timer programming or interrupt handling. Avoiding the interrupt would require complex changes to kernel timeout accounting, rescheduling, and SMP deadline coordination, adding significant risk and maintenance cost without a demonstrated benefit.

Devicetree-based MMU Region Mapping

On ARM64 platforms the MMU page tables can be populated automatically from devicetree at compile time. Any node with `compatible = "zephyr,memory-region"` that also carries a `zephyr,memory-attr` property will be turned into a static `arm_mmu_region` entry by the architecture startup code — no per-SoC `mmu_regions.c` changes are required.

Supported memory attributes Only **Normal** memory types are accepted. Device memory (peripheral MMIO) must be mapped through the `DEVICE_MMIO` API (see *Device Driver Model*).

The attribute macros are defined in include/zephyr/dt-bindings/memory-attr/memory-attr-arm64.h as composable combinations of the generic `DT_MEM_CACHEABLE` flag and architecture-specific sub-attributes:

DT macro	Description
DT_MEM_ARM64_MMU_NORMAL	Normal write-back cacheable (DT_MEM_CACHEABLE | ATTR_ARM64_CACHE_WB)
DT_MEM_ARM64_MMU_NORMAL_NC	Normal non-cacheable (0)
DT_MEM_ARM64_MMU_NORMAL_WT	Normal write-through cacheable (DT_MEM_CACHEABLE)

Example devicetree overlay

```
#include <zephyr/dt-bindings/memory-attr/memory-attr.h>
#include <zephyr/dt-bindings/memory-attr/memory-attr-arm64.h>

/ {
    soc {
        /* Cacheable shared memory pool */
        shm0: memory@42000000 {
            compatible = "zephyr,memory-region";
            reg = <0x0 0x42000000 0x0 0x1000>;
            zephyr,memory-region = "SHM0";
            zephyr,memory-attr = <DT_MEM_ARM64_MMU_NORMAL>;
        };

        /* Non-cacheable DMA buffer */
        dma_buf: memory@43000000 {
            compatible = "zephyr,memory-region";
            reg = <0x0 0x43000000 0x0 0x1000>;
            zephyr,memory-region = "DMA_BUF";
            zephyr,memory-attr = <DT_MEM_ARM64_MMU_NORMAL_NC>;
        };
    };
};
```

Each region is mapped with `MT_P_RW_U_NA | MT_DEFAULT_SECURE_STATE` combined with the memory type derived from `zephyr,memory-attr`.

Translation table sizing Every mapped region that falls on a different 2 MB boundary requires an additional Level 3 page table. If the boot hangs silently after adding new regions, increase `CONFIG_MAX_XLAT_TABLES` in the board or test configuration:

```
CONFIG_MAX_XLAT_TABLES=16
```

Supported attribute combinations All non-device attribute combinations are valid: the `DT_MEM_CACHEABLE` generic bit selects cacheable vs non-cacheable, and the arch-specific `ATTR_ARM64_CACHE_WB` sub-bit selects write-back vs write-through when cacheable.

6.1.4 Zephyr support status on RISC-V processors

Overview

This page describes current state of Zephyr for RISC-V processors. Currently, there's support for some boards, as well as Qemu support and support for some FPGA implementations such as neorv32 and litex_vexriscv.

Zephyr support includes PMP, *user mode*, several ISA extensions as well as *semihosting*.

User mode and PMP support

When the platform has Physical Memory Protection (PMP) support, enabling it on Zephyr allows user space support and stack protection to be selected.

ISA extensions

It's possible to set in Zephyr which ISA extensions (RV32/64I(E)MAFD(G)QC) are available on a given platform, by setting the appropriate `CONFIG_RISCV_ISA_*` kconfig. Look at `arch/riscv/Kconfig.isa` for more information.

Note that Zephyr SDK toolchain support may not be defined for all combinations.

Supervisor mode (S-mode)

When `CONFIG_RISCV_S_MODE` is enabled, the Zephyr kernel runs in RISC-V Supervisor mode (S-mode) rather than Machine mode (M-mode). This follows the standard RISC-V privilege architecture used by application-class operating systems and is required for MMU support.

A minimal in-tree M-mode runtime (`arch/riscv/core/sbi.S`) acts as firmware. It performs the initial M-mode to S-mode transition at boot and handles S-mode requests via the RISC-V Supervisor Binary Interface (SBI):

- **Boot sequence**: M-mode configures PMP, `medeleg`, `mideleg`, `mcounteren` and `mtvec`, grants S-mode access to the `seed` CSR through `mseccfg` when the Zkr extension is present, then drops to S-mode via `mret`.
- **Timer**: Machine timer interrupts (MTIP) are forwarded to S-mode as supervisor timer interrupts (STIP). S-mode programs the next timer deadline through the SBI `TIME` extension (`sbi_set_timer`).
- **Ecalls**: S-mode ecalls (exception cause 9) are routed to the M-mode runtime via `medeleg`; all other exceptions and interrupts are delegated to S-mode. See *S-mode ecall constraint* below.

No external firmware is required; the in-tree runtime is self-contained.

Known limitations **PLIC external interrupts**: The PLIC driver always configures the M-mode context for hart 0. In S-mode the correct context is the supervisor context, so PLIC-delivered external interrupts (UART, GPIO, SPI, I2C, etc.) do not reach the CPU. The timer interrupt is unaffected as it is forwarded directly by the M-mode runtime via `sip.STIP`, bypassing the PLIC.

S-mode ecall constraint In M-mode, Zephyr uses the `ecall` instruction as a kernel-internal self-trap: because the kernel owns M-mode unconditionally, `ecall` raises exception cause 11 (M-mode ecall), which `_isr_wrapper` dispatches to whatever kernel service was requested (fatal error, `irq_offload`, context switch).

In S-mode this mechanism is unavailable. `ecall` from S-mode always raises cause 9, and cause 9 is **not** delegated to S-mode (`medeleg` bit 9 = 0) because it must reach the M-mode SBI handler — that is the only path by which S-mode can request M-mode services such as programming the timer via `sbi_set_timer`. The `medeleg` bit offers no per-register-value discrimination: delegating cause 9 to S-mode would break SBI; keeping it in M-mode makes cause 9 unavailable for kernel-internal use.

The practical rule for S-mode ports is:

> `ecall` is reserved for the SBI interface. Any kernel mechanism that uses `ecall` in M-mode must be reimplemented with a direct S-mode path.

Two concrete examples in this port:

- `ARCH_EXCEPT` (`include/zephyr/arch/riscv/error.h`) — instead of issuing `ecall` to trigger a fatal-error entry through `_isr_wrapper`, it calls `z_riscv_fatal_error()` directly.

- `arch_irq_offload` (`arch/riscv/core/irq_offload.c`) — instead of issuing `ecall` to enter `_isr_wrapper`'s `do_irq_offload` path, it simulates ISR context directly: disables interrupts, increments `nested`, calls the routine, decrements `nested`, re-enables interrupts, then calls `z_reschedule_unlocked()` to handle any thread that became ready inside the routine.

Support for using an external SBI implementation such as OpenSBI as a West module is left as future work.

SMP support

SMP is supported on RISC-V for both QEMU-virtualized and hardware-based platforms. In order to test the SMP support, one can use qemu_riscv32 or qemu_riscv64 for QEMU-based platforms, or beaglev_fire or mpfs_icicle for hardware-based platforms.

6.1.5 Semihosting Guide

Overview

Semihosting is a mechanism that enables code running on ARM, RISC-V and Xtensa targets to communicate and use the Input/Output facilities on a host computer that is running a debugger or emulator.

More complete documentation on the available functionality is available at the ARM Github documentation.

The RISC-V functionality borrows from the ARM definitions, as described at the RISC-V Github documentation.

Semihosting implementation on Xtensa supports GDB File-I/O extension, which is explained in GDB File-I/O Remote Protocol.

File Operations

Semihosting enables files on the host computer to be opened, read, and modified by an application. This can be useful when attempting to validate the behaviour of code across datasets that are larger than what can fit into ROM of an emulated platform. File paths can be either absolute, or relative to the directory of the running process.

```
const char *path = "./data.bin";
long file_len, bytes_read, fd;
uint8_t buffer[16];

/* Open the data file for reading */
fd = semihost_open(path, SEMIHOST_OPEN_RB);
if (fd < 0) {
   return -ENOENT;
}
/* Read all data from the file */
file_len = semihost_flen(fd);
while(file_len > 0) {
   bytes_read = semihost_read(fd, buffer, MIN(file_len, sizeof(buffer)));
   if (bytes_read < 0) {
      break;
   }
   /* Process read data */
   do_data_processing(buffer, bytes_read);
   /* Update remaining length */
   file_len -= bytes_read;
}
```

(continues on next page)

(continued from previous page)

```
/* Close the file */
semihost_close(fd);
```

Additional Functionality

Additional functionality is available by running semihosting instructions directly with `semihost_exec()` with one of the instructions defined in `semihost_instr`. For complete documentation on the required arguments and return codes, see the ARM Github documentation.

API Reference

Semihosting APIs

6.1.6 x86 Developer Guide

Overview

This page contains information on certain aspects when developing for x86-based platforms.

Virtual Memory

During very early boot, page tables are loaded so technically the kernel is executing in virtual address space. By default, physical and virtual memory are identity mapped and thus giving the appearance of execution taking place in physical address space. The physical address space is marked by dts `DT_CHOSEN_SRAM_ADDR` and `DT_CHOSEN_SRAM_SIZE` while the virtual address space is marked by `CONFIG_KERNEL_VM_BASE` and `CONFIG_KERNEL_VM_SIZE`. Note that `CONFIG_SRAM_OFFSET` controls where the Zephyr kernel is being placed in the memory, and its counterpart `CONFIG_KERNEL_VM_OFFSET`.

Separate Virtual Address Space from Physical Address Space On 32-bit x86, it is possible to have separate physical and virtual address space. Code and data are linked in virtual address space, but are still loaded in physical memory. However, during boot, code and data must be available and also addressable in physical address space before `vm_enter` inside `arch/x86/core/ia32/crt0.S`. After `vm_enter`, code execution is done via virtual addresses and data can be referred via their virtual addresses. This is possible as the page table generation script (`arch/x86/gen_mmu.py`) identity maps the physical addresses at the page directory level, in addition to mapping virtual addresses to the physical memory. Later in the boot process, the entries for identity mapping at the page directory level are cleared in `z_x86_mmu_init()`, effectively removing the identity mapping of physical memory. This unmapping must be done for userspace isolation or else they would be able to access restricted memory via physical addresses. Since the identity mapping is done at the page directory level, there is no need to allocate additional space for the page table. However, additional space may still be required for additional page directory table.

There are restrictions on where virtual address space can be:

- Physical and virtual address spaces must be disjoint. This is required as the entries in page directory table will be cleared. If they are not disjoint, it would clear the entries needed for virtual addresses.
  - If `CONFIG_X86_PAE` is enabled (=y), each address space must reside in their own 1GB region, due to each entry of PDP (Page Directory Pointer) covers 1GB of memory. For example:
    - Assuming `CONFIG_SRAM_OFFSET` and `CONFIG_KERNEL_VM_OFFSET` are both `0x0`.
    - `DT_CHOSEN_SRAM_ADDR == 0x00000000` and `CONFIG_KERNEL_VM_BASE = 0x40000000` is valid, while

    * DT_CHOSEN_SRAM_ADDR == 0x00000000 and CONFIG_KERNEL_VM_BASE = 0x20000000 is not.
  - If CONFIG_X86_PAE is disabled (=n), each address space must reside in their own 4MB region, due to each entry of PD (Page Directory) covers 4MB of memory.
  - Both DT_CHOSEN_SRAM_ADDR and CONFIG_KERNEL_VM_BASE must also align with the starting addresses of targeted regions.

Specifying Additional Memory Mappings at Build Time

The page table generation script (arch/x86/gen_mmu.py) generates the necessary multi-level page tables for code execution and data access using the kernel image produced by the first linker pass. Additional command line arguments can be passed to the script to generate additional memory mappings. This is useful for static mappings and/or device MMIO access during very early boot. To pass extra command line arguments to the script, populate a CMake list named X86_EXTRA_GEN_MMU_ARGUMENTS in the board configuration file. Here is an example:

```
set(X86_EXTRA_GEN_MMU_ARGUMENTS
    --map 0xA0000000,0x2000
    --map 0x80000000,0x400000,LWUX,0xB0000000)
```

The argument --map takes the following value: <physical address>,<size>[,<flags:LUWX>[,<virtual address>]], where:

- <physical address> is the physical address of the mapping. (Required)
- <size> is the size of the region to be mapped. (Required)
- <flags> is the flag associated with the mapping: (Optional)
  - L: Large page at the page directory level.
  - U: Allow userspace access.
  - W: Read/write.
  - X: Allow execution.
  - D: Cache disabled.
    * Default is small page (4KB), supervisor only, read only, and execution disabled.
- <virtual address is the virtual address of the mapping. (Optional)

Note that specifying additional memory mappings requires larger storage space for the pre-allocated page tables (both kernel and per-domain tables). CONFIG_X86_EXTRA_PAGE_TABLE_PAGES is needed to specify how many more memory pages to be reserved for the page tables. If the needed space is not exactly the same as required space, the gen_mmu.py script will print out a message indicating what needs to be the value for the kconfig.

6.1.7 Xtensa Developer Guide

Overview

This page contains information on certain aspects when developing for Xtensa-based platforms.

HiFi Audio Engine DSP

The kernel allows threads to use the HiFi Audio Engine DSP registers on boards that support these registers. The kernel only supports the use of the HiFi registers by threads and not ISRs.

> **Note**
>
> Presently, only the Intel ADSP ACE hardware platforms are configured for HiFi support by default.

Concepts The kernel can be configured for an application to leverage the services provided by the Xtensa HiFi Audio Engine DSP. Three modes of operation are supported, which are described below.

No HiFi registers mode This mode is used when the application has no threads that use the HiFi registers. It is the kernel's default HiFi services mode.

Unshared HiFi registers mode This mode is used when the application has only a single thread that uses the HiFi registers. The HiFi registers are left unchanged whenever a context switch occurs.

> **Note**
>
> The behavior is undefined, if two or more threads attempt to use the HiFi registers, as the kernel does not attempt to detect (nor prevent) multiple threads from using these registers.

Shared HiFi registers mode This mode is used when the application has two or more threads that use HiFi registers. When enabled, the kernel automatically allows all threads to use the HiFi registers. Conceptually, it can be sub-divided into two sub-modes–eager mode and lazy mode. They will both save and restore the HiFi registers, but they differ in when the registers are saved and restored, as well as to where they are saved and from where they are restored.

In the eager sharing model, the HiFi registers are saved and restored during every thread context switch, regardless of whether the thread used them or not. Additional stack space may be required for each thread to account for the extra registers that must be saved. This is default of the two models.

In the lazy sharing model, the kernel tracks the thread that 'owns' the coprocessor. If the 'owning' thread is switched out, the HiFi registers will not be saved until a new thread attempts to use the HiFi, after which that new thread becomes the new owner and its HiFi registers are loaded.

> **Note**
>
> If an SMP system detects that the owner-to-be is still an owner on another CPU, an IPI will be sent to that CPU to initiate the saving of its HiFi registers to memory. The current processor will then spin until the HiFi registers are saved. This spinning may result in sporadically longer delays. For the best performance, it is recommended that a thread using HiFi be pinned to a single CPU.

Configuration Options The unshared HiFi registers mode is selected when configuration option `CONFIG_XTENSA_HIFI_SHARING` is disabled but configuration options `CONFIG_XTENSA_HIFI3` and/or `CONFIG_XTENSA_HIFI4` are enabled.

The shared HiFi registers mode is selected when the configuration option `CONFIG_XTENSA_HIFI_SHARING` is enabled in addition to configuration options `CONFIG_XTENSA_HIFI3` and/or `CONFIG_XTENSA_HIFI4`. Threads must have sufficient stack space for saving the HiFi register values during context switches as described above.

Both eager and lazy HiFi sharing modes require the configuration option CONFIG_XTENSA_HIFI_SHARING to be enabled. Although eager HiFi sharing is the default, it can be explicitly selected by enabling the configuration option CONFIG_XTENSA_EAGER_HIFI_SHARING. To select lazy HiFi sharing instead, enable the configuration option CONFIG_XTENSA_LAZY_HIFI_SHARING.

6.2 Barriers API

Barrier Services APIs

6.3 Caching

6.3.1 Cache Control Configuration

This is a high-level guide to Zephyr's cache interface and Kconfig options related to cache controllers. See *Cache API* for API reference material.

Zephyr has different Kconfig options to control how the cache controller is implemented and controlled.

- CONFIG_CPU_HAS_DCACHE / CONFIG_CPU_HAS_ICACHE: these hidden options should be selected at SoC / platform level when the CPU actually supports a data or instruction cache. The cache controller can be in the core or can be an external cache controller for which a driver is provided.

  These options have the goal to document an available hardware feature and should be set whether we plan to support and use the cache control in Zephyr or not.

- CONFIG_DCACHE / CONFIG_ICACHE: these options must be selected when support for data or instruction cache is present and working in zephyr. Note that if these options are disabled, caching may still be enabled depending on the hardware defaults.

  All the code paths related to cache control must be conditionally enabled depending on these symbols. When the symbol is set the cache is considered enabled and used.

  These symbols say nothing about the actual API interface exposed to the user. For example a platform using the data cache can enable the CONFIG_DCACHE symbol and use some HAL exported function in some platform-specific code to enable and manage the d-cache.

- CONFIG_CACHE_MANAGEMENT: this option must be selected when the cache operations are exposed to the user through a standard API (see *Cache API*).

  When this option is enabled we assume that all the cache functions are implemented in the architectural code or in an external cache controller driver.

- CONFIG_MEM_ATTR: this option allows the user to specify (using *memory region attributes*) a fixed region in memory that will have caching disabled once the kernel has initialized.

- CONFIG_NOCACHE_MEMORY: this option allows the user to specify individual global variables as uncached using __nocache. This will instruct the linker to place any marked variables into a special nocache region in memory and the MPU driver will configure that region as uncached.

- CONFIG_ARCH_CACHE/CONFIG_EXTERNAL_CACHE: mutually exclusive options for CACHE_TYPE used to define whether the cache operations are implemented at arch level or using an external cache controller with a provided driver.

  - CONFIG_ARCH_CACHE: the cache API is implemented by the arch code
  - CONFIG_EXTERNAL_CACHE: the cache API is implemented by a driver that supports the external cache controller. In this case the driver must be located as usual in the drivers/cache/ directory

Cache API

Cache Interface

6.3.2 Caching Basics

This section discusses the basics of cache coherency and under what situations a user needs to explicitly deal with caching. For more detailed info on Zephyr's caching tools, see *Cache Control Configuration* for Zephyr Kconfig options or *Cache API* for the API reference. This section primarily focuses on the data cache though there is typically also an instruction cache for systems with cache support.

> **Note**
>
> The information here assumes that the architecture-specific MPU support is enabled. See the architecture-specific documentation for details.

> **Note**
>
> While cache coherence can be a concern for data shared between SMP cores, Zephyr in general ensures that memory will be seen in a coherent state from multiple cores. Most applications will only need to use the cache APIs for interaction with external hardware like DMA controllers or foreign CPUs running a different OS image. For more information on cache coherence between SMP cores, see `CONFIG_KERNEL_COHERENCE`.

When dealing with memory shared between a processor core and other bus masters, cache coherency needs to be considered. Typically processor caches exist as close to each processor core as possible to maximize performance gain. Because of this, data moved into and out of memory by DMA engines will be stale in the processor's cache, resulting in what appears to be corrupt data. If you are moving data using DMA and the processor doesn't see the data you expect, cache coherency may be the issue.

There are multiple approaches to ensuring that the data seen by the processor core and peripherals is coherent. The simplest is just to disable caching, but this defeats the purpose of having a hardware cache in the first place and results in a significant performance hit. Many architectures provide methods for disabling caching for only a portion of memory. This can be useful when cache coherence is more important than performance, such as when using DMA with SPI. Finally, there is the option to flush or invalidate the cache for regions of memory at runtime.

Globally Disabling the Data Cache

As mentioned above, globally disabling data caching can have a significant performance impact but can be useful for debugging.

Requirements:

- `CONFIG_DCACHE`: DCACHE control enabled in Zephyr.
- `CONFIG_CACHE_MANAGEMENT`: cache API enabled.
- Call `sys_cache_data_disable()` to globally disable the data cache.

Disabling Caching for a Memory Region

Disabling caching for only a portion of memory can be a good performance compromise if performance on the uncached memory is not critical to the application. This is a good option if the application requires many small unrelated buffers that are smaller than a cache line.

Requirements:

- CONFIG_DCACHE: DCACHE control enabled in Zephyr.
- CONFIG_MEM_ATTR: enable the mem-attr library for handling memory attributes in the device tree.
- Annotate your device tree according to *Memory Attributes*.

Assuming the MPU driver is enabled, it will configure the specified regions according to the memory attributes specified during kernel initialization. When using a dedicated uncached region of memory, the linker needs to be instructed to place buffers into that region. This can be accomplished by specifying the memory region explicitly using Z_GENERIC_SECTION:

```
/* SRAM4 marked as uncached in device tree */
uint8_t buffer[BUF_SIZE] Z_GENERIC_SECTION("SRAM4");
```

Note

Configuring a distinct memory region with separate caching rules requires the use of an MPU region which may be a limited resource on some architectures. MPU regions may be needed by other memory protection features such as *userspace*, *stack protection*, or *memory domains*.

Automatically Disabling Caching by Variable

Zephyr has the ability to automatically define an uncached region in memory and allocate variables to it using __nocache. Any variables marked with this attribute will be placed in a special nocache linker region in memory. This region will be configured as uncached by the MPU driver during initialization. This is a simpler option than explicitly declaring a region of memory uncached but provides less control over the placement of these variables, as the linker may allocate this region anywhere in RAM.

Requirements:

- CONFIG_DCACHE: DCACHE control enabled in Zephyr.
- CONFIG_NOCACHE_MEMORY: enable allocation of the nocache linker region and configure it as uncached.
- Add the __nocache attribute at the end of any uncached buffer definition:

```
uint8_t buffer[BUF_SIZE] __nocache;
```

Note

See note above regarding possible limitations on MPU regions. The nocache region is still a distinct MPU region even though it is automatically created by Zephyr instead of being explicitly defined by the user.

Runtime Cache Control

The most performant but most complex option is to control data caching at runtime. The two most relevant cache operations in this case are **flushing** and **invalidating**. Both of these operations operate on the smallest unit of cacheable memory, the cache line. Data cache lines are typically 16 to 128 bytes. See CONFIG_DCACHE_LINE_SIZE. Cache line sizes are typically fixed in hardware and not configurable, but Zephyr does need to know the size of cache lines in order to correctly and efficiently manage the cache. If the buffers in question are smaller than the data cache line size, it may be more efficient to place them in an uncached region, as unrelated data packed into the same cache line may be destroyed when invalidating.

Flushing the cache involves writing all modified cache lines in a specified region back to shared memory. Flush the cache associated with a buffer after the processor has written to it and before a remote bus master reads from that region.

> **Note**
>
> Some architectures support a cache configuration called **write-through** caching in which data writes from the processor core propagate through to shared memory. While this solves the cache coherence problem for CPU writes, it also results in more traffic to main memory which may result in performance degradation.

Invalidating the cache works similarly but in the other direction. It marks cache lines in the specified region as stale, ensuring that the cache line will be refreshed from main memory when the processor next reads from the specified region. Invalidate the data cache of a buffer that a peripheral has written to before reading from that region.

In some cases, the same buffer may be reused for e.g. DMA reads and DMA writes. In that case it is possible to first flush the cache associated with a buffer and then invalidate it, ensuring that the cache will be refreshed the next time the processor reads from the buffer.

Requirements:

- `CONFIG_DCACHE`: DCACHE control enabled in Zephyr.
- `CONFIG_CACHE_MANAGEMENT`: cache API enabled.
- Call `sys_cache_data_flush_range()` to flush a memory region.
- Call `sys_cache_data_invd_range()` to invalidate a memory region.
- Call `sys_cache_data_flush_and_invd_range()` to flush and invalidate.

Alignment

As mentioned in `sys_cache_data_invd_range()` and associated functions, buffers should be aligned to the cache line size. This can be accomplished by using `__aligned`:

```
uint8_t buffer[BUF_SIZE] __aligned(CONFIG_DCACHE_LINE_SIZE);
```

6.4 Zephyr's device emulators/simulators

6.4.1 Overview

Zephyr includes in its codebase a set of device emulators/simulators. With this we refer to SW components which are built together with the embedded SW and present themselves as devices of a given class to the rest of the system.

These device emulators/simulators can be built for any target which has sufficient RAM and flash, even if some may have extra functionality which is only available in some targets.

> **Note**
>
> Zephyr also includes and uses many other types of simulators/emulators, including CPU and platform simulators, radio simulators, and several build targets which allow running the embedded code in the development host.
>
> Some of Zephyr communication controllers/drivers include also either loopback modes or loopback devices.

This page does not cover any of these.

Note

Drivers which are specific to some platform, like for example the native_sim specific drivers which emulate a peripheral class by connecting to host APIs are not covered by this page.

6.4.2 Available Emulators

ADC emulator

- A fake driver which pretends to be actual ADC, and can be used for testing higher-level API for ADC devices.
- Main Kconfig option: `CONFIG_ADC_EMUL`
- DT binding: `zephyr,adc-emul`

DMA emulator

- Emulated DMA controller
- Main Kconfig option: `CONFIG_DMA_EMUL`
- DT binding: `zephyr,dma-emul`

EEPROM emulator

- Emulate an EEPROM on a flash partition
- Main Kconfig option: `CONFIG_EEPROM_EMULATOR`
- DT binding: `zephyr,emu-eeprom`

EEPROM simulator

- Emulate an EEPROM on RAM
- Main Kconfig option: `CONFIG_EEPROM_SIMULATOR`
- DT binding: `zephyr,sim-eeprom`
- Note: For native targets it is also possible to keep the content as a file on the host filesystem.

External bus and bus connected peripheral emulators

- *Documentation*
- Allow emulating external buses like I2C or SPI and peripherals connected to them.

Flash simulator

- Emulate a flash on RAM
- Main Kconfig option: `CONFIG_FLASH_SIMULATOR`
- DT binding: `zephyr,sim-flash`
- Note: For native targets it is also possible to keep the content as a file on the host filesystem. Check the native_sim flash simulator section.

GPIO emulator

- Emulated GPIO controllers which can be driven from SW
- Main Kconfig option: `CONFIG_GPIO_EMUL`
- DT binding: `zephyr,gpio-emul`

I2C emulator

- Emulated I2C bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_I2C_EMUL`
- DT binding: `zephyr,i2c-emul-controller`

RTC emulator

- Emulated RTC peripheral. See *RTC emulated device section*
- Main Kconfig option: `CONFIG_RTC_EMUL`
- DT binding: `zephyr,rtc-emul`

SPI emulator

- Emulated SPI bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_SPI_EMUL`
- DT binding: `zephyr,spi-emul-controller`

MSPI emulator

- Emulated MSPI bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_MSPI_EMUL`
- DT binding: `zephyr,mspi-emul-controller`

UART emulator

- Emulated UART bus. See *bus emulators*.
- Main Kconfig option: `CONFIG_UART_EMUL`
- DT binding: `zephyr,uart-emul`

6.5 External Bus and Bus Connected Peripherals Emulators

6.5.1 Overview

Zephyr supports a simple emulator framework to support testing of external peripheral drivers without requiring real hardware.

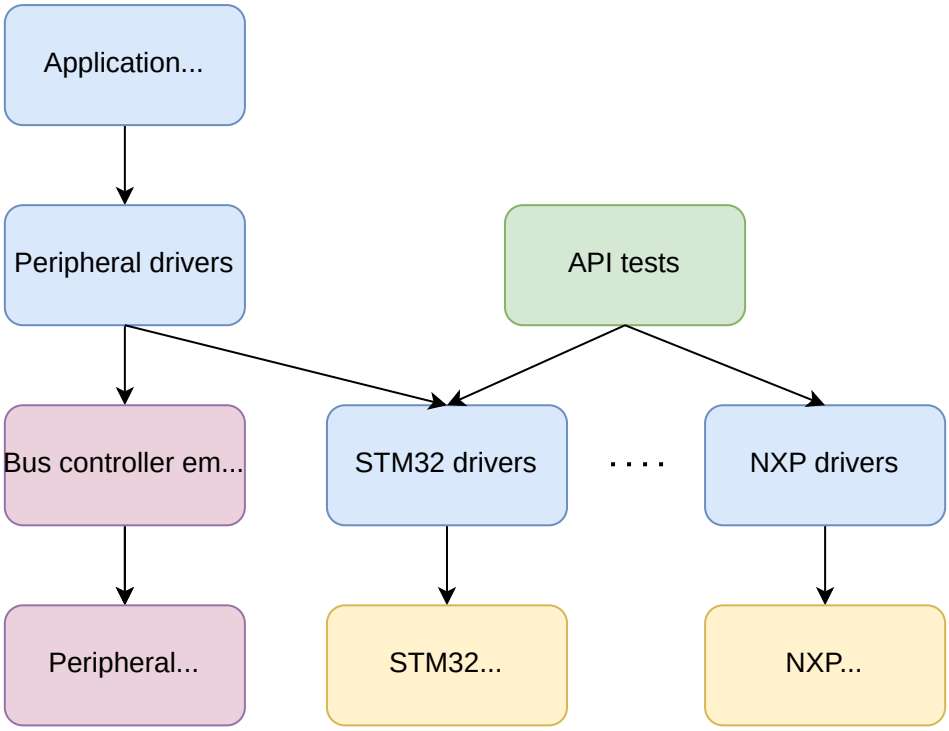
Emulators are used to emulate external hardware devices, to support testing of various subsystems. For example, it is possible to write an emulator for an I2C compass such that it appears on the I2C bus and can be used just like a real hardware device.

Emulators often implement special features for testing. For example a compass may support returning bogus data if the I2C bus speed is too high, or may return invalid measurements if calibration has not yet been completed. This allows for testing that high-level code can handle these situations correctly. Test coverage can therefore approach 100% if all failure conditions are emulated.

6.5.2 Concept

The diagram below shows application code / high-level tests at the top. This is the ultimate application we want to run.

Below that are peripheral drivers, such as the AT24 EEPROM driver. We can test peripheral drivers using a peripheral emulator connected via a bus controller emulator which passes e.g. I2C traffic from the AT24 driver (peripheral driver) to the AT24 simulator (the peripheral emulator).



Text is not SVG - cannot display

Separately we can test the STM32 and NXP I2C drivers on real hardware using API tests. These require some sort of peripheral device attached to the bus, but with this, we can validate much of the driver functionality.

Putting the two together, we can test the application and peripheral code entirely on native_sim. Since we know that the I2C driver on the real hardware works, we should expect the application and peripheral drivers to work on the real hardware also.

Using the above framework we can test an entire application (e.g. Embedded Controller) on native_sim using emulators for all non-chip drivers.

With this approach we can:

- Write individual tests for each driver (green), covering all failure modes, error conditions, etc.
- Ensure 100% test coverage for drivers (green)
- Write tests for combinations of drivers, such as GPIOs provided by an I2C GPIO expander driver talking over an I2C bus, with the GPIOs controlling a charger. All of this can work in the emulated environment or on real hardware.
- Write a complex application that ties together all of these pieces and runs on native_sim. We can develop on a host, use source-level debugging, etc.
- Transfer the application to any board which provides the required features (e.g. I2C, enough GPIOs), by adding Kconfig and devicetree fragments.

6.5.3 Creating a Device Driver Emulator

The emulator subsystem is modeled on the *Device Driver Model*. You create an emulator instance using one of the `EMUL_DT_DEFINE()` or `EMUL_DT_INST_DEFINE()` APIs.

Emulators for peripheral devices reuse the same devicetree node as the real device driver. This means that your emulator defines `DT_DRV_COMPAT` using the same `compat` value from the real driver.

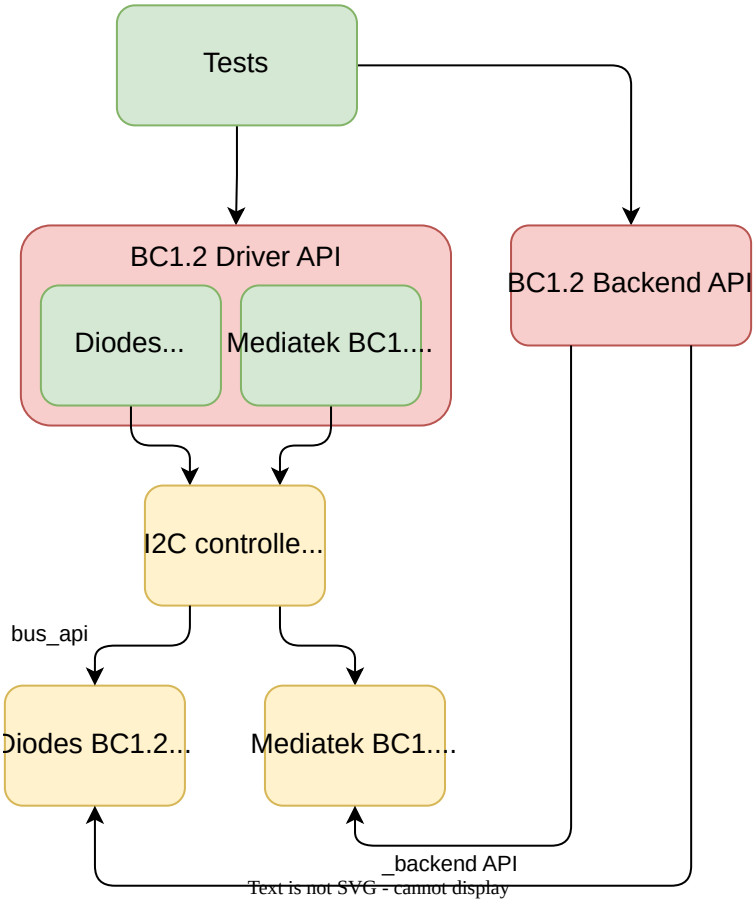
```
/* From drivers/sensor/bm160/bm160.c */
#define DT_DRV_COMPAT bosch_bmi160

/* From drivers/sensor/bmi160/emul_bmi160.c */
#define DT_DRV_COMPAT bosch_bmi160
```

The `EMUL_DT_DEFINE()` function accepts two API types:

1. `bus_api` - This points to the API for the upstream bus that the emulator connects to. The `bus_api` parameter is required. The supported emulated bus types include I2C, SPI, eSPI, and MSPI.
2. `_backend_api` - This points to the device-class specific backend API for the emulator. The `_backend_api` parameter is optional.

The diagram below demonstrates the logical organization of the `bus_api` and `_backend_api` using the BC1.2 charging detector driver as the model device-class.



The real code is shown in green, while the emulator code is shown in yellow.

The `bus_api` connects the BC1.2 emulators to the `native_sim` I2C controller. The real BC1.2 drivers are unchanged and operate exactly as if there was a physical I2C controller present in the system. The `native_sim` I2C controller uses the `bus_api` to initiate register reads and writes to the emulator.

The `_backend_api` provides a mechanism for tests to manipulate the emulator out of band. Each device class defines it's own API functions. The backend API functions focus on high-level behavior and do not provide hooks for specific emulators.

In the case of the BC1.2 charging detector the backend API provides functions to simulate connecting and disconnecting a charger to the emulated BC1.2 device. Each emulator is responsible for updating the correct vendor specific registers and potentially signalling an interrupt.

Example test flow:

1. Test registers BC1.2 detection callback using the Zephyr BC1.2 driver API.
2. Test connects a charger using the BC1.2 emulator backend.
3. Test verifies B1.2 detection callback invoked with correct charger type.
4. Test disconnects a charger using the BC1.2 emulator backend.

With this architecture, the same test can be used will all supported drivers in the same driver class.

6.5.4 Available Emulators

Zephyr includes the following emulators:

- I2C emulator driver, allowing drivers to be connected to an emulator so that tests can be performed without access to the real hardware
- SPI emulator driver, which does the same for SPI
- eSPI emulator driver, which does the same for eSPI. The emulator is being developed to support more functionalities.
- MSPI emulator driver, allowing drivers to be connected to an emulator so that tests can be performed without access to the real hardware.

I2C Emulation features

In the binding of the I2C emulated bus, there's a custom property for address based forwarding. Given the following devicetree node:

```
i2c0: i2c@100 {
  status = "okay";
  compatible = "zephyr,i2c-emul-controller";
  clock-frequency = <I2C_BITRATE_STANDARD>;
  #address-cells = <1>;
  #size-cells = <0>;
  #forward-cells = <1>;
  reg = <0x100 4>;
  forwards = <&i2c1 0x20>;
};
```

The final property, `forwards` indicates that any read/write requests sent to address `0x20` should be sent to `i2c1` with the same address. This allows us to test both the controller and the target end of the communication on the same image.

> **Note**
>
> The `#forward-cells` attribute should always be 1. Each entry in the `forwards` attribute consists of the phandle followed by the address. In the example above, `<&i2c1 0x20>` will forward all read/write operations made to `i2c0` at port `0x20` to `i2c1` on the same port. Since no additional cells are used by the emulated controller, the number of cells should remain 1.

6.5.5 Samples

Here are some examples present in Zephyr:

1. Bosch BMI160 sensor driver connected via both I2C and SPI to an emulator:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/sensor/bmi160
```

2. The same test can be built with a second EEPROM which is an Atmel AT24 EEPROM driver connected via I2C an emulator:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/eeprom/api -- -
↪DDTC_OVERLAY_FILE=at2x_emul.overlay -DEXTRA_CONF_FILE=at2x_emul.conf
```

6.5.6 API Reference

Emulator interfaces

6.6 Firmware

6.6.1 Overview

This class of drivers provides support for interacting with various service-providing (e.g. clock management, power management, pinctrl, etc.) firmware. The list of firmware currently supported by Zephyr is summarized below:

Name	Vendor	Description	Documentation
SCMI	ARM	System Control and Management Interface	Link
TISCI	TI	TI System Controller Interface	Link
QEMU FWCFG	QEMU	QEMU Firmware Configuration	Link
RPI FIRMWARE	Raspberry Pi	Raspberry Pi VideoCore Firmware Interface	Link

6.6.2 API Reference

Usually, the firmware drivers are meant to provide functionality to other classes of drivers. Furthermore, the purpose and functionality may differ from one firmware to another. Consequently, this class of drivers doesn't implement a standard API meant to be used by the end user.

6.6.3 Samples

See firmware.

6.6.4 Resources

ARM System Control and Management Interface

Overview System Control and Management Interface (SCMI) is a specification developed by ARM, which describes a set of OS-agnostic software interfaces used to perform system management (e.g: clock control, pinctrl, etc...). In this context, Zephyr acts as an SCMI agent.

> **Note**
>
> The Zephyr implementation may only include a subset of the features or functionalities mentioned throughout this document.

Standard protocols The set of supported **standard**[1] protocols is summarized below:

ID	Name	Supported version
0x10	Base protocol	2.1
0x11	Power domain management protocol	3.1
0x12	System power management protocol	2.1
0x14	Clock management protocol	3.0
0x19	Pin Control protocol	1.0

Transports The set of supported transports is summarized below:

Name	Description	Compatible
MBOX	Shared memory with mailbox-based doorbell	`arm,scmi`
SMC	Shared memory with SMC-based doorbell	`arm,scmi-smc`

Vendor extensions The SCMI specification allows vendors to introduce additional protocols and provides a degree of freedom with respect to how the platform firmware can behave in certain situations. In the context of this documentation, these are collectively known as **vendor extensions**.

NXP NXP provides an SCMI-compliant platform firmware known as **System Manager (SM)**. You may find its documentation here.

The set of supported NXP-specific protocols is summarized below:

ID	Name	Supported version
0x82	CPU	1.0

The set of supported NXP-specific quirks is summarized below:

1. Depending on how the platform firmware is configured, the shared memory may contain a message CRC field that's checked by the firmware on each received message. See `CONFIG_ARM_SCMI_NXP_VENDOR_EXTENSIONS`.

6.7 Peripherals

Zephyr supports a wide range of hardware peripherals through a set of device driver APIs. They are grouped into the categories below; each card lists the peripherals in that category, or follow the category title for more detail.

Analog

6.7.1 Analog

Analog-to-digital and digital-to-analog converters, comparators, and amplifiers.

Analog-to-Digital Converter (ADC)

Overview

[1] Meaning protocols covered by the SCMI specification.

API Reference

ADC

Related code samples

- ADC power managementUse ADC in a low-power context on STM32.
- Analog-to-Digital Converter (ADC) sequence sampleRead analog inputs from ADC channels, using a sequence.
- Analog-to-Digital Converter (ADC) with devicetreeRead analog inputs from ADC channels.
- AutAnalog ADC FIR+FIFORead AutAnalog ADC channels with FIR filtering and FIFO bulk collection.
- FRDM-MCXA156 lpdac opamp lpadc exampleAmplify the input of the OPAMP and sample it using the ADC.
- Generic ADC streamGet data from a ADC using stream.
- Power Delivery AnalyzerImplement a basic Power Delivery Analyzer to determine if a USB device is currently charging.

Digital-to-Analog Converter (DAC)

Overview The DAC API provides access to Digital-to-Analog Converter (DAC) devices.

Configuration Options Related configuration options:

- `CONFIG_DAC`

API Reference

DAC

Related code samples

- Digital-to-Analog Converter (DAC)Generate an analog sawtooth signal using the DAC driver API.

Comparator

Overview An analog comparator compares the voltages of two analog signals connected to its negative and positive inputs. If the voltage at the positive input is higher than the negative input, the comparator's output will be high, otherwise, it will be low.

Comparators can typically set a trigger which triggers on output changes. This trigger can either invoke a callback, or its status can be polled.

Related configuration options:

- `CONFIG_COMPARATOR`

Configuration Embedded comparators can typically be configured at runtime. When enabled, an initial configuration must be provided using the devicetree. At runtime, comparators can have their configuration updated using device driver specific APIs. The configuration will be applied when the comparator is resumed.

Power management Comparators are enabled using power management. When resumed, the comparator will actively compare its inputs, producing an output and detecting edges. When suspended, the comparator will be inactive.

Comparator shell The comparator shell provides the `comp` command with a set of subcommands for the *shell* module.

The `comp` shell command provides the following subcommands:

- `get_output` See `comparator_get_output()`
- `set_trigger` See `comparator_set_trigger()`
- `await_trigger` Awaits trigger using the following flow:
  - Set trigger callback using `comparator_set_trigger_callback()`
  - Await callback or time out after default or optionally provided timeout
  - Clear trigger callback using `comparator_set_trigger_callback()`
- `trigger_is_pending` See `comparator_trigger_is_pending()`

Related configuration options:

- `CONFIG_SHELL`
- `CONFIG_COMPARATOR_SHELL`
- `CONFIG_COMPARATOR_SHELL_AWAIT_TRIGGER_DEFAULT_TIMEOUT`
- `CONFIG_COMPARATOR_SHELL_AWAIT_TRIGGER_MAX_TIMEOUT`

Note

The power management shell can optionally be enabled alongside the comparator shell.

Related configuration options:

- `CONFIG_PM_DEVICE`
- `CONFIG_PM_DEVICE_SHELL`

API Reference

Comparator

Fake comparator

Operational Amplifier (OPAMP)

Overview The operational amplifier is an analog device that amplifies differential input signals (difference between inverting and non-inverting input) to give a resulting output voltage.

Configuration When OPAMP is enabled, an initial configuration shall be provided using the devicetree. The OPAMP gain can be adjusted at runtime.

Related configuration options:

- `CONFIG_OPAMP`

API Reference
OPAMP

Related code samples

- Operational Amplifier (OPAMP) output measure sampleAmplify the input of the OPAMP and sample it using the ADC.

Audio

6.7.2 Audio

Audio Codec

Overview The Audio Codec API provides access to digital audio codecs.

Configuration Options Related configuration options:

- `CONFIG_AUDIO_CODEC`

API Reference
Audio Codec Interface

Digital Microphone (DMIC)

Overview The audio DMIC interface provides access to digital microphones.

Digital microphones typically output a PDM (Pulse Density Modulation) bit stream instead of analog audio. PDM uses a 1-bit, high-rate signal in which audio amplitude is represented by the density of 1s and 0s over time.

Because applications usually consume PCM (Pulse Code Modulation) samples rather than raw PDM data, a DMIC controller converts the microphone stream into PCM audio. This conversion includes filtering and decimation: filtering removes the high-frequency noise shaped into the PDM stream, and decimation reduces the bit-stream rate to the desired PCM sample rate.

From a Zephyr point of view, a DMIC device is an audio capture peripheral. The application configures the microphone and controller, starts capture, and reads PCM buffers from the driver.

Key concepts The DMIC API separates configuration into three pieces that mirror the capture path:

1. How the microphone is driven on the PDM side,
2. How incoming PDM channels are arranged, and
3. How converted PCM samples are delivered to the application.

These pieces are combined in `dmic_cfg` and passed to `dmic_configure()`.

PDM I/O configuration (`dmic_cfg.io`)
This describes the electrical and timing requirements of the microphone interface, such as the supported PDM clock frequency range, duty cycle, and any controller-specific signal polarity settings.

Channel configuration (`dmic_cfg.channel`)
This tells the driver which physical PDM controller and left or right microphone lane should appear as each logical audio channel in the PCM output. It also declares how many channels and streams the application wants to use.

PCM stream configuration (dmic_cfg.streams)
This defines the PCM data delivered by the driver, including sample rate, sample width, block size, and the k_mem_slab used to allocate receive buffers for each enabled stream.

Typical application flow Typical use of the DMIC API is:

1. Get the DMIC device, usually from devicetree.
2. Fill a dmic_cfg structure with I/O, channel, and stream settings. See the *Buffering* section below for more details on how to define the memory buffer the DMIC uses to store received PCM data.
3. Call dmic_configure(), passing the configuration structure.
4. Start capture with dmic_trigger() using DMIC_TRIGGER_START.
5. Fetch PCM data with dmic_read().
6. Stop, pause, or reset capture with additional trigger commands as needed.

Buffering Received PCM data is returned through buffers owned by the driver. Applications provide the backing memory through a k_mem_slab referenced by each configured PCM stream.

A common pattern is to declare this slab statically:

Listing 1: Declaring a memory slab statically for PCM receive buffers

```
K_MEM_SLAB_DEFINE_STATIC(mem_slab,
                         SAMPLES_PER_BUFFER * sizeof(int16_t),
                         BUFFER_COUNT,
                         sizeof(void *));
```

In this example, each slab block stores one PCM buffer and BUFFER_COUNT sets how many buffers can be queued internally before the application reads them using dmic_read().

Shell commands When CONFIG_AUDIO_DMIC_SHELL is enabled, a set of dmic commands is available. They allow capturing audio from a DMIC device interactively without having to write a dedicated application.

Each subcommand takes the DMIC device name as its first argument, optionally followed by the audio capture parameters (sample rate, number of channels, and PCM sample width).

The following subcommands are available:

dmic read <device> [<count> [<rate_hz> [<channels> [<pcm_width>]]]]
Capture count audio blocks and print the peak level per channel. The default count is 5; each block is 50 ms of audio.

dmic vu <device> [<rate_hz> [<channels> [<pcm_width>]]]
Display a live, color-coded level meter with peak-hold and clip indication, one bar per channel. Runs until any key is pressed.

dmic dump <device> [<seconds> [<rate_hz> [<channels> [<pcm_width>]]]]
Capture seconds of audio and print it as base64-encoded raw PCM, together with the host commands needed to decode and play it back. The default duration is 2 seconds.

The built-in help (e.g. dmic read --help) lists the arguments and their defaults.

Configuration Options Related configuration options:

- CONFIG_AUDIO_DMIC
- CONFIG_AUDIO_DMIC_SHELL

API Reference
Digital Microphone Interface

Related code samples

- Digital Microphone (DMIC)Perform PDM transfers using different configurations.
- X-NUCLEO-IKS02A1 shield - MEMS microphoneAcquire audio using the digital MEMS microphone on X-NUCLEO-IKS02A1 shield.

Inter-IC Sound (I2S) Bus

Overview The I2S (Inter-IC Sound) API provides support for the standard I2S interface as well as common non-standard extensions such as PCM Short/Long Frame Sync and Left/Right Justified Data Formats.

Shell When `CONFIG_I2S_SHELL` is enabled, an `i2s` shell command is available to stream a generated sine test tone over any I2S controller without writing application code. It is transport only; pair it with the `codec` shell to route the audio through a codec.

The tone commands are grouped under `i2s tone`:

- `i2s tone start <device> [frequency_hz] [sample_rate] [bits]` starts a stereo test tone on the given I2S device. `frequency_hz` defaults to 440 Hz, `sample_rate` defaults to 48000 Hz, and `bits` defaults to 16 (valid values are 16, 24 and 32).
- `i2s tone stop` stops the running tone.
- `i2s tone info` shows the current tone stream status.

For example, to stream a 1 kHz, 48 kHz, 16-bit tone:

```
uart:~$ i2s tone start i2s@0 1000 48000 16
Streaming 1000 Hz tone on i2s@0 @ 48000 Hz, 16-bit stereo
uart:~$ i2s tone info
state    : streaming
i2s dev  : i2s@0
tone     : 1000 Hz
rate     : 48000 Hz
format   : 16-bit stereo, 64-frame blocks x 8
uart:~$ i2s tone stop
Stopped
```

The TX buffering used by the shell is controlled by `CONFIG_I2S_SHELL_BLOCK_FRAMES` and `CONFIG_I2S_SHELL_BLOCK_COUNT`.

Configuration Options Related configuration options:

- `CONFIG_I2S`
- `CONFIG_I2S_SHELL`

API Reference
I2S

Related code samples

- I2S codecProcess an audio stream to codec.
- I2S echoProcess an audio stream to add an echo effect.

- I2S outputSend I2S output stream
- USB Audio asynchronous explicit feedback sampleUSB Audio 2 explicit feedback sample playing audio on I2S.
- USB Audio asynchronous implicit feedback sampleUSB Audio 2 implicit feedback sample playing stereo and recording mono audio on I2S interface.

Digital Audio Interface (DAI)

Overview The DAI (Digital Audio Interface) is a generic high level API for audio drivers. It can be configured with bespoke data for vendor specific configuration.

Configuration Options Related configuration options:

- `CONFIG_DAI`

API Reference
DAI

Communication

6.7.3 Communication

Serial buses (I2C, SPI, UART, CAN) and other communication interfaces.

Controller Area Network (CAN)

CAN Controller

- *Overview*
- *Sending*
- *Receiving*
- *Setting the bitrate*
- *SocketCAN*
- *Samples*
- *CAN Controller API Reference*

Overview Controller Area Network is a two-wire serial bus specified by the Bosch CAN Specification, Bosch CAN with Flexible Data-Rate specification and the ISO 11898-1:2003 standard. CAN is mostly known for its application in the automotive domain. However, it is also used in home and industrial automation and other products.

Warning

CAN controllers can only initialize when the bus is in the idle (recessive) state for at least 11 recessive bits. Therefore you have to make sure that CAN RX is high, at least for a short time. This is also necessary for loopback mode.

The bit-timing as defined in ISO 11898-1:2003 looks as following:

Figure 2: CAN Timing. ©Alexander Wachter

A single bit is split into four segments.

- Sync_Seg: The nodes synchronize at the edge of the Sync_Seg. It is always one time quantum in length.
- Prop_Seg: The signal propagation delay of the bus and other delays of the transceiver and node.
- Phase_Seg1 and Phase_Seg2 :Define the sampling point. The bit is sampled at the end of Phase_Seg1.

The bit-rate is calculated from the time of a time quantum and the values defined above. A bit has the length of Sync_Seg plus Prop_Seg plus Phase_Seg1 plus Phase_Seg2 multiplied by the time of single time quantum. The bit-rate is the inverse of the length of a single bit.

A bit is sampled at the sampling point. The sample point is between Phase_Seg1 and PhaseSeg2 and therefore is a parameter that the user needs to choose. The CiA recommends setting the sample point to 87.5% of the bit.

The resynchronization jump width (SJW) defines the amount of time quantum the sample point can be moved. The sample point is moved when resynchronization is needed.

The timing parameters (SJW, bitrate and sampling point, or bitrate, Prop_Seg, Phase_Seg1and Phase_Seg2) are initially set from the device-tree and can be changed at run-time from the timing-API.

CAN uses so-called identifiers to identify the frame instead of addresses to identify a node. This identifier can either have 11-bit width (Standard or Basic Frame) or 29-bit in case of an Extended Frame. The Zephyr CAN API supports both Standard and Extended identifiers concurrently. A CAN frame starts with a dominant Start Of Frame bit. After that, the identifiers follow. This phase is called the arbitration phase. During the arbitration phase, write collisions are allowed. They resolve by the fact that dominant bits override recessive bits. Nodes monitor the bus and notice when their transmission is being overridden and in case, abort their transmission. This effectively gives lower number identifiers priority over higher number identifiers.

Filters are used to allowlist identifiers that are of interest for the specific node. An identifier that doesn't match any filter is ignored. Filters can either match exactly or a specified part of the identifier. This method is called masking. As an example, a mask with 11 bits set for standard or 29 bits set for extended identifiers must match perfectly. Bits that are set to zero in the mask are ignored when matching an identifier. Most CAN controllers implement a limited number of filters in hardware. The number of filters is also limited in Kconfig to save memory.

Errors may occur during transmission. In case a node detects an erroneous frame, it partially overrides the current frame with an error-frame. Error-frames can either be error passive or error active, depending on the state of the controller. In case the controller is in error active state, it sends six consecutive dominant bits, which is a violation of the stuffing rule that all nodes can detect. The sender may resend the frame right after.

An initialized node can be in one of the following states:

- Error-active
- Error-passive
- Bus-off

After initialization, the node is in the error-active state. In this state, the node is allowed to send active error frames, ACK, and overload frames. Every node has a receive- and transmit-error counter. If either the receive- or the transmit-error counter exceeds 127, the node changes to error-passive state. In this state, the node is not allowed to send error-active frames anymore. If the transmit-error counter increases further to 255, the node changes to the bus-off state. In

this state, the node is not allowed to send any dominant bits to the bus. Nodes in the bus-off state may recover after receiving 128 occurrences of 11 concurrent recessive bits.

You can read more about CAN bus in this CAN Wikipedia article.

Zephyr supports following CAN features:

- Standard and Extended Identifiers
- Filters with Masking
- Loopback and Silent mode
- Remote Request

Sending The following code snippets show how to send data.

This basic sample sends a CAN frame with standard identifier 0x123 and eight bytes of data. When passing NULL as the callback, as shown in this example, the send function blocks until the frame is sent and acknowledged by at least one other node or an error occurred. The timeout only takes effect on acquiring a mailbox. When a transmitting mailbox is assigned, sending cannot be canceled.

```
struct can_frame frame = {
        .flags = 0,
        .id = 0x123,
        .dlc = 8,
        .data = {1,2,3,4,5,6,7,8}
};
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));
int ret;

ret = can_send(can_dev, &frame, K_MSEC(100), NULL, NULL);
if (ret != 0) {
        LOG_ERR("Sending failed [%d]", ret);
}
```

This example shows how to send a frame with extended identifier 0x1234567 and two bytes of data. The provided callback is called when the message is sent, or an error occurred. Passing K_FOREVER to the timeout causes the function to block until a transfer mailbox is assigned to the frame or an error occurred. It does not block until the message is sent like the example above.

```
void tx_callback(const struct device *dev, int error, void *user_data)
{
        char *sender = (char *)user_data;

        if (error != 0) {
                LOG_ERR("Sending failed [%d]\nSender: %s\n", error, sender);
        }
}

int send_function(const struct device *can_dev)
{
        struct can_frame frame = {
                .flags = CAN_FRAME_IDE,
                .id = 0x1234567,
                .dlc = 2
        };

        frame.data[0] = 1;
        frame.data[1] = 2;
```

(continues on next page)

(continued from previous page)

```
        return can_send(can_dev, &frame, K_FOREVER, tx_callback, "Sender 1");
}
```

Receiving Frames are only received when they match a filter. The following code snippets show how to receive frames by adding filters.

Here we have an example for a receiving callback as used for `can_add_rx_filter()`. The user data argument is passed when the filter is added.

```
void rx_callback_function(const struct device *dev, struct can_frame *frame, void␣
↪*user_data)
{
        ... do something with the frame ...
}
```

The following snippet shows how to add a filter with a callback function. It is the most efficient but also the most critical way to receive messages. The callback function is called from an interrupt context, which means that the callback function should be as short as possible and must not block. Adding callback functions is not allowed from userspace context.

The filter for this example is configured to match the identifier 0x123 exactly.

```
const struct can_filter my_filter = {
        .flags = 0U,
        .id = 0x123,
        .mask = CAN_STD_ID_MASK
};
int filter_id;
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));

filter_id = can_add_rx_filter(can_dev, rx_callback_function, callback_arg, &
↪my_filter);
if (filter_id < 0) {
  LOG_ERR("Unable to add rx filter [%d]", filter_id);
}
```

Here an example for `can_add_rx_filter_msgq()` is shown. With this function, it is possible to receive frames synchronously. This function can be called from userspace context. The size of the message queue should be as big as the expected backlog.

The filter for this example is configured to match the extended identifier 0x1234567 exactly.

```
const struct can_filter my_filter = {
        .flags = CAN_FILTER_IDE,
        .id = 0x1234567,
        .mask = CAN_EXT_ID_MASK
};
CAN_MSGQ_DEFINE(my_can_msgq, 2);
struct can_frame rx_frame;
int filter_id;
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));

filter_id = can_add_rx_filter_msgq(can_dev, &my_can_msgq, &my_filter);
if (filter_id < 0) {
  LOG_ERR("Unable to add rx msgq [%d]", filter_id);
  return;
```

(continues on next page)

(continued from previous page)

```
}

while (true) {
  k_msgq_get(&my_can_msgq, &rx_frame, K_FOREVER);
  ... do something with the frame ...
}
```

`can_remove_rx_filter()` removes the given filter.

```
can_remove_rx_filter(can_dev, filter_id);
```

Setting the bitrate The bitrate and sampling point is initially set at runtime. To change it from the application, one can use the `can_set_timing()` API. The `can_calc_timing()` function can calculate timing from a bitrate and sampling point in permille. The following example sets the bitrate to 250k baud with the sampling point at 87.5%.

```
struct can_timing timing;
const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));
int ret;

ret = can_calc_timing(can_dev, &timing, 250000, 875);
if (ret > 0) {
  LOG_INF("Sample-Point error: %d", ret);
}

if (ret < 0) {
  LOG_ERR("Failed to calc a valid timing");
  return;
}

ret = can_stop(can_dev);
if (ret != 0) {
  LOG_ERR("Failed to stop CAN controller");
}

ret = can_set_timing(can_dev, &timing);
if (ret != 0) {
  LOG_ERR("Failed to set timing");
}

ret = can_start(can_dev);
if (ret != 0) {
  LOG_ERR("Failed to start CAN controller");
}
```

A similar API exists for calculating and setting the timing for the data phase for CAN FD capable controllers. See `can_set_timing_data()` and `can_calc_timing_data()`.

SocketCAN Zephyr additionally supports SocketCAN, a BSD socket implementation of the Zephyr CAN API. SocketCAN brings the convenience of the well-known BSD Socket API to Controller Area Networks. It is compatible with the Linux SocketCAN implementation, where many other high-level CAN projects build on top. Note that frames are routed to the network stack instead of passed directly, which adds some computation and memory overhead.

Samples We have two ready-to-build samples demonstrating use of the Zephyr CAN API: Zephyr CAN counter sample and SocketCAN sample.

CAN Controller API Reference

CAN Controller

Related code samples

- Controller Area Network (CAN) babbling nodeSimulate a babbling CAN node.
- Controller Area Network (CAN) counterSend and receive CAN messages.

Fake CAN controller

CAN Transceiver

- *Overview*
- *CAN Transceiver API Reference*

Overview A CAN transceiver is an external device that converts the logic level signals from the CAN controller to the bus-levels. The bus lines are called CAN High (CAN H) and CAN Low (CAN L). The transmit wire from the controller to the transceiver is called CAN TX, and the receive wire is called CAN RX. These wires use the logic levels whereas the bus-level is interpreted differentially between CAN H and CAN L. The bus can be either in the recessive (logical one) or dominant (logical zero) state. The recessive state is when both lines, CAN H and CAN L, are roughly at the same voltage level. This state is also the idle state. To write a dominant bit to the bus, open-drain transistors tie CAN H to Vdd and CAN L to ground. The first and last node use a 120-ohm resistor between CAN H and CAN L to terminate the bus. The dominant state always overrides the recessive state. This structure is called a wired-AND.

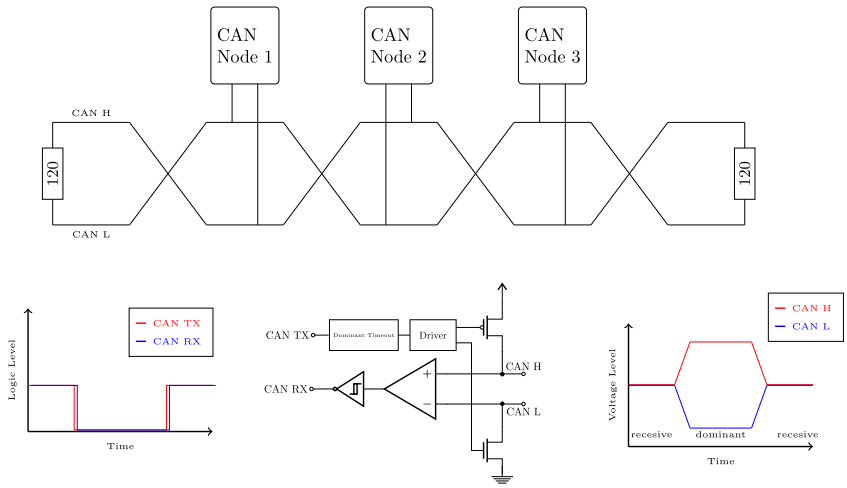


Figure 1: CAN Transceiver, from Logic Levels to Bus Levels. ©Alexander Wachter

CAN Transceiver API Reference

CAN Transceiver

CAN Shell

- *Overview*

- *Inspection*
- *Configuration*
- *Receiving*
- *Sending*
- *Bus Recovery*

Overview The CAN shell provides a `can` command with a set of subcommands for the *shell* module. It allows for testing and exploring the *CAN Controller* driver API through an interactive interface without having to write a dedicated application. The CAN shell can also be enabled in existing applications to aid in interactive debugging of CAN issues.

The CAN shell provides access to most CAN controller features, including inspection, configuration, sending and receiving of CAN frames, and bus recovery.

In order to enable the CAN shell, the following *Kconfig* options must be enabled:

- `CONFIG_SHELL`
- `CONFIG_CAN`
- `CONFIG_CAN_SHELL`

The following *Kconfig* options enable additional subcommands and features of the `can` command:

- `CONFIG_CAN_FD_MODE` enables CAN FD specific subcommands (e.g. for setting the timing for the CAN FD data phase).
- `CONFIG_CAN_RX_TIMESTAMP` enables printing of timestamps for received CAN frames.
- `CONFIG_CAN_STATS` enables printing of various statistics for the CAN controller in the `can show` subcommand. This depends on `CONFIG_STATS` being enabled as well.
- `CONFIG_CAN_MANUAL_RECOVERY_MODE` enables the `can recover` subcommand.

For example, building the hello_world sample for the frdm_k64f with the CAN shell and CAN statistics enabled:

```
# From the root of the zephyr repository
west build -b frdm_k64f samples/hello_world -- -DCONFIG_SHELL=y -DCONFIG_CAN=y -
↪DCONFIG_CAN_SHELL=y -DCONFIG_STATS=y -DCONFIG_CAN_STATS=y
```

See the *shell* documentation for general instructions on how to connect and interact with the shell. The CAN shell comes with built-in help (unless `CONFIG_SHELL_HELP` is disabled). The built-in help messages can be printed by passing `-h` or `--help` to the `can` command or any of its subcommands. All subcommands also support tab-completion of their arguments.

Tip

All of the CAN shell subcommands take the name of a CAN controller as their first argument, which also supports tab-completion. A list of all devices available can be obtained using the `device list` shell command when `CONFIG_DEVICE_SHELL` is enabled. The examples below all use the device name `can@0`.

Inspection The properties of a given CAN controller can be inspected using the `can show` subcommand as shown below. The properties include the core CAN clock rate, the maximum supported bitrate, the number of RX filters supported, capabilities, current mode, current state, error counters, timing limits, and more:

```
uart:~$ can show can@0
core clock:      144000000 Hz
max bitrate:     5000000 bps
max std filters: 15
max ext filters: 15
capabilities:    normal loopback listen-only fd
mode:            normal
state:           stopped
rx errors:       0
tx errors:       0
timing:          sjw 1..128, prop_seg 0..0, phase_seg1 2..256, phase_seg2 2..128,␣
↪prescaler 1..512
timing data:     sjw 1..16, prop_seg 0..0, phase_seg1 1..32, phase_seg2 1..16,␣
↪prescaler 1..32
transceiver:     passive/none
statistics:
  bit errors:    0
    bit0 errors: 0
    bit1 errors: 0
  stuff errors:  0
  crc errors:    0
  form errors:   0
  ack errors:    0
  rx overruns:   0
```

Note

The statistics are only printed if `CONFIG_CAN_STATS` is enabled.

Configuration The CAN shell allows for configuring the CAN controller mode and timing, along with starting and stopping the processing of CAN frames.

Note

The CAN controller mode and timing can only be changed while the CAN controller is stopped, which is the initial setting upon boot-up. The initial CAN controller mode is set to `normal` and the initial timing is set according to the `bitrate`, `sample-point`, `bitrate-data`, and `sample-point-data` *Devicetree* properties.

Timing The classic CAN bitrate/CAN FD arbitration phase bitrate can be configured using the `can bitrate` subcommand as shown below. The bitrate is specified in bits per second.

```
uart:~$ can bitrate can@0 125000
setting bitrate to 125000 bps
```

If `CONFIG_CAN_FD_MODE` is enabled, the data phase bitrate can be configured using the `can dbitrate` subcommand as shown below. The bitrate is specified in bits per second.

```
uart:~$ can dbitrate can@0 1000000
setting data bitrate to 1000000 bps
```

Both of these subcommands allow specifying an optional sample point in per mille and a (Re)Synchronization Jump Width (SJW) in Time Quanta as positional arguments. Refer to the interactive help of the subcommands for more details.

It is also possible to configure the raw bit timing using the `can timing` and `can dtiming` subcommands. Refer to the interactive help output for these subcommands for details on the required arguments.

Mode The CAN shell allows for setting the mode of the CAN controller using the `can mode` subcommand. An example for enabling loopback mode is shown below.

```
uart:~$ can mode can@0 loopback
setting mode 0x00000001
```

The subcommand accepts multiple modes given on the same command line (e.g. `can mode can@0 fd loopback` for setting CAN FD and loopback mode). Vendor-specific modes can be specified in hexadecimal.

Starting and Stopping After the timing and mode has been configured as needed, the CAN controller can be started using the `can start` subcommand as shown below. This will enable reception and transmission of CAN frames.

```
uart:~$ can start can@0
starting can@0
```

Prior to reconfiguring the timing or mode, the CAN controller needs to be stopped using the `can stop` subcommand as shown below:

```
uart:~$ can stop can@0
stopping can@0
```

Receiving In order to receive CAN frames, one or more CAN RX filters need to be configured. CAN RX filters are added using the `can filter add` subcommand as shown below. The subcommand accepts a CAN ID in hexadecimal format along with an optional CAN ID mask, also in hexadecimal format, for setting which bits in the CAN ID are to be matched. Refer to the interactive help output for this subcommand for further details on the supported arguments.

```
uart:~$ can filter add can@0 010
adding filter with standard (11-bit) CAN ID 0x010, CAN ID mask 0x7ff, data frames␣
↪1, RTR frames 0, CAN FD frames 0
filter ID: 0
```

The filter ID (0 in the example above) returned is to be used when removing the CAN RX filter.

Received CAN frames matching the added filter(s) are printed to the shell. A few examples are shown below:

```
# Dev Flags    ID   Size  Data bytes
can0  --       010   [8]  01 02 03 04 05 06 07 08
can0  B-       010  [08]  01 02 03 04 05 06 07 08
can0  BP       010  [03]  01 aa bb
can0  --  00000010   [0]
can0  --       010   [1]  20
can0  --       010   [8]  remote transmission request
```

The columns have the following meaning:

- Dev
  - Name of the device receiving the frame.
- Flags
  - B: The frame has the CAN FD Baud Rate Switch (BRS) flag set.

  - P: The frame has the CAN FD Error State Indicator (ESI) flag set. The transmitting node is in error-passive state.
  - -: Unset flag.
- ID
  - 010: The standard (11-bit) CAN ID of the frame in hexadecimal format, here 10h.
  - 00000010: The extended (29-bit) CAN ID of the frame in hexadecimal format, here 10h.
- Size
  - [8]: The number of frame data bytes in decimal format, here a classic CAN frame with 8 data bytes.
  - [08]: The number of frame data bytes in decimal format, here a CAN FD frame with 8 data bytes.
- Data bytes
  - 01 02 03 04 05 06 07 08: The frame data bytes in hexadecimal format, here the numbers from 1 through 8.
  - remote transmission request: The frame is a Remote Transmission Request (RTR) frame and thus carries no data bytes.

Tip

If CONFIG_CAN_RX_TIMESTAMP is enabled, each line will be prepended with a timestamp from the free-running timestamp counter in the CAN controller.

Configured CAN RX filters can be removed again using the can filter remove subcommand as shown below. The filter ID is the ID returned by the can filter add subcommand (0 in the example below).

```
uart:~$ can filter remove can@0 0
removing filter with ID 0
```

Another option is to use the can dump subcommand, which adds standard (11-bit) and extended (29-bit) CAN filters matching any RX frame, starts the CAN controller, and prints all received CAN frames to the shell:

```
uart:~$ can dump can@0
dumping CAN RX frames on device can@0, press Ctrl+C to exit
```

After exiting the can dump subcommand by pressing Ctrl+C, the added filters are automatically removed and the CAN controller stopped again.

Sending CAN frames can be queued for transmission using the can send subcommand as shown below. The subcommand accepts a CAN ID in hexadecimal format and optionally a number of data bytes, also specified in hexadecimal. Refer to the interactive help output for this subcommand for further details on the supported arguments.

```
uart:~$ can send can@0 010 1 2 3 4 5 6 7 8
enqueuing CAN frame #2 with standard (11-bit) CAN ID 0x010, RTR 0, CAN FD 0, BRS␣
↪0, DLC 8
CAN frame #2 successfully sent
```

Bus Recovery The can recover subcommand can be used for initiating manual recovery from a CAN bus-off event as shown below:

```
uart:~$ can recover can@0
recovering, no timeout
```

The subcommand accepts an optional bus recovery timeout in milliseconds. If no timeout is specified, the command will wait indefinitely for the bus recovery to succeed.

> **Note**
>
> The `recover` subcommand is only available if `CONFIG_CAN_MANUAL_RECOVERY_MODE` is enabled.

Inter-Integrated Circuit (I2C) Bus

> **Note**
>
> The terminology used in Zephyr I2C APIs follows that of the NXP I2C Bus Specification Rev 7.0. These changed from previous revisions as of its release October 1, 2021.

Overview I2C (Inter-Integrated Circuit, pronounced "eye squared see") is a commonly-used two-signal shared peripheral interface bus. Many system-on-chip solutions provide controllers that communicate on an I2C bus. Devices on the bus can operate in two roles: as a "controller" that initiates transactions and controls the clock, or as a "target" that responds to transaction commands. A I2C controller on a given SoC will generally support the controller role, and some will also support the target mode. Zephyr has API for both roles.

I2C Controller API Zephyr's I2C controller API is used when an I2C peripheral controls the bus, particularly the start and stop conditions and the clock. This is the most common mode, used to interact with I2C devices like sensors and serial memory.

This API is supported in all in-tree I2C peripheral drivers and is considered stable.

I2C Target API Zephyr's I2C target API is used when an I2C peripheral responds to transactions initiated by a different controller on the bus. It might be used for a Zephyr application with transducer roles that are controlled by another device such as a host processor.

This API is supported in very few in-tree I2C peripheral drivers. The API is considered experimental, as it is not compatible with the capabilities of all I2C peripherals supported in controller mode.

Configuration Options Related configuration options:

- `CONFIG_I2C`
- `CONFIG_I2C_TRANSFER_TIMEOUT_MS`

Transfer Timeout The I2C subsystem provides two complementary mechanisms to control how long a driver waits for a transfer to complete before returning `-ETIMEDOUT`.

Application-wide default (Kconfig) `CONFIG_I2C_TRANSFER_TIMEOUT_MS` sets the default timeout in milliseconds for all I2C controllers whose drivers opt in by selecting `I2C_TRANSFER_TIMEOUT_SUPPORTED`. A value of `0` means wait forever (`K_FOREVER`). Drivers that do not permit an infinite timeout (e.g. those that program the value directly into a hardware register) use `BUILD_ASSERT_INVALID_I2C_TRANSFER_TIMEOUT` to enforce a non-zero value at build time.

Per-controller DT override Individual controllers can override the application-wide default through the `zephyr,transfer-timeout-ms` devicetree property defined in `dts/bindings/i2c/i2c-controller.yaml`. When the property is absent the driver falls back to `CONFIG_I2C_TRANSFER_TIMEOUT_MS`.

Example board overlay:

```
&i2c1 {
    /* Fast sensor bus - fail quickly on a hung device */
    zephyr,transfer-timeout-ms = <50>;
};

&i2c2 {
    /* EEPROM bus - allow for long internal write cycles */
    zephyr,transfer-timeout-ms = <2000>;
};
```

Drivers that support per-instance timeouts use `I2C_DT_INST_TRANSFER_TIMEOUT` (or `I2C_DT_INST_TRANSFER_TIMEOUT_MS` for raw integer use) which resolve the timeout with the following priority:

1. `zephyr,transfer-timeout-ms` DT property on the controller node
2. `CONFIG_I2C_TRANSFER_TIMEOUT_MS`
3. `K_FOREVER` when the resolved value is `0`

API Reference

I2C

Related code samples

- I2C Custom TargetSetup a custom I2C target on the I2C interface.
- I2C RTIO loopbackPerform I2C transfers between I2C controller and custom I2C target using RTIO.
- I2C TargetSetup an I2C target on the I2C interface.
- I2C V2 timingsRetrieve I2C V2 timings at runtime.
- I2C-Tiny-USB bridgeExpose a Zephyr I2C controller to a host as a USB to I2C adapter.

Improved Inter-Integrated Circuit (I3C) Bus

I3C (Improved Inter-Integrated Circuit) is a two-signal shared peripheral interface bus. Devices on the bus can operate in two roles: as a "controller" that initiates transactions and controls the clock, or as a "target" that responds to transaction commands.

Currently, the API is based on I3C Specification version 1.1.1.

- *I3C Controller API*
  - *In-Band Interrupt (IBI)*
  - *Device Tree*
  - *Device Drivers for I3C Devices*
  - *I^2C Devices under I3C Bus*
- *Configuration Options*

- *API Reference*

I3C Controller API Zephyr's I3C controller API is used when an I3C controller controls the bus, particularly the start and stop conditions and the clock. This is the most common mode, used to interact with I3C target devices such as sensors.

Due to the nature of the I3C, there are devices on the bus where they may not have addresses when powered on. Therefore, an additional dynamic address assignment needs to be carried out by the I3C controller. Because of this, the controller needs to maintain separate structures to keep track of device status. This can be done at build time, for example, by creating arrays of device descriptors for both I3C and I^2C devices:

```
static struct i3c_device_desc i3c_device_array[] = I3C_DEVICE_ARRAY_DT_INST(inst);
static struct i3c_i2c_device_desc i2c_device_array[] =
↪I3C_I2C_DEVICE_ARRAY_DT_INST(inst);
```

The macros `I3C_DEVICE_ARRAY_DT_INST` and `I3C_I2C_DEVICE_ARRAY_DT_INST` are helper macros to aid in create arrays of device descriptors corresponding to the devicetree nodes under the I3C controller.

Here is a list of generic steps for initializing the I3C controller and the I3C bus inside the device driver initialization function:

1. Initialize the data structure of the I3C controller device driver instance. The usual device defining macros such as `DEVICE_DT_INST_DEFINE` can be used, and the initialization function provided as a parameter to the macro.
   - The `i3c_addr_slots` and `i3c_dev_list` are structures to aid in address assignments and device list management. If this is being used, this struct needs to be initialized by calling `i3c_addr_slots_init()`. These two structures can also be used with various helper functions.
   - Initialize the device descriptors if needed by the controller driver.
2. Initialize the hardware, including but not limited to:
   - Setup pin mux and directions.
   - Setup the clock for the controller.
   - Power on the hardware.
   - Configure the hardware (e.g. SCL clock frequency).
3. Perform bus initialization. There is a generic helper function, `i3c_bus_init()`, which performs the following steps. This function can be used if the controller does not require any special handling during bus initialization.
   1. Do `RSTDAA` to reset dynamic addresses of connected devices. If any connected devices have already been assigned an address, the bookkeeping data structures do not have records of these, for example, at power-on. So it is a good idea to reset and assign them new addresses.
   2. Do `DISEC` to disable any events from devices.
   3. Do `SETDASA` to assign a dynamic address using the static address of the device if so desired.
      - `SETAASA` may not be supported for all connected devices to assign static addresses as dynamic addresses.
      - BCR and DCR need to be obtained separately to populate the relevant fields in the I3C target device descriptor struct.

4. Do ENTDAA to start dynamic address assignment, if there are still devices without addresses.
   - If there is a device waiting for address, it will send its Provisioned ID, BCR, and DCR back. Match the received Provisioned ID to the list of registered I3C devices.
     - If there is a match, assign an address (either from the stated static address if SETDASA has not been done, or use a free address).
       - Also, set the BCR and DCR fields in the device descriptor struct.
     - If there is no match, depending on policy, it can be assigned a free address, or the device driver can stop the assignment process and errors out.
       - Note that the I3C API requires device descriptor to function. A device without a device descriptor cannot be accessed through the API.
   - This step can be skipped if there is no connected devices requiring DAA.
5. These are optional but highly recommended:
   - Do GETMRL and GETMWL to get maximum read/write length.
   - Do GETMXDS to get maximum read/write speed and maximum read turnaround time.
   - The helper function, i3c_bus_init(), would retrieve basic device information such as BCR, DCR, MRL and MWL.
6. Do ENEC to re-enable events from devices.
   - The helper function, i3c_bus_init(), only re-enables hot-join events. IBI event should only be enabled when enabling IBI of a device.

In-Band Interrupt (IBI) If a target device can generate In-Band Interrupt (IBI), the controller needs to be made aware of it.

- i3c_ibi_enable() to enable IBI of a target device.
  - Some controller hardware have IBI slots which need to be programmed so that the controller can recognize incoming IBIs from a particular target device.
    - If the hardware has IBI slots, i3c_ibi_enable() needs to program those IBI slots.
    - Note that there are usually limited IBI slots on the controller so this operation may fail.
  - The implementation in driver should also send the ENEC command to enable interrupt of this target device.
- i3c_ibi_disable() to disable IBI of a target device.
  - If controller hardware makes use of IBI slots, this will remove description of the target device from the slots.
  - The implementation in driver should also send the DISEC command to disable interrupt of this target device.

Device Tree Here is an example for defining a I3C controller in device tree:

```
i3c0: i3c@10000 {
        compatible = "vendor,i3c";

        #address-cells = <0x3>;
        #size-cells = <0x0>;

        reg = <0x10000 0x1000>;
```

(continues on next page)

(continued from previous page)

```
        interrupts = <0x1F 0x0>;

        pinctrl-0 = <&pinmux-i3c>;
        pinctrl-names = "default";

        i2c-scl-hz = <400000>;

        i3c-scl-hz = <12000000>;

        status = "okay";

        i3c-dev0: i3c-dev0@420000ABCD12345678 {
                compatible = "vendor,i3c-dev";

                reg = <0x42 0xABCD 0x12345678>;

                status = "okay";
        };

        i2c-dev0: i2c-dev0@380000000000000050 {
                compatible = "vendor-i2c-dev";

                reg = <0x38 0x0 0x50>;

                status = "okay";
        };
};
```

I3C Devices For I3C devices, the `reg` property has 3 elements:

- The first one is the static address of the device.
  - Can be zero if static address is not used. Address will be assigned during DAA (Dynamic Address Assignment).
  - If non-zero and property `assigned-address` is not set, this will be the address of the device after SETDASA (Set Dynamic Address from Static Address) is issued.
- Second element is the upper 16-bit of the Provisioned ID (PID) which contains the manufacturer ID left-shifted by 1. This is the bits 33-47 (zero-based) of the 48-bit Provisioned ID.
  - Must be non-zero. A zero second element marks the node as an I^2C device, as described below, so the helper macros create a legacy I^2C descriptor for it instead of an I3C one. Specify the PID even when the device is addressed by SETDASA and the PID is otherwise unused.
- Third element contains the lower 32-bit of the Provisioned ID which is a combination of the part ID (left-shifted by 16, bits 16-31 of the PID) and the instance ID (left-shifted by 12, bits 12-15 of the PID).

Note that the unit-address (the part after `@`) must match the `reg` property fully where each element is treated as 32-bit integer, combining to form a 96-bit integer. This is required for properly generating device tree macros.

I^2C Devices For I^2C devices where the device driver has support for working under I3C bus, the device node can be described as a child of the I3C controller. If the device driver is written to only work with I^2C controllers, define the node under the I^2C virtual controller as described below. Otherwise, the `reg` property, similar to I3C devices, has 3 elements:

- The first one is the static address of the device. This must be a valid address as I^{2}C devices do not support dynamic address assignment.
- Second element is always zero.
  - This is used by various helper macros to determine whether the device tree entry corresponds to a I^{2}C device.
- Third element is the LVR (Legacy Virtual Register):
  - bit[31:8] are unused.
  - bit[7:5] are the I^{2}C device index:
    - Index `0`
      - I3C device has a 50 ns spike filter where it is not affected by high frequency on SCL.
    - Index 1
      - I^{2}C device does not have a 50 ns spike filter but can work with high frequency on SCL.
    - Index 2
      - I3C device does not have a 50 ns spike filter and cannot work with high frequency on SCL.
  - bit[4] is the I^{2}C mode indicator:
    - `0` is FM+ mode.
    - 1 is FM mode.

Similar to I3C devices, the unit-address must match the `reg` property fully where each element is treated as 32-bit integer, combining to form a 96-bit integer.

Device Drivers for I3C Devices All of the transfer functions of I3C controller API require the use of device descriptors, `i3c_device_desc`. This struct contains runtime information about a I3C device, such as, its dynamic address, BCR, DCR, MRL and MWL. Therefore, the device driver of a I3C device should grab a pointer to this device descriptor from the controller using `i3c_device_find()`. This function takes an ID parameter of type `i3c_device_id` for matching. The returned pointer can then be used in subsequent API calls to the controller.

I^{2}C Devices under I3C Bus Since I3C is backward compatible with I^{2}C, the I3C controller API can accommodate I2C API calls without modifications if the controller device driver implements the I2C API. This has the advantage of using existing I2C devices without any modifications to their device drivers. However, since the I3C controller API works on device descriptors, any calls to I2C API will need to look up the corresponding device descriptor from the I2C device address. This adds a bit of processing cost to any I2C API calls.

On the other hand, a device driver can be extended to utilize native I2C device support via the I3C controller API. During device initialization, `i3c_i2c_device_find()` needs to be called to retrieve the pointer to the device descriptor. This pointer can be used in subsequent API calls.

Note that, with either methods mentioned above, the devicetree node of the I2C device must be declared according to I3C standard:

The I^{2}C virtual controller device driver provides a way to interface I^{2}C devices on the I3C bus where the associated device drivers can be used as-is without modifications. This requires adding an intermediate node in the device tree:

```
i3c0: i3c@10000 {
        <... I3C controller related properties ...>
        <... Nodes of I3C devices, if any ...>
```

(continues on next page)

(continued from previous page)

```
        i2c-dev0: i2c-dev0@420000000000000050 {
                compatible = "vendor-i2c-dev";

                reg = <0x42 0x0 0x50>;

                status = "okay";
        };
};
```

Configuration Options Related configuration options:

- CONFIG_I3C
- CONFIG_I3C_USE_IBI
- CONFIG_I3C_IBI_MAX_PAYLOAD_SIZE
- CONFIG_I3C_CONTROLLER_INIT_PRIORITY

API Reference

I3C

I3C Common Command Codes

I3C Address-related Helper Code

I3C Target Device API

Serial Peripheral Interface (SPI) Bus

Overview

Terminology The Zephyr SPI API uses the inclusive terminology selected by *Rule A.2: Inclusive Language*, following the OSHWA resolution to redefine SPI signal names:

- The device driving the clock is the *controller*, devices it addresses are *peripherals* (see SPI_OP_MODE_CONTROLLER and SPI_OP_MODE_PERIPHERAL).
- Data signals are named from each device's own perspective: *SDO* (Serial Data Out) and *SDI* (Serial Data In), with *CS* (Chip Select) for the select line.

The former master/slave and MOSI/MISO names remain available as compatibility aliases. They are deprecated since Zephyr v4.5 and will be removed in Zephyr v5.0.

API Reference

SPI

Related code samples

- Enhanced Serial Peripheral Interface (eSPI)Use eSPI to connect to a slave device and exchange virtual wire packets.
- HCI SPIExpose a Bluetooth controller to another device or CPU over SPI.
- SPI bitbangUse the bitbang SPI driver for communicating with a peripheral.
- SPI RTIO loopbackPerform SPI transfers between SPI controller and SPI target using RTIO.

- SPIS wake upReduce current consumption by handling the wake line while using an SPIS.

Peripheral Component Interconnect express Bus (PCIe)

Overview

API Reference

PCIe Host

Universal Asynchronous Receiver-Transmitter (UART)

Overview Zephyr provides three different ways to access the UART peripheral. Depending on the method, different API functions are used according to below sections:

1. *Polling API*
2. *Interrupt-driven API*
3. *Asynchronous API* using *Direct Memory Access (DMA)*

Polling is the most basic method to access the UART peripheral. The reading function, `uart_poll_in()`, is a non-blocking function and returns a character or `-1` when no valid data is available. The writing function, `uart_poll_out()`, is a blocking function and the thread waits until the given character is sent.

With the Interrupt-driven API, possibly slow communication can happen in the background while the thread continues with other tasks. The Kernel's *Data Passing* features can be used to communicate between the thread and the UART driver.

The Asynchronous API allows to read and write data in the background using DMA without interrupting the MCU at all. However, the setup is more complex than the other methods.

Warning

Interrupt-driven API and the Asynchronous API should NOT be used at the same time for the same hardware peripheral, since both APIs require hardware interrupts to function properly. Using the callbacks for both APIs would result in interference between each other. `CONFIG_UART_EXCLUSIVE_API_CALLBACKS` is enabled by default so that only the callbacks associated with one API is active at a time.

Configuration Options Most importantly, the Kconfig options define whether the polling API (default), the interrupt-driven API or the asynchronous API can be used. Only enable the features you need in order to minimize memory footprint.

Related configuration options:

- `CONFIG_SERIAL`
- `CONFIG_UART_INTERRUPT_DRIVEN`
- `CONFIG_UART_ASYNC_API`
- `CONFIG_UART_WIDE_DATA`
- `CONFIG_UART_USE_RUNTIME_CONFIGURE`
- `CONFIG_UART_LINE_CTRL`
- `CONFIG_UART_DRV_CMD`

API Reference
UART

Related code samples

- 802.15.4 ”serial-radio”Implement a slip-radio device for Contiki-based border routers.
- AT Command ClientCreate a bidirectional UART bridge between console and modem for AT command communication.
- Console over USB CDC ACMOutput ”Hello World!” to the console over USB CDC ACM.
- HCI 3-wire (H:5)Expose a Bluetooth controller to another device or CPU over H5:HCI transport.
- HCI UARTExpose a Bluetooth controller to another device or CPU over UART.
- HCI UART asyncExpose a Bluetooth controller to another device or CPU over asynchronous UART.
- Legacy USB CDC ACM UART sampleUse USB CDC ACM UART driver to implement a serial port echo.
- Native TTY UARTUse native TTY driver to send and receive data between two UART-to-USB bridge dongles.
- Single-wire UARTUse single-wire/half-duplex UART functionality of STM32 devices.
- UART ASYNC APIDemonstrate the use of the asynchronous API
- UART circular modeRead data from the console and echo it back using a circular DMA configuration.
- UART echoRead data from the console and echo it back.
- UART PassthroughPass data directly between the console and another UART interface.
- USB CDC ACM UART sampleUse USB CDC ACM UART driver to implement a serial port echo.
- USB CDC-ACM bridgeUse USB CDC-ACM driver to implement a serial port bridge.

Polling API
Polling UART API

Interrupt-driven API
Interrupt-driven UART API

Asynchronous API
Async UART API

Enhanced Serial Peripheral Interface (eSPI) Bus

Overview The eSPI (enhanced serial peripheral interface) is a serial bus that is based on SPI. It also features a four-wire interface (receive, transmit, clock and target select) and three configurations: single IO, dual IO and quad IO.

The technical advancements include lower voltage signal levels (1.8V vs. 3.3V), lower pin count, and the frequency is twice as fast (66MHz vs. 33MHz) Because of its enhancements, the eSPI is used to replace the LPC (lower pin count) interface, SPI, SMBus and sideband signals.

See eSPI interface specification for additional details.

API Reference
ESPI

Related code samples

- Enhanced Serial Peripheral Interface (eSPI)Use eSPI to connect to a slave device and exchange virtual wire packets.

Management Data Input/Output (MDIO)

Overview MDIO is a bus that is commonly used to communicate with ethernet PHY devices. Many ethernet MAC controllers also provide hardware to communicate over MDIO bus with a peripheral device.

This API is intended to be used primarily by PHY drivers but can also be used by user firmware.

API Reference
MDIO

Multi-bit SPI Bus

The MSPI (multi-bit SPI) is provided as a generic API to accommodate advanced SPI peripherals and devices that typically require command, address and data phases, and multiple signal lines during these phases. While the API supports advanced features such as *XIP* and scrambling, it is also compatible with generic SPI.

- *MSPI Controller API*
  - *Transceive*
  - *Device Tree*
  - *Multi Peripheral*
- *Configuration Options*
- *API Reference*

MSPI Controller API Zephyr's MSPI controller API may be used when a multi-bit SPI controller is present. E.g. Ambiq MSPI, QSPI, OSPI, Flexspi, etc. The API supports single to hex SDR/DDR IO with variable latency and advanced features such as *XIP* and scrambling. Applicable devices include but not limited to high-speed, high density flash/psram memory devices, displays and sensors.

The MSPI interface contains controller drivers that are SoC platform specific and implement the MSPI APIs, and device drivers that reference these APIs. The relationship between the controller and device drivers is many-to-many to allow for easy switching between platforms.

Here is a list of generic steps for initializing the MSPI controller and the MSPI bus inside the device driver initialization function:

1. Initialize the data structure of the MSPI controller driver instance. The usual device defining macros such as `DEVICE_DT_INST_DEFINE` can be used, and the initialization function, config and data provided as a parameter to the macro.
2. Initialize the hardware, including but not limited to:
   - Check `mspi_cfg` against hardware's own capabilities to prevent incorrect usages.

- Setup default pinmux.
- Setup the clock for the controller.
- Power on the hardware.
- Configure the hardware using `mspi_cfg` and possibly more platform specific settings.
- Usually, the `mspi_cfg` is filled from device tree and contains static, boot time parameters. However, if needed, one can use `mspi_config()` to re-initialize the hardware with new parameters during runtime.
- Release any lock if applicable.

3. Perform device driver initialization. As usually, `DEVICE_DT_INST_DEFINE` can be used. Inside device driver initialization function, perform the following required steps.
   1. Call `mspi_dev_config()` with device specific hardware settings obtained from device datasheets.
      - The `mspi_dev_cfg` should be filled by device tree and helper macro `MSPI_DEVICE_CONFIG_DT` can be used.
      - The controller driver should then validate the members of `mspi_dev_cfg` to prevent incorrect usage.
      - The controller driver should implement a mutex to protect from accidental access.
      - The controller driver may also switch between different devices based on `mspi_dev_id`.
   2. Call API for additional setups if supported by hardware
      - `mspi_memmap_config()` for memory mapped access (e.g. *XIP*)
      - `mspi_scramble_config()` for scrambling feature
      - `mspi_timing_config()` for platform specific timing setup.
   3. Register any callback with `mspi_register_callback()` if needed.
   4. Release the controller mutex lock.

Transceive The transceive request is of type `mspi_xfer` which allows dynamic change to the transfer related settings once the mode of operation is determined and configured by `mspi_dev_config()`.

The API also supports bulk transfers with different starting addresses and sizes with `mspi_xfer_packet`. However, it is up to the controller implementation whether to support scatter IO and callback management. The controller can determine which user callback to trigger based on `mspi_bus_event_cb_mask` upon completion of each async/sync transfer if the callback had been registered using `mspi_register_callback()`. Or not to trigger any callback at all with `MSPI_BUS_NO_CB` even if the callbacks are already registered. If the implemented driver support it, the API supports `MSPI_BUS_XFER_COMPLETE_CB` to signal when a transfer finishes and `MSPI_BUS_TIMEOUT_CB` to signal when a transfer or request has timed-out. In which case that a controller supports hardware command queue, user could take full advantage of the hardware performance if scatter IO and callback management are supported by the driver implementation.

Device Tree Here is an example for defining an MSPI controller in device tree: The mspi controller's bindings should reference mspi-controller.yaml as one of the base.

```
mspi0: mspi@400 {
        status = "okay";
        compatible = "zephyr,mspi-emul-controller";
```

(continues on next page)

(continued from previous page)

```
        reg = <0x400 0x4>;
        #address-cells = <0x1>;
        #size-cells = <0x0>;

        clock-frequency = <0x17d7840>;
        op-mode = "MSPI_CONTROLLER";
        duplex = "MSPI_HALF_DUPLEX";
        ce-gpios = <&gpio0 0x5 0x1>, <&gpio0 0x12 0x1>;
        dqs-support;

        pinctrl-0 = <&pinmux-mspi0>;
        pinctrl-names = "default";
};
```

Here is an example for defining an MSPI device in device tree: The mspi device's bindings should reference mspi-device.yaml as one of the base.

```
&mspi0 {

        mspi_dev0: mspi_dev0@0 {
                status = "okay";
                compatible = "zephyr,mspi-emul-device";

                reg = <0x0>;
                size = <0x10000>;

                mspi-max-frequency = <0x2dc6c00>;
                mspi-io-mode = "MSPI_IO_MODE_QUAD";
                mspi-data-rate = "MSPI_DATA_RATE_SINGLE";
                mspi-hardware-ce-num = <0x0>;
                read-instruction = <0xb>;
                write-instruction = <0x2>;
                instruction-length = "INSTR_1_BYTE";
                address-length = "ADDR_4_BYTE";
                rx-dummy = <0x8>;
                tx-dummy = <0x0>;
                memmap-config = <0x0 0x0 0x0 0x0>;
                ce-break-config = <0x0 0x0>;
        };

};
```

User should specify target operating parameters in the DTS such as `mspi-max-frequency`, `mspi-io-mode` and `mspi-data-rate` even though they may subject to change during runtime. It should represent the typical configuration of the device during normal operations.

Multi Peripheral With `mspi_dev_id` defined as collection of the device index and CE GPIO from device tree, the API supports multiple devices on the same controller instance. The controller driver implementation may or may not support device switching, which can be performed either by software or by hardware. If the switching is handled by software, it should be performed in `mspi_dev_config()` call.

The device driver should record the current operating conditions of the device to support software controlled device switching by saving and updating `mspi_dev_cfg` and other relevant mspi struct or private data structures. In particular, `mspi_dev_id` which contains the identity of the device needs to be used for every API call.

Configuration Options Related configuration options:

- CONFIG_MSPI
- CONFIG_MSPI_ASYNC
- CONFIG_MSPI_PERIPHERAL
- CONFIG_MSPI_MEMMAP
- CONFIG_MSPI_SCRAMBLE
- CONFIG_MSPI_TIMING
- CONFIG_MSPI_INIT_PRIORITY
- CONFIG_MSPI_COMPLETION_TIMEOUT_TOLERANCE
- CONFIG_MSPI_DMA

API Reference

MSPI

Related code samples

- MSPI asynchronous transferUse the MSPI API to interact with MSPI memory device asynchronously.
- MSPI throughput performance testMeasure MSPI throughput performance with different buffer sizes and packet counts.

System Management Bus (SMBus)

- *Overview*
- *SMBus Controller API*
- *Configuration Options*
- *API Reference*

Overview System Management Bus (SMBus) is derived from I2C for communication with devices on the motherboard. A system may use SMBus to communicate with the peripherals on the motherboard without using dedicated control lines. SMBus peripherals can provide various manufacturer information, report errors, accept control parameters, etc.

Devices on the bus can operate in three roles: as a Controller that initiates transactions and controls the clock, a Peripheral that responds to transaction commands, or a Host, which is a specialized Controller, that provides the main interface to the system's CPU. Zephyr has API for the Controller role.

SMBus peripheral devices can initiate communication with Controller with two methods:

- **Host Notify protocol**: Peripheral device that supports the Host Notify protocol behaves as a Controller to perform the notification. It writes a three-bytes message to a special address "SMBus Host (0x08)" with own address and two bytes of relevant data.
- **SMBALERT# signal**: Peripheral device uses special signal SMBALERT# to request attention from the Controller. The Controller needs to read one byte from the special "SMBus Alert Response Address (ARA) (0x0c)". The peripheral device responds with a data byte containing its own address.

Currently, the API is based on SMBus Specification version 2.0

> **Note**
>
> See *Rule A.2: Inclusive Language* for information about the terminology used in this API.

SMBus Controller API Zephyr's SMBus controller API is used when an SMBus device controls the bus, particularly the start and stop conditions and the clock. This is the most common mode used to interact with SMBus peripherals.

Configuration Options Related configuration options:

- `CONFIG_SMBUS`

API Reference
SMBus

> **Related code samples**
>
> - SMBus shellInteract with SMBus peripherals using shell commands.

Inter-Processor Mailbox (IPM)

Overview

API Reference
IPM

> **Related code samples**
>
> - Flash IPM on ESP32Sample of flash API sharing between CPU cores using inter-processor mailbox (IPM).
> - Hardware Semaphore (HSEM) Inter-Processor Communication on STM32 H7Use the Hardware Semaphore (HSEM) to trigger LED blinking from one core to another.
> - IPM on ESP32Implement inter-processor mailbox (IPM) between ESP32 APP and PRO CPUs.
> - IPM on NXP i.MXImplement inter-processor mailbox (IPM) on i.MX SoCs containing a Messaging Unit peripheral.
> - IPM on NXP LPCImplement inter-processor mailbox (IPM) on NXP LPC family.
> - IPM over IVSHMEMImplement inter-processor mailbox (IPM) over IVSHMEM (Inter-VM shared memory)
> - IPM with ARM MHUImplement inter-processor mailbox (IPM) using an MHU (Message Handling Unit)
> - OpenAMPSend messages between two cores using OpenAMP.
> - OpenAMP DSP AMP SampleSend messages between multiple cores using OpenAMP with concurrent communication channels.
> - OpenAMP using resource tableSend messages between two cores using OpenAMP and a resource table.

Local Interconnect Network (LIN)

Overview LIN (Local Interconnect Network) is a serial bus system. Since 2016, it is standardized internationally by ISO. In 2025, several parts of the ISO 17987 series have been updated. The ISO 17987 document series covers the requirements of the seven OSI (Open Systems Interconnection) layers and the corresponding conformance test plans.

LIN lower layers The LIN data link layer, often called the LIN protocol, specifies a commander-responder protocol. The LIN Commander uses one or more pre-programmed scheduling tables to start the sending and receiving to the LIN frames. These scheduling tables contain at least the relative timing, when the LIN frame sending is initiated. The LIN frame consists of two parts: header and response. The LIN Commander sends the header, while either one dedicated LIN Responder or the LIN Commander itself transmits the response.

The header comprises the 1-byte Break, the Inter-Byte Space, the 1-byte SYNC (0x55), the Identifier, and the Response Space. The responses consist of the payload-bytes and the CRC byte.

Transmitted data within the LIN frame is transmitted serially as 8-bit data-bytes with one additional start bit, one additional stop-bit, but no parity bit. Note that the break field in the header has no start bit and no stop bit.

Bit rates may vary within the range of 1 kbit/s to 20 kbit/s. Bit values on the bus are recessive (logical high) or dominant (logical low). The time normal is considered by the LIN Commanders stable clock source, the smallest entity is one bit time (52 µs at a bitrate of 19,2 kbit/s). Two bus states are defined: sleep mode and active mode. While data is on the bus, all LIN-nodes are requested to be in active mode. After a specified timeout, the nodes enter the sleep mode and are released back to active mode by a wake-up frame. This frame may be sent by any node requesting activity on the bus, either the LIN Commander following its internal schedule, or one of the attached LIN Responders being activated by its internal software application. After all nodes are awakened, the LIN Commander continues to schedule the next LIN frame.

For more technical information on LIN you may visit LIN on Wikipedia.

Zephyr LIN controller support following LIN features:

- Send and receive LIN frames in commander and responder mode.
- Filters with header masking (Responder mode) that allow a header ID to trigger a responder callback.
- Wake up pulse sending.

Samples We have samples demonstrating the use of Zephyr LIN controller API:

- lin-ncv7430: Demonstrates how to use the LIN API in commander mode.
- lin-ncv7430-responder: Demonstrates how to use the LIN API in responder mode by mocking the NCV7430 LED Controller.

LIN Transceivers A LIN transceiver is an external device that converts the logic level signals from the LIN controller to the LIN bus levels. Some boards integrate LIN transceiver control (enable/wake pins) directly via the `lin-transceiver-gpio` compatible; the LIN controller driver enables and disables the transceiver automatically as part of `lin_start()` and `lin_stop()`, so applications should not normally need to call the transceiver API directly.

API Reference
LIN Controller

Related code samples

- NCV7430 LED Controller over LINSample application for using the NCV7430 LED Controller with Zephyr's LIN API.
- NCV7430 mock LIN responderSample application that mocks the NCV7430 LED Controller as a LIN responder node.

LIN Transceiver API Reference

LIN Transceiver

Multi-Channel Inter-Processor Mailbox (MBOX)

Overview An MBOX device is a peripheral capable of passing signals (and data depending on the peripheral) between CPUs and clusters in the system. Each MBOX instance is providing one or more channels, each one targeting one other CPU cluster (multiple channels can target the same cluster).

API Reference

MBOX

Related code samples

- HWSPINLOCKDemonstrate cross-core synchronization using the HWSPINLOCK API together with the MBOX API.
- MBOXPerform inter-processor mailbox communication using the MBOX API.
- MBOX DataPerform inter-processor mailbox communication using the MBOX API with data.
- OpenAMP Linux Zephyr RPMsgEnable message exchange between two cores, with the application core running Linux and the real-time core running Zephyr, using the OpenAMP library.

1-Wire Bus

Overview 1-Wire is a low speed half-duplex serial bus using only a single wire plus ground for both data transmission and device power supply. Similarly to I2C, 1-Wire uses a bidirectional open-collector data line, and is a single master multidrop bus. This means one master initiates all data exchanges with the slave devices. The 1-Wire bus supports longer bus lines than I2C, while it reaches speeds of up to 15.4 kbps in standard mode and up to 125 kbps in overdrive mode. Reliable communication in standard speed configuration is possible with 10 nodes over a bus length of 100 meters. Using overdrive speed, 3 nodes on a bus of 10 meters length are expected to work solid. Optimized timing parameters and fewer nodes on the bus may allow to reach larger bus extents.

The implementation details are specified in the BOOK OF IBUTTON STANDARDS.

W1 Master API Zephyr's 1-Wire Master API is used to interact with 1-Wire slave devices like temperature sensors and serial memories.

In Zephyr this API is split into the following layers.

- The link layer handles basic communication functions such as bus reset, presence detect and bit transfer operations. It is the only hardware-dependent layer in Zephyr. This layer is supported by a driver using the Zephyr *Universal Asynchronous Receiver-Transmitter (UART)* interface, which should work on most Zephyr platforms. In the future, a GPIO/Timer based driver and hardware specific drivers might be added.

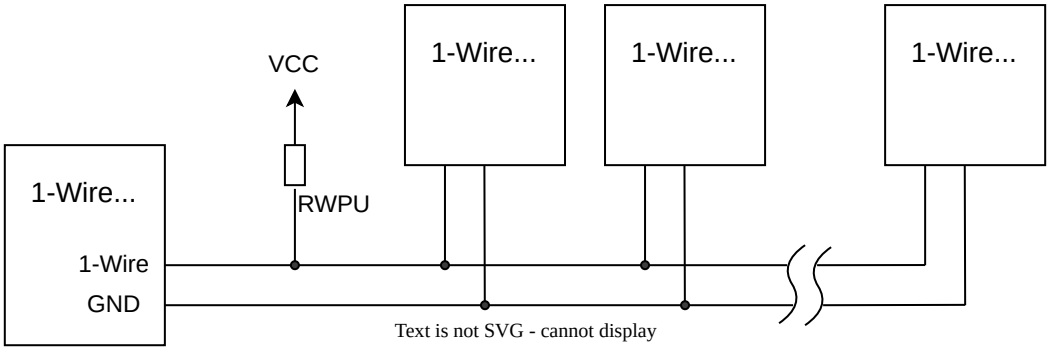


Fig. 1: A typical 1-Wire bus topology

- The 1-Wire network layer handles all means for slave identification and bus arbitration. This includes ROM commands like Match ROM, or Search ROM.
  - All slave devices have a unique 64-bit identification number, which includes a 8-bit 1-Wire Family Code and a 8-bit CRC.
  - In order to find slaves on the bus, the standard specifies an search algorithm which successively detects all slaves on the bus. This algorithm is described in the 1-Wire Search Algorithm Application Note.
- Transport layer and Presentation layer functions are not implemented in the generic 1-Wire driver and therefore must be handled in individual slave drivers.

The 1-Wire API is considered experimental.

Configuration Options Related configuration options:

- `CONFIG_W1`
- `CONFIG_W1_NET`

W1 Serial Driver The `zephyr,w1-serial` driver implements the 1-Wire interface using a UART peripheral. Individual 1-Wire bits are encoded as bytes sent over the UART. For standard speed, it uses 115200 Baud for data and 9600 Baud for reset/presence detection. For overdrive speed, it uses configurable baud rates with a default of 1000000 Baud for data and 115200 Baud for reset/presence.

API Reference

1-Wire data link layer

1-Wire data link layer

1-Wire network layer

1-Wire network layer

1-Wire generic functions and helpers Functions that are not directly related to any of the networking layers.

1-Wire

Related code samples

- 1-Wire scannerScan for 1-Wire devices and print their family ID and serial number.

Pulse IO

Overview The Pulse IO subsystem provides a vendor-neutral API for hardware that generates and captures timed digital edges on a single GPIO line. Several MCU families ship a dedicated peripheral for this purpose under different names. Pulse IO abstracts that hardware class so client drivers such as addressable LED strips, infrared transmit and receive, stepper pulse generation, frequency and duty measurement, and one-wire style protocols can bind once and run on any SoC that provides a backend.

Related configuration options:

- `CONFIG_PULSE_IO`

Submission modes A channel is locked to one submission mode at configure time, selected from those the backend advertises through its capabilities.

`PULSE_IO_MODE_SYMBOL`
: The application submits an array of `pulse_symbol`, each carrying an explicit level and a duration in ticks of the configured resolution. Consecutive symbols may have arbitrary, unrelated durations. Use it for protocols whose edge lengths vary within one stream, such as infrared remotes, one-wire and stepper acceleration ramps.

`PULSE_IO_MODE_CELL`
: The application submits an array of `pulse_cell`. Every cell has the same period, set once in the channel configuration, and each cell carries either a level or a duty value within that period. Use it for naturally periodic streams where only the per-period level or duty changes, such as addressable LED bit shaping or varying-duty PWM. Cells are more memory efficient than symbols for these cases.

Backends advertise the supported modes in `pulse_io_caps`. Requesting an unsupported mode causes `pulse_io_channel_configure()` to return `-ENOTSUP`.

Configuration Clients query `pulse_io_get_capabilities()` at probe time to decide which mode and feature set to use, reserve a channel with `pulse_io_channel_get()`, and configure it with `pulse_io_channel_configure()` before any transfer. Blocking transfers use `pulse_io_transmit_sync()` and `pulse_io_receive_sync()`; asynchronous and streaming transfers use the RTIO path.

The byte-to-symbol helper `pulse_io_encode_bytes()` and its inverse `pulse_io_decode_bytes()` convert between a protocol byte stream and pulse symbols using a per-bit template.

RTIO A backend may optionally integrate with *Real Time I/O (RTIO)*, exposing an iodev submit path along with encoder and decoder vtables fetched through `pulse_io_get_encoder()` and `pulse_io_get_decoder()`. With RTIO the client encodes a payload into a symbol buffer, submits it through an RTIO queue, and decodes any captured reply, gaining queued and chained transfers, streaming receive, and the userspace path provided by the RTIO framework. The RTIO ops are optional; a backend that does not implement them leaves the corresponding driver API members `NULL` and the accessors return `-ENOSYS`.

API Reference

Pulse IO

Related code samples

- pulse_io byte transferTransfer bytes over a pulse_io controller using the codec helpers.

Display

6.7.4 Display

Visual output, video interfaces, and LED and lighting control.

Display

The Display subsystem in Zephyr provides a unified way to interact with a wide range of display devices. The Display API is transport-agnostic: it describes what you want the display to do, without exposing how the data moves over the wire.

MIPI Display Bus Interface (DBI) The **MIPI DBI** specification defines several parallel and serial buses for connecting a host to a display controller. In Zephyr, DBI support provides bus-level primitives for command writes, reads, pixel transfers, reset, and related operations that a driver uses internally to implement the generic API.

Applications do not use DBI functions directly. Instead, they call the generic Display API (e.g., to write pixels), and the display driver handles the DBI protocol under the hood.

MIPI-DBI defines 3 interface types:

- Type A: Motorola 6800 parallel bus
- Type B: Intel 8080 parallel bus
- Type C: SPI Type serial bit bus with 3 options:
  1. 9 write clocks per byte, final bit is command/data selection bit
  2. Same as above, but 16 write clocks per byte
  3. 8 write clocks per byte. Command/data selected via GPIO pin

Currently, the API does not support Type C controllers with 16 write clocks (option 2).

MIPI Display Serial Interface (DSI) The **MIPI DSI** standard is a high-speed differential serial bus designed for modern color TFT panels. Zephyr's DSI support provides the primitives needed by drivers to implement the generic Display API on top of a DSI link.

As with DBI, applications never call DSI functions directly. They remain portable by using the generic Display API, while the driver handles the DSI transactions internally.

API Reference

Generic Display Interface
Display

Related code samples

- Color Palette DisplayDisplay vertical color bands using indexed color palette.
- DisplayDraw basic rectangles on a display device.
- Draw touch eventsVisualize touch events on a display.
- LVGL basic sampleDisplay a "Hello World" and react to user input using LVGL.
- LVGL demosRun LVGL built-in demos.
- LVGL line chart with accelerometer dataDisplay acceleration data on a real-time chart using LVGL.
- LVGL Multi-displayRun different LVGL demos on multiple displays.
- LVGL screen transparencyRendering to screens with transparency support using LVGL.

MIPI Display Bus Interface (DBI)
MIPI-DBI

MIPI Display Serial Interface (DSI)
MIPI-DSI

Grove LCD Display
Grove display

Related code samples

- Grove LCDDisplay an incrementing counter and change the backlight color.

BBC micro:bit Display
BBC micro:bit display APIs

Monochrome Character Framebuffer
Monochrome Character Framebuffer

Related code samples

- Bluetooth Mesh badgeImplement a smart badge using the reel board and Bluetooth Mesh.
- Character frame bufferDisplay character strings using the Character Frame Buffer (CFB).
- Character Framebuffer shell moduleUse the CFB shell module to interact with a monochrome display.
- Custom fontsGenerate and use a custom font.

Auxiliary Display (auxdisplay)

Overview Auxiliary Displays are text-based displays that have simple interfaces for displaying textual, numeric or alphanumeric data, as opposed to the *Display*, auxiliary displays do not support custom graphical output to displays (and most often monochrome), the most advanced custom feature supported is generation of custom characters. These inexpensive displays are commonly found with various configurations and sizes, a common display size is 16 characters by 2 lines.

This API is unstable and subject to change.

Configuration Options Related configuration options:

- `CONFIG_AUXDISPLAY`
- `CONFIG_AUXDISPLAY_INIT_PRIORITY`

API Reference
Auxiliary (Text) Display

Related code samples

- Auxiliary digits displayOutput increasing numbers to an auxiliary display.
- Auxiliary displayOutput ”Hello World” to an auxiliary display.

Video

The video driver API offers a generic interface to video devices.

Basic Operation

Video Device A video device is the abstraction of a hardware or software video function, which can produce, process, consume or transform video data. The video API is designed to offer flexible way to create, handle and combine various video devices.

Endpoint Each video device can have one or more endpoints. Output endpoints configure video output function and generate data. Input endpoints configure video input function and consume data.

Video Buffer A video buffer provides the transport mechanism for the data. There is no particular requirement on the content. The requirement for the content is defined by the endpoint format. A video buffer can be queued to a device endpoint for filling (input ep) or consuming (output ep) operation, once the operation is achieved, buffer can be dequeued for post-processing, release or reuse.

Controls A video control is accessed and identified by a CID (control identifier). It represents a video control property. Different devices will have different controls available which can be generic, related to a device class or vendor specific. The set/get control functions provide a generic scalable interface to handle and create controls.

Configuration Options Related configuration options:

- `CONFIG_VIDEO`

API Reference

Video Subsystem

Video Drivers

Related code samples

- USB Host UVC CameraCapture video from a USB camera.
- USB Video webcamSend video frames over USB.
- Video captureUse the video API to retrieve video frames from a capture device.
- Video capture to LVGLCapture video frames and display them on an LCD using LVGL.
- Video TCP server sinkCapture video frames and send them over the network to a TCP client.

Video Control IDs

Light-Emitting Diode (LED)

Overview The LED API provides access to Light Emitting Diodes, both in individual and strip form. It abstracts the differences between simple GPIO-driven LEDs, PWM-dimmable LEDs, dedicated LED controller ICs, and addressable LED strips behind a small set of common operations.

Two related subsystems are exposed:

- **LED** — general-purpose API for controlling discrete or controller-backed LEDs (on/off, brightness, color, blink)
- **LED Strip** — API tailored to addressable LED strings such as WS2812 and APA102 where each pixel carries its own color value

LED Operations A driver implements a subset of the operations below. Calls to optional operations return `-ENOSYS` if the underlying driver does not provide them.

Mandatory operations (at least one of):

- `led_on()` / `led_off()` — turn an LED fully on or off
- `led_set_brightness()` — set brightness in the range 0 to `LED_BRIGHTNESS_MAX` (100); when implemented, this is also used automatically by `led_on()` and `led_off()`

Optional operations:

- `led_blink()` — start the LED blinking with given on/off durations
- `led_set_color()` — set the per-channel color values of a multi-color LED
- `led_get_info()` — retrieve the `led_info` describing a particular LED (label, index, color mapping)
- `led_write_channels()` — write a range of raw channel values, used by drivers that expose LEDs as an array of channels

LED Indexing Each LED controlled by a driver is addressed by a zero-based `led` index. Multi-color LEDs occupy a single index; their color channels are submitted together to `led_set_color()`.

The color channel order for a multi-color LED is described by the `color_mapping` field of `led_info`. Each entry is one of the `LED_COLOR_ID_*` values defined in `include/zephyr/dt-bindings/led/led.h`:

- `LED_COLOR_ID_WHITE`
- `LED_COLOR_ID_RED`
- `LED_COLOR_ID_GREEN`
- `LED_COLOR_ID_BLUE`
- `LED_COLOR_ID_AMBER`
- `LED_COLOR_ID_VIOLET`
- `LED_COLOR_ID_YELLOW`
- `LED_COLOR_ID_IR`

Device Tree Configuration Simple LEDs are typically described as child nodes of an LED controller and referenced via a devicetree alias:

```
/ {
    aliases {
        led0 = &status_led;
    };

    leds {
        compatible = "gpio-leds";
        status_led: led_0 {
            gpios = <&gpio0 13 GPIO_ACTIVE_HIGH>;
            label = "Status LED";
```

(continues on next page)

(continued from previous page)

```
        };
    };
};
```

For multi-color LEDs, color information is encoded with the `color-mapping` property using `LED_COLOR_ID_*` values from `include/zephyr/dt-bindings/led/led.h`.

Usage Examples Basic on/off control using a devicetree alias:

```
#include <zephyr/drivers/led.h>
#include <zephyr/devicetree.h>

#define LED_NODE DT_ALIAS(led0)
static const struct device *led_dev = DEVICE_DT_GET(DT_PARENT(LED_NODE));
static const uint32_t led_idx = DT_NODE_CHILD_IDX(LED_NODE);

int turn_on_status_led(void)
{
    if (!device_is_ready(led_dev)) {
        return -ENODEV;
    }
    return led_on(led_dev, led_idx);
}
```

Setting brightness (0 to 100):

```
/* Dim the LED to 25% brightness */
led_set_brightness(led_dev, led_idx, 25);
```

Blinking with a 500 ms on / 500 ms off pattern:

```
int err = led_blink(led_dev, led_idx, 500, 500);

if (err == -ENOSYS) {
    /* Driver does not implement blink natively */
}
```

Setting the color of a multi-color (RGB) LED:

```
/* Order of the values must match the color_mapping reported by
 * led_get_info(); for a standard RGB mapping the order is R, G, B.
 */
const uint8_t purple[] = { 0x80, 0x00, 0x80 };

led_set_color(led_dev, led_idx, ARRAY_SIZE(purple), purple);
```

Configuration Options Related configuration options:

- `CONFIG_LED`
- `CONFIG_LED_STRIP`
- `CONFIG_LED_SHELL`
- `CONFIG_LED_INIT_PRIORITY`

API Reference

LED
LED

Related code samples

- Breathing-blinking LED (BBLED)Control a BBLED (Breathing-Blinking LED) using Microchip XEC driver.
- HT16K33 LED driver with keyscanControl up to 128 LEDs connected to an HT16K33 LED driver and log keyscan events.
- IS31FL319x RGB LEDCycle colors on an RGB LED connected to the IS31FL3193, IS31FL3194 or IS31FL3197 using the LED API.
- IS31FL3216A LEDControl up to 16 PWM LEDs connected to an IS31FL3216A driver chip.
- IS31FL3733 LED MatrixControl a matrix of up to 192 LEDs connected to an IS31FL3733 driver chip.
- LED PWMControl PWM LEDs using the LED API.
- LP3943 RGBW LEDControl up to 16 RGBW LEDs connected to an LP3943 driver chip.
- LP50XX RGB LEDControl up to 12 RGB LEDs connected to an LP50xx driver chip.
- LP5562 RGB LEDControl 4 RGB LEDs connected to an LP5562 driver chip.
- LP5569 9-channel LED controllerControl 9 LEDs connected to an LP5569 driver chip.
- LP586x LED matrixControl LED dots on an LP586x LED matrix controller.
- PCA9633 LEDControl 4 LEDs connected to a PCA9633 driver chip.
- SCT2024 LED ControllerControl up to 16 LEDs connected to an SCT2024 chip.
- SX1509B RGB LEDControl an RGB LED connected to an SX1509B driver chip.

LED Strip
LED Strip

Related code samples

- LED stripControl an LED strip.

Fake LED controller
Fake LED controller

DALI

DALI is the digital addressable lighting interface, a communication standard for professional lighting solutions. This API is unstable and subject to change.

Basic Operation The DALI standard uses a communication model that is based on the exchange of frames via a bus system. A frame is manchester encoded data, terminated by a stop condition. The DALI standard requires specific behaviour of transmitter and receiver.

Configuration Options Related configuration options:

- `CONFIG_DALI`
- `CONFIG_DALI_PWM`

API Reference
DALI
GPIO & Input

6.7.5 GPIO & Input

General-purpose I/O and input devices.

General-Purpose Input/Output (GPIO)

Overview A General-Purpose Input/Output (GPIO) is a digital signal pin that does not have a specific function but can be controlled by software to function as an input or an output.

The GPIO API provides a generic method to interact with General Purpose Input/Output (GPIO) pins. It allows applications to configure pins as inputs or outputs, read and write their state, and manage interrupts. Key features include:

Pin Configuration
Configure pins as input, output, or disconnected. Support for internal pull-up/pull-down resistors and drive strength configuration.

Data Access
Read input values and write output values.

Interrupts
Configure interrupts on pin state changes (rising edge, falling edge, level low, level high) and register callbacks to handle these interrupts.

Devicetree Integration
GPIOs are typically defined in the Devicetree, allowing drivers and applications to reference them in a hardware-agnostic way using `gpio_dt_spec`.

Devicetree Configuration GPIO controllers are defined in the Devicetree as nodes with the `gpio-controller` property. The `#gpio-cells` property typically specifies that 2 cells are used to describe a GPIO: the pin number and the flags.

Example of a GPIO controller definition:

```
gpio0: gpio@40022000 {
    compatible = "ti,cc13xx-cc26xx-gpio";
    reg = <0x40022000 0x400>;
    interrupts = <0 0>;
    gpio-controller;
    #gpio-cells = <2>;
};
```

Example of referencing a GPIO:

```
leds {
    compatible = "gpio-leds";
    led0: led_0 {
        gpios = <&gpio0 25 GPIO_ACTIVE_HIGH>;
        label = "Green LED";
    };
};
```

Basic Operation GPIO operations are usually performed on a `gpio_dt_spec` structure, which is a container for the GPIO pin information specified in the Devicetree.

This structure is typically populated using `GPIO_DT_SPEC_GET` macro (or any of its variants).

Listing 2: Populating a gpio_dt_spec structure for a GPIO pin defined as alias `led0`

```
#define LED0_NODE DT_ALIAS(led0)
static const struct gpio_dt_spec led = GPIO_DT_SPEC_GET(LED0_NODE, gpios);
```

The `gpio_dt_spec` structure can then be used to perform GPIO operations.

Listing 3: Configure a GPIO pin (`led` from the previous snippet) as output, initially inactive, and then set it to its active level.

```
int ret;

ret = gpio_pin_configure_dt(&led, GPIO_OUTPUT_INACTIVE);
if (ret < 0) {
    return ret;
}

ret = gpio_pin_set_dt(&led, 1);
if (ret < 0) {
    return ret;
}
```

Refer to blinky for a complete example of basic operations on GPIOs using the `gpio_dt_spec` structure.

GPIO operations can also be performed directly on a GPIO controller device, in which case you will use the GPIO API functions that take a device pointer as an argument. For example `gpio_pin_configure()` instead of `gpio_pin_configure_dt()`.

GPIO Hogs GPIO hogs provide a mechanism to automatically configure GPIO pins during system initialization. This is useful for pins that need to be set to a specific state (e.g., reset lines, power enables) and don't require runtime control by an application.

Hogs are defined in the Devicetree as children of a GPIO controller node.

- The `gpio-hog` property marks the node as a hog.
- The `gpios` property specifies the pin and its active state.
- One of `input`, `output-low`, or `output-high` properties specifies the configuration.

Example Devicetree overlay:

```
&gpio0 {
    hog1 {
        gpio-hog;
        gpios = <1 GPIO_ACTIVE_LOW>;
        output-high;
    };

    hog2 {
        gpio-hog;
        gpios = <2 GPIO_ACTIVE_HIGH>;
        output-low;
    };
};
```

If you need runtime control of a pin configured as a hog beyond the initial state being automatically set during system initialization, you may consider using the *Regulators* instead, with a

`regulator-fixed` Devicetree node.

Configuration Options Main configuration options:

- `CONFIG_GPIO`
- `CONFIG_GPIO_SHELL`
- `CONFIG_GPIO_GET_DIRECTION`
- `CONFIG_GPIO_GET_CONFIG`
- `CONFIG_GPIO_HOGS`
- `CONFIG_GPIO_ENABLE_DISABLE_INTERRUPT`

API Reference
GPIO

Related code samples

- B-M2MEM-PACK1 shieldCheck a B-M2MEM-PACK1 memory module against the data stored in its identification EEPROM.
- Basic thread manipulationSpawn multiple threads that blink LEDs and print information to the console.
- BlinkyBlink an LED forever using the GPIO API.
- Button interruptHandle GPIO inputs with interrupts.
- GPIO as wake-up pinUse a GPIO pin as wake-up pin.
- GPIO with custom Devicetree bindingUse custom Devicetree binding to control a GPIO.
- HD44780 LCD controllerControl an HD44780-based LCD display using GPIO pins.
- Pico W WiFi + LEDBlink the onboard LED while scanning for WiFi networks.
- X-NUCLEO-53L0A1 shieldInteract with the 7-segment display and VL53L0X ranging sensor of an X-NUCLEO-53L0A1 shield.

PS/2

Overview The PS/2 connector first hit the market in 1987 on IBM's desktop PC line of the same name before becoming an industry-wide standard for mouse and keyboard connections. Starting around 2007, USB superseded PS/2 and is the modern peripheral device connection standard. For legacy support on boards with a PS/2 connector, Zephyr provides these PS/2 driver APIs.

Configuration Options Related configuration options:

- `CONFIG_PS2`

API Reference
PS/2

Related code samples

- PS/2 interfaceCommunicate with a PS/2 mouse.

Time-aware General-Purpose Input/Output (TGPIO)

Overview

Configuration Options Related configuration options:

- `CONFIG_TIMEAWARE_GPIO`

API Reference
Time-aware GPIO

Related code samples

- Time-aware GPIOSynchronize clocks.

Memory & Storage

6.7.6 Memory & Storage

Non-volatile memory, flash storage, EEPROM, and memory controllers.

Flash

Overview **Flash offset concept**

Offsets used by the user API are expressed in relation to the flash memory beginning address. This rule shall be applied to all flash controller regular memory that layout is accessible via API for retrieving the layout of pages (see `CONFIG_FLASH_PAGE_LAYOUT`).

An exception from the rule may be applied to a vendor-specific flash dedicated-purpose region (such a region obviously can't be covered under API for retrieving the layout of pages).

User API Reference
Flash

Related code samples

- Ambiq MSPI timing scanFind the appropriate timing for a given device on a given board.
- AT45 DataFlash driverUse the AT45 family DataFlash driver to interact with the flash memory over SPI.
- B-M2MEM-PACK1 shieldCheck a B-M2MEM-PACK1 memory module against the data stored in its identification EEPROM.
- Flash shellExplore a flash device using shell commands.
- JEDEC MSPI-NOR flashUse the flash API to interact with an MSPI NOR serial flash memory device.
- JEDEC SPI-NOR flashUse the flash API to interact with an SPI NOR serial flash memory device.
- JESD216 flashUse the JESD216 flash API to extract information from a compatible serial memory device.
- Memory-Mapped FlashWrite data into the storage partition and read it using flash API and memory-mapped pointer.
- Nordic flash write during Bluetooth LE activityWrite to flash while the Bluetooth LE radio is active.
- nRF SoC Internal StorageUse the flash API to interact with the SoC flash.
- RW612 PM flash checkCheck external NOR flash health across RW612 low-power modes.

- X-NUCLEO-PGEEZ1 M95P32 extended operationsAccess the M95P32 identification pages and buffered page programming.

Implementation interface API Reference

Flash Internal

Electrically Erasable Programmable Read-Only Memory (EEPROM)

Overview EEPROMs have an erase block size of 1 byte, a long lifetime, and allow overwriting data on byte-by-byte access.

EEPROM API

Overview The EEPROM API provides read and write access to Electrically Erasable Programmable Read-Only Memory (EEPROM) devices.

Configuration Options Related configuration options:

- `CONFIG_EEPROM`

API Reference

EEPROM

Related code samples

- B-M2MEM-PACK1 shieldCheck a B-M2MEM-PACK1 memory module against the data stored in its identification EEPROM.
- EEPROMStore a boot count value in EEPROM.

Fake EEPROM

EEPROM Shell

- *Overview*
- *EEPROM Size*
- *Writing Data*
- *Reading Data*

Overview The EEPROM shell provides an `eeprom` command with a set of subcommands for the *shell* module. It allows testing and exploring the *EEPROM* driver API through an interactive interface without having to write a dedicated application. The EEPROM shell can also be enabled in existing applications to aid in interactive debugging of EEPROM issues.

In order to enable the EEPROM shell, the following *Kconfig* options must be enabled:

- `CONFIG_SHELL`
- `CONFIG_EEPROM`
- `CONFIG_EEPROM_SHELL`

For example, building the hello_world sample for the native_sim with the EEPROM shell:

```
# From the root of the zephyr repository
west build -b native_sim samples/hello_world -- -DCONFIG_SHELL=y -DCONFIG_EEPROM=y␣
↪-DCONFIG_EEPROM_SHELL=y
```

See the *shell* documentation for general instructions on how to connect and interact with the shell. The EEPROM shell comes with built-in help (unless `CONFIG_SHELL_HELP` is disabled). The built-in help messages can be printed by passing `-h` or `--help` to the `eeprom` command or any of its subcommands. All subcommands also support tab-completion of their arguments.

> **Tip**
>
> All of the EEPROM shell subcommands take the name of an EEPROM peripheral as their first argument, which also supports tab-completion. A list of all devices available can be obtained using the `device list` shell command when `CONFIG_DEVICE_SHELL` is enabled. The examples below all use the device name `eeprom@0`.

EEPROM Size The size of an EEPROM can be inspected using the `eeprom size` subcommand as shown below:

```
uart:~$ eeprom size eeprom@0
32768 bytes
```

Writing Data Data can be written to an EEPROM using the `eeprom write` subcommand. This subcommand takes at least three arguments; the EEPROM device name, the offset to start writing to, and at least one data byte. In the following example, the hexadecimal sequence of bytes `0x0d 0x0e 0x0a 0x0d 0x0b 0x0e 0x0e 0x0f` is written to offset `0x0`:

```
uart:~$ eeprom write eeprom@0 0x0 0x0d 0x0e 0x0a 0x0d 0x0b 0x0e 0x0e 0x0f
Writing 8 bytes to EEPROM...
Verifying...
Verify OK
```

It is also possible to fill a portion of the EEPROM with the same pattern using the `eeprom fill` subcommand. In the following example, the pattern `0xaa` is written to 16 bytes starting at offset `0x8`:

```
uart:~$ eeprom fill eeprom@0 0x8 16 0xaa
Writing 16 bytes of 0xaa to EEPROM...
Verifying...
Verify OK
```

Reading Data Data can be read from an EEPROM using the `eeprom read` subcommand. This subcommand takes three arguments; the EEPROM device name, the offset to start reading from, and the number of bytes to read:

```
uart:~$ eeprom read eeprom@0 0x0 8
Reading 8 bytes from EEPROM, offset 0...
00000000: 0d 0e 0a 0d 0b 0e 0e 0f                          |........         |
```

Secure Digital (SD card) interface

Zephyr can communicate with an attached SD card either using a system's native SD card interface or over SPI (Serial Peripheral Interface). Some devices can also communicate with MMC (MultiMediaCard) devices.

Applications can use Zephyr's *the disk access API* to use SD cards as storage devices, or Zephyr's SD card subsystem to directly read from and write to a card.

SD Host Controller (SDHC) An SD host controller (SDHC) is a device capable of sending commands to an SD card. These commands can be sent using a system's native SD card interface, or over SPI.

Applications should generally not use the SD host controller API directly, instead they should use Zephyr's SD card subsystem.

Requests The core of the SD host controller (SDHC) API is the `sdhc_request()` API. Requests contain a `sdhc_command` command structure, and an optional `sdhc_data` data structure. The caller may check the return code, or the `response` field of the SD command structure to determine if the SDHC request succeeded. The data structure allows the caller to specify a number of blocks to transfer, and a buffer location to read or write them from. Whether the provided buffer is used for sending or reading data depends on the command opcode provided.

Host Controller I/O The `sdhc_set_io()` API allows the user to change I/O settings of the SD host controller, such as clock frequency, I/O voltage, and card power. Not all controllers will support applying all I/O settings. For example, SPI mode controllers typically cannot toggle power to the SD card.

Related configuration options:

- `CONFIG_SDHC`

API Reference
SDHC

Battery Backed RAM (BBRAM)

The BBRAM APIs allow interfacing with the unique properties of this memory region. The following common types of BBRAM properties are easily accessed via this API:

- IBBR (invalid) state - check that the BBRAM is not corrupt.
- VSBY (voltage standby) state - check if the BBRAM is using standby voltage.
- VCC (active power) state - check if the BBRAM is on normal power.
- Size - get the size (in bytes) of the BBRAM region.

Along with these, the API provides a means for reading and writing to the memory region via `bbram_read()` and `bbram_write()` respectively. Both functions are expected to only succeed if the BBRAM is in a valid state and the operation is bounded to the memory region.

API Reference
BBRAM

Retained Memory

Overview The retained memory driver API provides a way of reading from/writing to memory areas whereby the contents of the memory is retained whilst the device is powered (data may be lost in low power modes).

Configuration Options Related configuration options:

- CONFIG_RETAINED_MEM
- CONFIG_RETAINED_MEM_INIT_PRIORITY
- CONFIG_RETAINED_MEM_MUTEX_FORCE_DISABLE

Mutex protection Mutex protection of retained memory drivers is enabled by default when applications are compiled with multithreading support. This means that different threads can safely call the retained memory functions without clashing with other concurrent thread function usage, but means that retained memory functions cannot be used from ISRs. It is possible to disable mutex protection globally on all retained memory drivers by enabling CONFIG_RETAINED_MEM_MUTEX_FORCE_DISABLE - users are then responsible for ensuring that the function calls do not conflict with each other.

API Reference
Retained memory

I2C EEPROM Target

Overview

API Reference
I2C EEPROM Target

Memory Controller (MEMC)

Overview The MEMC API provides a generic interface for accessing external memory devices such as PSRAM and NOR flash. It supports two access modes:

- **Memory-mapped:** The controller exposes the device through a direct CPU-accessible address window. The driver may implement memc_driver_api.get_mem_base to advertise the mapped base address, allowing memc_read() and memc_write() to use memcpy automatically. Alternatively, the driver may provide no runtime API at all if the hardware maps memory transparently - in this case upper layers access the device directly using a platform-specific base address.
- **Bus transaction:** The controller issues explicit bus commands (e.g., MSPI) for each transfer. The driver implements memc_driver_api.read and memc_driver_api.write. Used when no memory-mapped aperture exists on the controller.

memc_read() and memc_write() automatically select the memory-mapped path when available, falling back to bus transactions otherwise.

Additional optional APIs are provided for device introspection: memc_get_size() returns the device capacity and memc_read_id() returns device identification bytes.

Legacy (Init-Only) Drivers Existing memc drivers that pass NULL as the API pointer to DEVICE_DT_INST_DEFINE are not affected by this API. Callers should use DEVICE_API_IS to test whether a device implements the memc interface before calling any memc function:

```
if (DEVICE_API_IS(memc, dev)) {
   memc_read(dev, offset, buf, sizeof(buf));
} else {
   /* Legacy init-only driver - device is memory-mapped.
    * Access via platform-specific base address.
    */
```

(continues on next page)

(continued from previous page)

```
    memcpy(buf, (const uint8_t *)MEMC_BASE + offset, sizeof(buf));
}
```

Calling `memc_read()` or `memc_write()` on a device that does not implement the memc API returns `-ENOTSUP`.

Configuration Options Related configuration options:

- `CONFIG_MEMC`
- `CONFIG_MEMC_INIT_PRIORITY`

API Reference
MEMC

One Time Programmable (OTP) memory devices

Overview OTP memory devices handle memory that is expected to be permanent. This memory is usually provisioned during manufacturing, but there are some use-cases in which one needs to program the OTP memory at a later lifecycle state (e.g: monotonic counters, cryptographic keys, etc...).

OTP API

Overview The OTP API provides means to provision and read OTP memory devices

API implementation Reference
One Time Programmable Memory

Configuration Options OTP related configuration options:

- `CONFIG_OTP`
- `CONFIG_OTP_PROGRAM`
- `CONFIG_OTP_INIT_PRIORITY`
- `CONFIG_OTP_SHELL.*`

Power

6.7.7 Power

Power management, charging, regulators, reset, and wake-up controllers.

BC1.2 Devices (Experimental)

The Battery Charging specification, currently at revision 1.2, is commonly referred to as just BC1.2. BC1.2 defines limits and detection mechanisms for USB devices to draw current exceeding the USB 2.0 specification limit of 0.5A, 2.5W.

The BC1.2 specification uses the term Charging Port for the device that supplies VBUS on the USB connection and the term Portable Device for the device that sinks current from the USB connection.

Note that the BC1.2 Specification uses the acronym PD for Portable Device. This should not be confused with the USB-C Power Delivery, which also uses the acronym PD.

On many devices, BC1.2 is the fallback mechanism to determine the connected charger capability on USB type C ports when the attached type-C partner does not support Power Delivery.

Key parameters from the BC1.2 Specification include:

Parameter	Symbol	Range
Allowed PD (portable device) Current Draw from Charging Port	IDEV_CHG	1.5 A
Charging Downstream Port Rated Current	ICDP	1.5 - 5.0 A
Maximum Configured Current when connected to a SDP	ICFG_MAX	500 mA
Dedicated Charging Port Rated Current	IDCP	1.5 - 5.0 A
Suspend current	ISUSP	2.5 mA
Unit load current	IUNIT	100 mA

While the ICDP and IDCP rated currents go up to 5.0 A, the BC1.2 current is limited by the IDEV_CHG parameter. So the BC1.2 support is capped at 1.5 A in the Zephyr implementation when using portable-device mode.

Basic Operation The BC1.2 device driver provides only two APIs, `bc12_set_role()` and `bc12_set_result_cb()`.

The application calls `bc12_set_role()` to transition the BC1.2 device to either a disconnected, portable-device, or charging port mode.

For the disconnected state, the BC1.2 driver powers down the detection chip. The power down operation is vendor specific.

The application calls `bc12_set_role()` with the type set to BC12_PORTABLE_DEVICE when both the following conditions are true:

- The application configured the port as an upstream facing port, i.e. a USB device port.
- The application detects VBUS on the USB connection.

For portable-device mode, the BC1.2 driver powers up the detection chip and starts charger detection. At completion of the charger detection, the BC1.2 driver notifies the callback registered with `bc12_set_result_cb()`. By default, the BC1.2 driver clamps the current to 1.5A to comply with the BC1.2 specification.

To comply with the USB 2.0 specification, when the driver detects a SDP (Standard Downstream Port) charging partner or if BC1.2 detection fails, the driver reports the available current as ISUSP (2.5 mA). The application may increase the current draw to IUNIT (100 mA) when the connected USB host resumes the USB bus and may increase the current draw to ICFG_MAX (500 mA) when the USB host configures the USB device.

Charging port mode is used by the application when the USB port is configured as a downstream facing port, i.e. a USB host port. For charging port mode, the BC1.2 driver powers up the detection chip and configures the charger type specified by a devicetree property. If the driver supports detection of plug and unplug events, the BC1.2 driver notifies the callback registered with `bc12_set_result_cb()` to indicate the current connection state of the portable device partner.

Configuration Options Related configuration options:

- `CONFIG_USB_BC12`

API Reference

Battery Charging (BC1.2)

BC1.2 backed emulator APIs

Chargers

The charger subsystem exposes an API to uniformly access battery charger devices.

A charger device, or charger peripheral, is a device used to take external power provided to the system as an input and provide power as an output downstream to the battery pack(s) and system. The charger device can exist as a module, an integrated circuit, or as a functional block in a power management integrated circuit (PMIC).

The action of charging a battery pack is referred to as a charge cycle. When the charge cycle is executed the battery pack is charged according to the charge profile configured on the charger device. The charge profile is defined in the battery pack's specification that is provided by the manufacturer. On charger devices with a control port, the charge profile can be configured by the host controller by setting the relevant properties, and can be adjusted at runtime to respond to environmental changes.

Basic Operation

Initiating a Charge Cycle A charge cycle is initiated or terminated using `charger_charge_enable()`.

Properties Fundamentally, a property is a configurable setting, state, or quantity that a charger device can measure.

Chargers typically support multiple properties, such as temperature readings of the battery-pack or present-time current/voltage.

Properties are fetched by the client one at a time using `charger_get_prop()`. Properties are set by the client one at a time using `charger_set_prop()`.

API Reference

Battery Charger

Related code samples

- ChargerCharge a battery using the charger driver API.
- Multi-Function Device (MFD) with Charger, Fuel Gauge, and RegulatorsDemonstrate a multi-function device using charger, fuel gauge, and regulator subsystems.

Fuel Gauge

The fuel gauge subsystem exposes an API to uniformly access battery fuel gauge devices.

Basic Operation

Properties Fundamentally, a property is a quantity that a fuel gauge device can measure.

Fuel gauges typically support multiple properties, such as temperature readings of the battery-pack or present-time current/voltage.

Properties are fetched by the client one at a time using `fuel_gauge_get_prop()`, or fetched in a batch using `fuel_gauge_get_props()`. Buffer properties, e.g. device name, are fetched by using `fuel_gauge_get_buffer_prop()`.

Properties are set by the client one at a time using `fuel_gauge_set_prop()`, or set in a batch using `fuel_gauge_set_props()`. Buffer properties, e.g. a battery configuration image, are set by using `fuel_gauge_set_buffer_prop()`.

Battery Cutoff Many fuel gauges embedded within battery packs expose a register address that when written to with a specific payload will do a battery cutoff. This battery cutoff is often referred to as ship, shelf, or sleep mode due to its utility in reducing battery drain while devices are stored or shipped.

The fuel gauge API exposes battery cutoff with the `fuel_gauge_battery_cutoff()` function.

Caching The Fuel Gauge API explicitly provides no caching for its clients.

API Reference
Fuel Gauge

Related code samples

- Multi-Function Device (MFD) with Charger, Fuel Gauge, and RegulatorsDemonstrate a multi-function device using charger, fuel gauge, and regulator subsystems.

Fuel gauge backend emulator APIs

Regulators

This subsystem provides control of voltage and current regulators. A common example is a GPIO that controls a transistor that supplies current to a device that is not always needed. Another example is a PMIC, typically a much more complex device.

The `*-supply` devicetree properties are used to identify the regulator(s) that a devicetree node directly depends on. Within the driver for the node the regulator API is used to issue requests for power when the device is to be active, and release the power request when the device shuts down.

The simplest case where a regulator is needed is one where there is only one client. For those situations the cost of using the regulator device infrastructure is not justified, and `*-gpios` devicetree properties should be used. There is no device interface to these regulators as they are entirely controlled within the driver for the corresponding node, e.g. a sensor.

API Reference
Regulator

Related code samples

- Multi-Function Device (MFD) with Charger, Fuel Gauge, and RegulatorsDemonstrate a multi-function device using charger, fuel gauge, and regulator subsystems.

Fake regulator

Reset Controller

Overview Reset controllers are units that control the reset signals to multiple peripherals. The reset controller API allows peripheral drivers to request control over their reset input signals, including the ability to assert, deassert and toggle those signals. Also, the reset status of the reset input signal can be checked.

Mainly, the line_assert and line_deassert API functions are optional because in most cases we want to toggle the reset signals.

Configuration Options Related configuration options:

- `CONFIG_RESET`

API Reference
Reset Controller

USB Type-C Port Controller (TCPC)

Overview TCPC (USB Type-C Port Controller) The TCPC is a device used to simplify the implementation of a USB-C system by providing the following three function:

- VBUS and VCONN control USB Type-C: The TCPC may provide a Source device, the mechanism to control VBUS sourcing, and a Sink device, the mechanism to control VBUS sinking. A similar mechanism is provided for the control of VCONN.
- CC control and sensing: The TCPC implements logic for controlling the CC pin pull-up and pull-down resistors. It also provides a way to sense and report what resistors are present on the CC pin.
- Power Delivery message reception and transmission USB Power Delivery: The TCPC sends and receives messages constructed in the TCPM and places them on the CC lines.

TCPC API The TCPC device driver functions as the liaison between the TCPC device and the application software; this is accomplished by the Zephyr's API provided by the device driver that's used to communicate with and control the TCPC device.

Configuration Options Related configuration options:

- `CONFIG_USBC_TCPC_DRIVER`

API Reference
USB Type-C

USB Type-C Port Controller API

USB Power Delivery

USB-C VBUS

Overview USB-C VBUS is the line in a USB Type-C connection that delivers power from a Source to a Sink device.

USB-C VBUS API The USB-C VBUS device driver presents an API that's used to control and measure VBUS.

Configuration Options Related configuration options:

- `CONFIG_USBC_VBUS_DRIVER`

API Reference
USB-C VBUS API

Wakeup Controller (WUC)

Overview The Wakeup Controller (WUC) API provides a common interface for enabling and managing wakeup sources that can bring the system out of low-power states. WUC devices are typically described in Devicetree and referenced by clients using `wuc_dt_spec`.

Devicetree Configuration Wakeup controllers are referenced from client nodes with the `wakeup-ctrls` property. The property encodes a phandle to the WUC device and an identifier for the wakeup source.

Example Devicetree fragment:

```
wuc0: wakeup-controller@40000000 {
    compatible = "nxp,mcx-wuc";
    reg = <0x40000000 0x1000>;
    #wakeup-ctrl-cells = <1>;
};

button0: button@0 {
    wakeup-ctrls = <&wuc0 10>;
};
```

Basic Operation Applications typically obtain a `wuc_dt_spec` using `WUC_DT_SPEC_GET` and then enable or disable the wakeup source as needed.

Listing 4: Enable a wakeup source from Devicetree

```
#define BUTTON0_NODE DT_NODELABEL(button0)

static const struct wuc_dt_spec button_wuc =
    WUC_DT_SPEC_GET(BUTTON0_NODE);

if (!device_is_ready(button_wuc.dev)) {
    return -ENODEV;
}

return wuc_enable_wakeup_source_dt(&button_wuc);
```

If a driver supports it, applications can check and clear a wakeup source's triggered state. When not implemented, the APIs return `-ENOSYS`.

Listing 5: Check and clear a wakeup source

```
int ret;

ret = wuc_check_wakeup_source_triggered_dt(&button_wuc);
if (ret > 0) {
    (void)wuc_clear_wakeup_source_triggered_dt(&button_wuc);
}
```

API Reference

WUC

Related code samples

- NXP MCX suspend-to-RAMSuspend an NXP MCXA/MCXN SoC to RAM and resume on an LPTMR timer (or button) wakeup delivered through the WUU wakeup controller.
- NXP MCX system offUse system off (Deep Power Down) on NXP MCXA/MCXN SoCs and wake through the WUU wakeup controller.

Sensors

6.7.8 Sensors

Environmental and motion sensors, GNSS, biometrics, and sensor interfaces.

Sensors

The sensor driver API provides functionality to uniformly read, configure, and setup event handling for devices that take real world measurements in meaningful units.

Sensors range from very simple temperature-reading devices that must be polled with a fixed scale to complex devices taking in readings from multitudes of sensors and themselves producing new inferred sensor data such as step counts, presence detection, orientation, and more.

Supporting this wide breadth of devices is a demanding task and the sensor API attempts to provide a uniform interface to them.

Using Sensors Using sensors from an application there are some APIs and terms that are helpful to understand. Sensors in Zephyr are composed of *Sensor Channels*, *Sensor Attributes*, and *Sensor Triggers*. Attributes and triggers may be device or channel specific.

> **Note**
>
> Getting samples from sensors using the sensor API today can be done in one of two ways. A stable and long-lived API *Fetch and Get*, or a newer but rapidly stabilizing API *Read and Decode*. It's expected that in the near future *Fetch and Get* will be deprecated in favor of *Read and Decode*. Triggers are handled entirely differently for *Fetch and Get* or *Read and Decode* and the differences are noted in each of those sections.

Sensor Attributes *Attributes*, enumerated in `sensor_attribute`, are immutable and mutable properties of a sensor and its channels.

Attributes allow for obtaining metadata and changing configuration of a sensor. Common configuration parameters like channel scale, sampling frequency, adjusting channel offsets, signal filtering, power modes, on chip buffers, and event handling options are very common. Attributes provide a flexible API for inspecting and manipulating such device properties.

Attributes are specified using `sensor_attribute` which can be used with `sensor_attr_get()` and `sensor_attr_set()` to get and set a sensors attributes.

A quick example...

```
const struct device *accel_dev = DEVICE_DT_GET(DT_ALIAS(accel0));
struct sensor_value accel_sample_rate;
int rc;

rc = sensor_attr_get(accel_dev, SENSOR_CHAN_ACCEL_XYZ,␣
↪SENSOR_ATTR_SAMPLING_FREQUENCY, &accel_sample_rate);
if (rc != 0) {
            printk("Failed to get sampling frequency\n");
}

printk("Sample rate for accel %p is %d.06%d\n", accel_dev, accel_sample_rate.val1,␣
↪accel_sample_rate.val2*1000000);

accel_sample_rate.val1 = 2000;

rc = sensor_attr_set(accel_dev, SENSOR_CHAN_ACCEL_XYZ,␣
↪SENSOR_ATTR_SAMPLING_FREQUENCY, accel_sample_rate);
```

(continues on next page)

(continued from previous page)

```
if (rc != 0) {
            printk("Failed to set sampling frequency\n");
}
```

Sensor Channels *Channels*, enumerated in `sensor_channel`, are quantities that a sensor device can measure.

Sensors may have multiple channels, either to represent different axes of the same physical property (e.g. acceleration); or because they can measure different properties altogether (ambient temperature, pressure and humidity). Sensors may have multiple channels of the same measurement type to enable measuring many readings of perhaps temperature, light intensity, amperage, voltage, or capacitance for example.

A channel is specified in Zephyr using a `sensor_chan_spec` which is a pair containing both the channel type (`sensor_channel`) and channel index. At times only `sensor_channel` is used but this should be considered historical since the introduction of `sensor_chan_spec` for Zephyr 3.7.

Sensor Triggers *Triggers*, enumerated in `sensor_trigger_type`, are sensor generated events. Typically sensors allow setting up these events to cause digital line signaling for easy capture by a microcontroller. The events can then commonly be inspected by reading registers to determine which event caused the digital line signaling to occur.

There are many kinds of triggers sensors provide, from informative ones such as data ready to physical events such as taps or steps.

Power Management Power management of sensors is often a non-trivial task as sensors may have multiple power states for various channels. Some sensors may allow for low noise, low power, or suspending channels potentially saving quite a bit of power at the cost of noise or sampling speed performance. In very low power states sensors may lose their state, turning off even the digital logic portion of the device.

All this is to say that power management of sensors is typically application specific! Often the channel states are mutable using *Sensor Attributes*. While total device suspending and resume can be done using the power management ref counting APIs if the device implements the necessary functionality.

Most likely the API sensors should use for their fully suspended/resume power states is *Device Runtime Power Management* using explicit calls at an application level to `pm_device_runtime_get()` and `pm_device_runtime_put()`.

In the future, with *Read and Decode* its possible that automatic management of device power management would be possible in the streaming case as the application informs the driver of usage at all times through requests to read on given events.

Device Tree In the context of sensors device tree provides the initial hardware configuration for sensors on a per device level. Each device must specify a device tree binding in Zephyr, and ideally, a set of hardware configuration options for things such as channel power modes, data rates, filters, decimation, and scales. These can then be used in a boards devicetree to configure a sensor to its initial state.

```
#include <zephyr/dt-bindings/icm42688.h>

&spi0 {
    /* SPI bus options here, not shown */

    accel_gyro0: icm42688p@0 {
        compatible = "invensense,icm42688", "invensense,icm4268x";
```

(continues on next page)

(continued from previous page)

```
        reg = <0>;
        int-gpios = <&pioc 6 GPIO_ACTIVE_HIGH>; /* SoC specific pin to select for␣
↪interrupt line */
        spi-max-frequency = <DT_FREQ_M(24)>; /* Maximum SPI bus frequency */
        accel-pwr-mode = <ICM42688_ACCEL_LN>; /* Low noise mode */
        accel-odr = <ICM42688_ACCEL_ODR_2000>; /* 2000 Hz sampling */
        accel-fs = <ICM42688_ACCEL_FS_16>; /* 16G scale */
        gyro-pwr-mode = <ICM42688_GYRO_LN>; /* Low noise mode */
        gyro-odr = <ICM42688_GYRO_ODR_2000>; /* 2000 Hz sampling */
        gyro-fs = <ICM42688_GYRO_FS_16>; /* 16G scale */
    };
};
```

Fetch and Get The stable and long existing APIs for reading sensor data and handling triggers are:

- `sensor_sample_fetch()`
- `sensor_sample_fetch_chan()`
- `sensor_channel_get()`
- `sensor_trigger_set()`

These functions work together. The fetch APIs block the calling context which must be a thread until the requested `sensor_channel` (or all channels) has been obtained and stored into the driver instance's private data.

The channel data most recently fetched can then be obtained as a `sensor_value` by calling `sensor_channel_get()` for each channel type.

Warning

It should be noted that calling fetch and get from multiple contexts without a locking mechanism is undefined and most sensor drivers do not attempt to internally provide exclusive access to the device during or between these calls.

Polling Using fetch and get sensor can be read in a polling manner from software threads.

```
/*
 * Copyright (c) 2016 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/kernel.h>
#include <zephyr/device.h>
#include <zephyr/drivers/sensor.h>
#include <zephyr/sys/printk.h>
#include <stdio.h>

int main(void)
{
        const struct device *const dev = DEVICE_DT_GET(DT_ALIAS(magn0));
        struct sensor_value value_x, value_y, value_z;
        int ret;
```

(continues on next page)

(continued from previous page)

```
        if (!device_is_ready(dev)) {
                printk("sensor: device not ready.\n");
                return 0;
        }

        printk("Polling magnetometer data from %s.\n", dev->name);

        while (1) {
                ret = sensor_sample_fetch(dev);
                if (ret) {
                        printk("sensor_sample_fetch failed ret %d\n", ret);
                        return 0;
                }

                ret = sensor_channel_get(dev, SENSOR_CHAN_MAGN_X, &value_x);
                ret = sensor_channel_get(dev, SENSOR_CHAN_MAGN_Y, &value_y);
                ret = sensor_channel_get(dev, SENSOR_CHAN_MAGN_Z, &value_z);
                printf("( x y z ) = ( %f  %f  %f )\n",
                       sensor_value_to_double(&value_x),
                       sensor_value_to_double(&value_y),
                       sensor_value_to_double(&value_z));

                k_sleep(K_MSEC(500));
        }
        return 0;
}
```

Triggers Triggers in the stable API require enabling triggers with a device specific Kconfig. The device specific Kconfig typically allows selecting the context the trigger runs. The application then needs to register a callback with a function signature matching `sensor_trigger_handler_t` using `sensor_trigger_set()` for the specific triggers (events) to listen for.

> **Note**
>
> Triggers may not be set from user mode threads, and the callback is not run in a user mode context.

There are typically two options provided for each driver where to run trigger handlers. Either the trigger handler is run using the system work queue thread (*Workqueue Threads*) or a dedicated thread. A great example can be found in the BMI160 driver which has Kconfig options for selecting a trigger mode. See `CONFIG_BMI160_TRIGGER_NONE`, `CONFIG_BMI160_TRIGGER_GLOBAL_THREAD` (work queue), `CONFIG_BMI160_TRIGGER_OWN_THREAD` (dedicated thread).

Some notable attributes of using a driver dedicated thread vs the system work queue.

- Driver dedicated threads have dedicated stack (RAM) which only gets used for that single trigger handler function.
- Driver dedicated threads *do* get their own priority typically which lets you prioritize trigger handling among other threads.
- Driver dedicated threads will not have head of line blocking if the driver needs time to handle the trigger.

> **Note**
>
> In all cases it's very likely there will be variable delays from the actual interrupt to your callback function being run. In the work queue (GLOBAL_THREAD) case the work queue itself can be the source of variable latency!

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>


const struct device *const accel0 = DEVICE_DT_GET(DT_ALIAS(accel0));

static struct tap_count_state {
        struct sensor_trigger trig;
        uint32_t count;
} tap_count_state = {
        .trig = {
                .chan = SENSOR_CHAN_ACCEL_XYZ,
                .type = SENSOR_TRIG_TAP,
        },
        .count = 0,
};

void tap_handler(const struct device *dev, const struct sensor_trigger *trig)
{
        struct tap_count_state *state = CONTAINER_OF(trig, struct tap_count_state,␣
↪trig);

        state->count++;

        printk("Tap! Total Taps: %u\n", state->count);
}

int main(void)
{
        int rc;

        printk("Tap Counter Example (%s)\n", CONFIG_ARCH);

        rc = sensor_trigger_set(accel0, &tap_count_state.trig, tap_handler);

        if (rc != 0) {
                printk("Failed to set trigger handler for taps, error %d\n", rc);
                return rc;
        }

        return rc;
}
```

Read and Decode The quickly stabilizing APIs for reading sensor data are:

- `sensor_read()`
- `sensor_read_async_mempool()`
- `sensor_get_decoder()`
- `sensor_decode()`

Benefits over *Fetch and Get* These APIs allow for a wider usage of sensors, sensor types, and data flows with sensors. These are the future looking APIs in Zephyr and solve many issues that have been run into with *Fetch and Get*.

`sensor_read()` and similar functions acquire sensor encoded data into a buffer provided by the caller. Decode (`sensor_decode()`) then decodes the sensor specific encoded data into fixed point `q31_t` values as vectors per channel. This allows further processing using fixed point DSP functions that work on vectors of data to be done (e.g. low-pass filters, FFT, fusion, etc).

Reading is by default asynchronous in its implementation and takes advantage of *Real Time I/O (RTIO)* to enable chaining asynchronous requests, or starting requests against many sensors simultaneously from a single call context.

This enables incredibly useful code flows when working with sensors such as:

- Obtaining the raw sensor data, decoding never, later, or on a separate processor (e.g. a phone).
- Starting a read for sensors directly from an interrupt handler. No dedicated thread needed saving precious stack space. No work queue needed introducing variable latency. Starting a read for multiple sensors simultaneously from a single call context (interrupt/thread/work queue).
- Requesting multiple reads to the same device for Ping-Pong (double buffering) setups.
- Creating entire pipelines of data flow from sensors allowing for software defined virtual sensors (*Sensing Subsystem*) all from a single thread with DAG process ordering.
- Potentially pre-programming DMAs to trigger on GPIO events, leaving the CPU entirely out of the loop in handling sensor events like FIFO watermarks.

Additionally, other shortcomings of *Fetch and Get* related to memory and trigger handling are solved.

- Triggers result in enqueued events, not callbacks.
- Triggers can be setup to automatically fetch data. Potentially enabling pre-programmed DMA transfers on GPIO interrupts.
- Far less likely triggers are missed due to long held interrupt masks from callbacks and context swapping.
- Sensor FIFOs supported by wiring up FIFO triggers to read data into mempool allocated buffers.
- All sensor processing can be done in user mode (memory protected) threads.
- Multiple sensor channels of the same type are better supported.

Note

For *Read and Decode* benefits to be fully realized requires *Real Time I/O (RTIO)* compliant communication access to the sensor. Typically this means an *Real Time I/O (RTIO)* enabled bus driver for SPI or I2C.

Polling Read Polling reads with *Read and Decode* can be accomplished by instantiating a polling I/O device (akin to a file descriptor) for the sensor with the desired channels to poll. Requesting either blocking or non-blocking reads, then optionally decoding the data into fixed point values.

Polling a temperature sensor and printing its readout is likely the simplest sample to show how this all works.

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>

#define TEMP_CHANNEL {SENSOR_CHAN_AMBIENT_TEMP, 0}

const struct device *const temp0 = DEVICE_DT_GET(DT_ALIAS(temp0));

SENSOR_DT_READ_IODEV(temp_iodev, DT_ALIAS(temp0), TEMP_CHANNEL);
RTIO_DEFINE(temp_ctx, 1, 1);

int main(void)
{
        int rc;
        uint8_t buf[8];
        struct sensor_q31_data temp_data = {0};
        struct sensor_decode_context temp_decoder = SENSOR_DECODE_CONTEXT_INIT(
                SENSOR_DECODER_DT_GET(DT_ALIAS(temp0)), buf,␣
↪SENSOR_CHAN_AMBIENT_TEMP, 0);

        while (1) {
                /* Blocking read */
                rc = sensor_read(temp_iodev, &temp_ctx, buf, sizeof(buf));

                if (rc != 0) {
                        printk("sensor_read() failed %d\n", rc);
                }

                /* Decode the data into a single q31 */
                sensor_decode(&temp_decoder, &temp_data, 1);

                printk("Temperature " PRIsensor_q31_data "\n",
                       PRIsensor_q31_data_arg(temp_data, 0));

                k_msleep(1);
        }
}
```

Polling Read with Multiple Sensors One of the benefits of Read and Decode is the ability to concurrently read many sensors with many channels in one thread. Effectively read requests are started asynchronously for all sensors and their channels. When each read completes we then decode the sensor data. Examples speak loudly and so a sample showing how this might work with multiple temperature sensors with multiple temperature channels:

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>

#define TEMP_CHANNELS                                   \
        { SENSOR_CHAN_AMBIENT_TEMP, 0 },        \
        { SENSOR_CHAN_AMBIENT_TEMP, 1 }
#define TEMP_ALIAS(id) DT_ALIAS(CONCAT(temp, id))
#define TEMP_IODEV_SYM(id) CONCAT(temp_iodev, id)
#define TEMP_IODEV_PTR(id) &TEMP_IODEV_SYM(id)
#define TEMP_DEFINE_IODEV(id)                           \
        SENSOR_DT_READ_IODEV(                           \
                TEMP_IODEV_SYM(id),                 \
                TEMP_ALIAS(id),                         \
                TEMP_CHANNELS                           \
                );

#define NUM_SENSORS 2

LISTIFY(NUM_SENSORS, TEMP_DEFINE_IODEV, (;));

struct sensor_iodev *iodevs[NUM_SENSORS] = { LISTIFY(NUM_SENSORS, TEMP_IODEV_PTR,␣
↪(,)) };

RTIO_DEFINE_WITH_MEMPOOL(temp_ctx, NUM_SENSORS, NUM_SENSORS, NUM_SENSORS, 8,␣
↪sizeof(void *));

int main(void)
{
        int rc;
        uint32_t temp_frame_iter = 0;
        struct sensor_q31_data temp_data[2] = {0};
        struct sensor_decoder_api *decoder;
        struct rtio_cqe *cqe;
        uint8_t *buf;
        uint32_t buf_len;

        while (1) {
                /* Non-Blocking read for each sensor */
                for (int i = 0; i < NUM_SENSORS; i++) {
                        rc = sensor_read_async_mempool(iodevs[i], &temp_ctx,␣
↪iodevs[i]);

                        if (rc != 0) {
                                printk("sensor_read() failed %d\n", rc);
                                return;
                        }
                }

                /* Wait for read completions */
                for (int i = 0; i < NUM_SENSORS; i++) {
                        cqe = rtio_cqe_consume_block(&temp_ctx);
```

(continues on next page)

(continued from previous page)

```
                        if (cqe->result != 0) {
                                printk("async read failed %d\n", cqe->result);
                                return;
                        }

                        /* Get the associated mempool buffer with the completion */
                        rc = rtio_cqe_get_mempool_buffer(&temp_ctx, cqe, &buf, &
↪buf_len);

                        if (rc != 0) {
                                printk("get mempool buffer failed %d\n", rc);
                                return;
                        }

                        struct device *sensor = ((struct sensor_read_config *)
                                ((struct rtio_iodev *)cqe->userdata)->data)->
↪sensor;

                        /* Done with the completion event, release it */
                        rtio_cqe_release(&temp_ctx, cqe);

                        rc = sensor_get_decoder(sensor, &decoder);
                        if (rc != 0) {
                                printk("sensor_get_decoder failed %d\n", rc);
                                return;
                        }

                        /* Frame iterators, one per channel we are decoding */
                        uint32_t temp_fits[2] = { 0, 0 };

                        decoder->decode(buf, {SENSOR_CHAN_AMBIENT_TEMP, 0},
                                        &temp_fits[0], 1, &temp_data[0]);
                        decoder->decode(buf, {SENSOR_CHAN_AMBIENT_TEMP, 1},
                                        &temp_fits[1], 1, &temp_data[1]);

                        /* Done with the buffer, release it */
                        rtio_release_buffer(&temp_ctx, buf, buf_len);

                        printk("Temperature for %s channel 0 " PRIsensor_q31_data
↪", channel 1 "
                            PRIsensor_q31_data "\n",
                            dev->name,
                            PRIsensor_q31_data_arg(temp_data[0], 0),
                            PRIsensor_q31_data_arg(temp_data[1], 0));
                }
        }

        k_msleep(1);
}
```

Streaming Handling triggers with *Read and Decode* works by setting up a stream I/O device configuration. A stream specifies the set of triggers to capture and if data should be captured with the event.

```
/*
 * Copyright (c) 2024 Intel Corporation
 *
 * SPDX-License-Identifier: Apache-2.0
 */

#include <zephyr/drivers/sensor.h>

#define ACCEL_TRIGGERS                                                  \
        { SENSOR_TRIG_DRDY, SENSOR_STREAM_DATA_INCLUDE },        \
        { SENSOR_TRIG_TAP, SENSOR_STREAM_DATA_NOP }
#define ACCEL_ALIAS(id) DT_ALIAS(CONCAT(accel, id))
#define ACCEL_IODEV_SYM(id) CONCAT(accel_iodev, id)
#define ACCEL_IODEV_PTR(id) &ACCEL_IODEV_SYM(id)
#define ACCEL_DEFINE_IODEV(id)                                          \
        SENSOR_DT_STREAM_IODEV(                                          \
                ACCEL_IODEV_SYM(id),                                 \
                ACCEL_ALIAS(id),                                 \
                ACCEL_TRIGGERS                                          \
                );

#define NUM_SENSORS 2

LISTIFY(NUM_SENSORS, ACCEL_DEFINE_IODEV, (;));

struct sensor_iodev *iodevs[NUM_SENSORS] = { LISTIFY(NUM_SENSORS, ACCEL_IODEV_PTR,␣
↪(,)) };

RTIO_DEFINE_WITH_MEMPOOL(accel_ctx, NUM_SENSORS, NUM_SENSORS, NUM_SENSORS, 16,␣
↪sizeof(void *));

int main(void)
{
        int rc;
        uint32_t accel_frame_iter = 0;
        struct sensor_three_axis_data accel_data[2] = {0};
        struct sensor_decoder_api *decoder;
        struct rtio_cqe *cqe;
        uint8_t *buf;
        uint32_t buf_len;
        struct rtio_sqe *handles[2];

        /* Start the streams */
        for (int i = 0; i < NUM_SENSORS; i++) {
                sensor_stream(iodevs[i], &accel_ctx, NULL, &handles[i]);
        }

        while (1) {
                cqe = rtio_cqe_consume_block(&accel_ctx);

                if (cqe->result != 0) {
                        printk("async read failed %d\n", cqe->result);
                        return;
                }

                rc = rtio_cqe_get_mempool_buffer(&accel_ctx, cqe, &buf, &buf_len);
```

(continues on next page)

(continued from previous page)

```
                if (rc != 0) {
                        printk("get mempool buffer failed %d\n", rc);
                        return;
                }

                struct device *sensor = ((struct sensor_read_config *)
                                         ((struct rtio_iodev *)cqe->userdata)->
↪data)->sensor;

                rtio_cqe_release(&accel_ctx, cqe);

                rc = sensor_get_decoder(sensor, &decoder);

                if (rc != 0) {
                        printk("sensor_get_decoder failed %d\n", rc);
                        return;
                }

                /* Frame iterator values when data comes from a FIFO */
                uint32_t accel_fit = 0;

                /* Number of accelerometer data frames */
                uint32_t frame_count;

                rc = decoder->get_frame_count(buf, {SENSOR_CHAN_ACCEL_XYZ, 0},
                                              &frame_count);
                if (rc != 0) {
                        printk("sensor_get_decoder failed %d\n", rc);
                        return;
                }

                /* If a tap has occurred lets print it out */
                if (decoder->has_trigger(buf, SENSOR_TRIG_TAP)) {
                        printk("Tap! Sensor %s\n", dev->name);
                }

                /* Decode all available accelerometer sample frames */
                for (int i = 0; i < frame_count; i++) {
                        decoder->decode(buf, {SENSOR_CHAN_ACCEL_XYZ, 0},
                                        accel_fit, 1, &accel_data);
                        printk("Accel data for %s " PRIsensor_three_axis_data "\n",
                               dev->name,
                               PRIsensor_three_axis_data_arg(accel_data, 0));
                }

                rtio_release_buffer(&accel_ctx, buf, buf_len);
        }
}
```

Note

Implementing the driver side of the sensor API requires an understanding how the sensor APIs are used. Please read through *Using Sensors* first!

Implementing Sensor Drivers

Implementing Attributes

- SHOULD implement attribute setting in a blocking manner.
- SHOULD provide the ability to get and set channel scale if supported by the device.
- SHOULD provide the ability to get and set channel sampling rate if supported by the device.

Implementing Fetch and Get

- SHOULD implement `sensor_sample_fetch_t` as a blocking call that stashes the specified channels (or all sensor channels) as driver instance data.
- SHOULD implement `sensor_channel_get_t` without side effects manipulating driver state returning stashed sensor readings.
- SHOULD implement `sensor_trigger_set_t` storing the address of the `sensor_trigger` rather than copying the contents. This is so `CONTAINER_OF` may be used for trigger callback context.

Implementing Read and Decode

- MUST implement `sensor_submit_t` as a non-blocking call.
- SHOULD implement `sensor_submit_t` using *Real Time I/O (RTIO)* to do non-blocking bus transfers if possible.
- MAY implement `sensor_submit_t` using a work queue if *Real Time I/O (RTIO)* is unsupported by the bus.
- SHOULD implement `sensor_submit_t` checking the `rtio_sqe` is of type `RTIO_SQE_RX` (read request).
- SHOULD implement `sensor_submit_t` checking all requested channels supported or respond with an error if not.
- SHOULD implement `sensor_submit_t` checking the provided buffer is large enough for the requested channels.
- SHOULD implement `sensor_submit_t` in a way that directly reads into the provided buffer avoiding copying of any kind, with few exceptions.
- MUST implement `sensor_decoder_api` with pure stateless functions. All state needed to convert the raw sensor readings into fixed point SI united values must be in the provided buffer.
- MUST implement `sensor_get_decoder_t` returning the `sensor_decoder_api` for that device type.

API Reference

Sensor

Related code samples

- 6dof device FIFO streamingGet accelerometer/gyroscope/temperature FIFO data frames from a sensor using SENSOR_TRIG_FIFO_WATERMARK as a trigger.
- ADT7420 high-accuracy digital I2C temperature sensorGet temperature data from an ADT7420 sensor using polling and window mode.
- AMG88XX infrared array sensorGet temperature data from an AMG88XX 8x8 thermal camera sensor.
- ams iAQcore indoor air quality sensorGet CO2 equivalent and VOC data from an ams iAQcore sensor.
- APDS9960 RGB, ambient light, and gesture sensorGet ambient light, RGB, and proximity/gesture data from an APDS9960 sensor.
- Barometric pressure and temperature sensor interrupt exampleManage interrupts from a barometric pressure and temperature sensor.
- Barometric pressure and temperature sensor polling exampleGet barometric pressure and temperature data from a sensor.
- BME280 humidity and pressure sensorGet temperature, pressure, and humidity data from a BME280 sensor.
- BMG160 3-axis gyroscopeGet temperature, and angular velocity data from a BMG160 sensor.
- BMI270 6-axis IMU sensorConfigure and read accelerometer and gyroscope data from a BMI270 sensor.
- BQ274XX fuel gauge sensorGet various fuel gauge parameters from a BQ274XX sensor.
- CCS811 indoor air quality sensorGet CO2 equivalent and VOC data from a CCS811 sensor.
- CPU die temperature pollingGet CPU die temperature data from a sensor using polling.
- DS18B20 1-Wire Temperature SensorGet ambient temperature data from a DS18B20 sensor (polling mode).
- ESP32 qdec per-unit threshold triggerInstall per-unit threshold callbacks on the ESP32 PCNT driver.
- FDC2X1X Capacitance-to-Digital ConverterGet capacitance and frequency data from a FDC2X1X sensor (polling & trigger).
- Flow MeterRead flow rate and volume from a Hall-effect pulse-output flow meter.
- FXAS21002 Gyroscope SensorGet gyroscope data synchronously from an FXAS21002 sensor.
- Generic 3-Axis accelerometer pollingGet 3-Axis accelerometer data from a sensor (polling mode).
- Generic 3-Axis accelerometer streamingGet 3-Axis accelerometer data from a sensor (stream mode).
- Generic 6DOF Motion DatareadyGet 6-Axis accelerometer and gyroscope data from a sensor (data ready interrupt mode).
- Generic CO2 polling sampleGet CO2 data from a sensor (polling mode).
- Generic device sample streaming using Data Ready triggerGet accelerometer data frames from a sensor using SENSOR_TRIG_DATA_READY.
- Generic digital humidity temperature sensor pollingGet temperature and humidity data from a DHT sensor (polling mode).
- Generic distance measurementMeasure distance to an object using a distance sensor

- Generic Light Sensor PollingGet illuminance data from a light sensor.
- Grove Temperature SensorGet temperature data from a Grove temperature sensor and display it on an LCD display.
- GROW R502-A Fingerprint SensorStore and match fingerprints using the GROW R502-A fingerprint sensor.
- Heart Rate SensorGet heart rate data from a sensor (polling/interrupt mode).
- HTS221 Temperature and Humidity MonitorGet temperature and humidity data from an HTS221 sensor (polling & trigger mode).
- I3G4250D 3-axis gyroscope sensorGet gyroscope data from an I3G4250D sensor.
- Iclegend S3KM1110 mmWave sensorRead standard and custom channels from an Iclegend S3KM1110 mmWave radar sensor.
- INA219 Bidirectional Power/Current MonitorGet shunt voltage, bus voltage, power and current from an INA219 sensor.
- ISL29035 Digital Light SensorGet light intensity data from an ISL29035 sensor (polling & trigger mode).
- JEDEC JC 42.4 compliant Temperature SensorGet ambient temperature from a JEDEC JC 42.4 compliant temperature sensor (polling & trigger mode).
- LIS2DH Motion SensorGet accelerometer data from an LIS2DH sensor (polling & trigger mode).
- LPS22HB Temperature and Pressure SensorGet pressure and temperature data from an LPS22HB sensor (polling mode).
- LPS22HH Temperature and Pressure SensorGet pressure and temperature data from an LPS22HH sensor (polling & trigger mode).
- LPS22HH Temperature and Pressure Sensor (I3C)Get pressure and temperature data from an LPS22HH sensor over I3C (polling & trigger mode).
- LSM303DLHC Magnetometer and Accelerometer sensorGet magnetometer and accelerometer data from an LSM303DLHC sensor (polling mode).
- LSM6DSL IMU sensorGet accelerometer and gyroscope data from an LSM6DSL sensor (polling & trigger mode).
- LSM6DSO IMU sensorGet accelerometer and gyroscope data from an LSM6DSO sensor (polling & trigger mode).
- LSM6DSO IMU sensor (I2C on I3C bus)Get accelerometer and gyroscope data from an LSM6DSO sensor using I2C on I3C bus (polling & trigger mode).
- LVGL line chart with accelerometer dataDisplay acceleration data on a real-time chart using LVGL.
- Magnetometer SensorGet magnetometer data from a magnetometer sensor (polling mode).
- MAX17262 Fuel Gauge SensorGet voltage, current and temperature data from a MAX17262 sensor (polling mode).
- MAX32664C + MAX86141 Sensor Hub
- MAX6675 K-thermocouple to digital converterGet temperature from a MAX6675 K-thermocouple to digital converter (polling mode).
- MPU6050 Invensense Motion Tracking DeviceGet temperature, acceleration, and angular velocity from an ICM42605 sensor (polling & trigger mode).

- MPU6050 motion tracking deviceGet temperature, acceleration, and angular velocity from an MPU6050 sensor (polling & trigger mode).
- MS5837 Digital Pressure SensorGet pressure and temperature data from an MS5837 sensor (polling mode).
- MTCH9010 Liquid Leak Detector TestbenchConfigure an MTCH9010 and receive data from it periodically.
- Multi-unit quadrature decoderRead several quadrature encoders from a single sensor device using the Read-and-Decode API.
- NPCX ADC ComparatorDetect upper/lower voltage limits using NPCX ADC Comparator driver.
- NXP MCUX Analog Comparator (ACMP)Get analog comparator data from an NXP MCUX Analog Comparator (ACMP).
- NXP MCUX Low-power Analog Comparator (LPCMP)Get analog comparator data from an NXP MCUX Low-power Analog Comparator (LPCMP).
- PAJ7620 Gesture SensorGet hand gesture data from PAJ7620 sensor.
- Proximity sensorGet proximity data from up to 10 proximity sensors (polling mode).
- Quadrature Decoder SensorGet rotation data from a quadrature decoder sensor.
- Secure MQTT Sensor/ActuatorImplement an MQTT-based IoT sensor/actuator device
- Sensor ClockTest and debug Sensor Clock functionality.
- Sensor shellInteract with sensors using the shell module.
- SGP40 and SHT4X digital humidity and multipixel gas sensorGet temperature, humidity and gas sensor data from SGP40 and SHT4X sensors (polling mode).
- SHT3XD humidity sensorGet temperature and humidity from a SHT3XD sensor (polling & trigger mode).
- SM351LT Magnetoresistive SensorDetect a magnet's presence using the SM351LT magnetoresistive sensor (polling & trigger mode).
- SoC Voltage SensorGet voltage data from an SoC's voltage sensor(s).
- SX9500 SAR proximity sensorGet proximity data from an SX9500 SAR proximity sensor.
- TDK Advanced Pedometer and Event Detection (APEX)Get TDK APEX event detection (trigger mode).
- TDK integrated angle sensor with TMR (Tunnel Magneto Resistance)Get angle and temperature data from a TDK TAD2144 sensor using SPI, I2C and Encoder(angle data only).
- TH02 Temperature and Humidity SensorGet temperature and humidity data from a TH02 sensor (polling mode).
- ThermometerGet ambient temperature data from a temperature sensor and get alerts when temperature drifts above a threshold. (polling & trigger mode).
- TMP108 Temperature SensorGet temperature data from a TMP108 sensor (polling & trigger mode).
- TMP112 Temperature SensorGet temperature data from a TMP112 sensor (polling & trigger mode).
- VCNL4040 Proximity and Ambient Light SensorGet proximity and ambient light data from a VCNL4040 sensor (polling & trigger mode).
- VEAA-X-3 proportional pressure control valveControl a VEAA-X-3 proportional pressure control valve.

- VEML6031 High Accuracy Ambient Light SensorGet ambient light data from a VEML4040 sensor (polling mode).
- VEML6046 RGBIR Color SensorGet red, green, blue and IR light data from a VEML6046 sensor (polling mode).
- VL53L0X Time Of Flight sensorGet distance data from a VL53L0X sensor (polling mode).
- X-NUCLEO-53L0A1 shieldInteract with the 7-segment display and VL53L0X ranging sensor of an X-NUCLEO-53L0A1 shield.
- X-NUCLEO-IKS01A1 shieldInteract with all the sensors of an X-NUCLEO-IKS01A1 shield.
- X-NUCLEO-IKS01A2 shield - SensorHub (Mode 2)Interact with all the sensors of an X-NUCLEO-IKS01A2 shield using Sensor Hub mode.
- X-NUCLEO-IKS01A2 shield - Standard (Mode 1)Interact with all the sensors of an X-NUCLEO-IKS01A2 shield using Standard Mode.
- X-NUCLEO-IKS01A3 shield - SensorHub (Mode 2)Interact with all the sensors of an X-NUCLEO-IKS01A3 shield using Sensor Hub mode.
- X-NUCLEO-IKS01A3 shield - Standard (Mode 1)Interact with all the sensors of an X-NUCLEO-IKS01A3 shield using Standard mode.
- X-NUCLEO-IKS02A1 shield - SensorHub (Mode 2)Interact with all the sensors of an X-NUCLEO-IKS02A1 shield using Sensor Hub mode.
- X-NUCLEO-IKS02A1 shield - Standard (Mode 1)Interact with all the sensors of an X-NUCLEO-IKS02A1 shield using Standard mode.
- X-NUCLEO-IKS4A1 shield - SensorHub (Mode 2)Interact with all the sensors of an X-NUCLEO-IKS4A1 shield using SHUB2 (mode 2).
- X-NUCLEO-IKS4A1 shield - SensorHub (Mode 3)Interact with all the sensors of an X-NUCLEO-IKS4A1 shield using SHUB1 (mode 3).
- X-NUCLEO-IKS4A1 shield - Standard (Mode 1 + I3C)Interact with all the sensors of an X-NUCLEO-IKS4A1 shield using Standard mode over I3C.
- X-NUCLEO-IKS4A1 shield - Standard (Mode 1)Interact with all the sensors of an X-NUCLEO-IKS4A1 shield using Standard mode.

Sensor emulator backend API

GNSS (Global Navigation Satellite System)

Overview GNSS is a general term which covers satellite systems used for navigation, like GPS (Global Positioning System). GNSS services are usually accessed through GNSS modems which receive and process GNSS signals to determine their position, or more specifically, their antennas position. They usually additionally provide a precise time synchronization mechanism, commonly named PPS (Pulse-Per-Second).

Subsystem support The GNSS subsystem is based on the *Modem modules*. The GNSS subsystem covers everything from sending and receiving commands to and from the modem, to parsing, creating and processing NMEA0183 messages.

Adding support for additional NMEA0183 based GNSS modems requires little more than implementing power management and configuration for the specific GNSS modem.

Adding support for GNSS modems which use other protocols and/or buses than the usual NMEA0183 over UART is possible, but will require a bit more work from the driver developer.

Power management GNSS receivers typically begin acquiring and tracking GNSS signals as soon as they are powered on, unless configured otherwise. In order to preserve power, an application can either remove power from the GNSS receiver or stop the internal GNSS engine from actively acquiring and tracking signals. The latter option halts position computation while keeping the receiver powered with reduced power consumption and the communication link to the host application available for other operations or for resuming the GNSS engine.

In the GNSS subsystem, this can be done using the `gnss_stop()` API call. This is distinct from Zephyr's device power management (suspend/resume): the receiver remains powered and able to communicate with the host application, but GNSS tracking is stopped.

GNSS tracking can be resumed using `gnss_start()`. The application can specify a `gnss_start_mode`, which affects time-to-first-fix. A hot start preserves navigation data and allows the receiver to reacquire signals quickly, while warm and cold starts discard some or all navigation data, respectively, resulting in longer acquisition times.

GNSS device drivers must ensure the GNSS modem starts tracking when resumed from a suspended or powered-off state, without requiring the application to call `gnss_start()` explicitly. Most receivers will end up performing a cold start in this situation, simply because they lost their tracking data while unpowered, but a driver may instead attempt a warm or hot start if the modem retained enough state to do so faster. The requirement is that tracking resumes automatically on power up; clearing navigation data is only a side effect of the modem having been unpowered, not an intended part of resuming from suspend.

Configuration Options Related configuration options:

- `CONFIG_GNSS`
- `CONFIG_GNSS_SATELLITES`
- `CONFIG_GNSS_DUMP_TO_LOG`

Navigation Reference

Navigation

Related code samples

- GNSSConnect to a GNSS device to obtain time, navigation data, and satellite information.

GNSS API Reference

GNSS

Related code samples

- GNSSConnect to a GNSS device to obtain time, navigation data, and satellite information.

Peripheral Sensor Interface (PSI5)

Overview The PSI5 API provides functionality to communicate with Peripheral Sensor Interface (PSI5) devices.

Configuration Options Related configuration options:

- `CONFIG_PSI5`

API Reference
PSI5

Related code samples

- PSI5 interfaceUse the PSI5 (Peripheral Sensor Interface) driver.

Single Edge Nibble Transmission (SENT)

Overview The SENT API provides functionality to communicate with Single Edge Nibble Transmission (SENT) devices.

Configuration Options Related configuration options:

- `CONFIG_SENT`

API Reference
SENT

Related code samples

- SENT interfaceUse the SENT (Single Edge Nibble Transmission) driver.

Biometrics

Overview The biometrics API provides a unified interface for biometric sensors such as fingerprint scanners, iris scanners, and face recognition modules. These sensors are commonly used for secure authentication in embedded systems, access control devices, and IoT applications.

The API supports the full lifecycle of biometric operations including enrollment, template management, and matching. Sensors can store templates on-device or on the host system depending on hardware capabilities.

A typical fingerprint enrollment process requires capturing multiple samples of the same finger to create a reliable template. The matching process compares a captured sample against stored templates to verify identity.

Configuration Options Related configuration options:

- `CONFIG_BIOMETRICS`
- `CONFIG_BIOMETRICS_INIT_PRIORITY`

API Reference
Biometrics

Related code samples

- Fingerprint sensorDemonstrate fingerprint sensor enrollment, verification, identification, and template management using the biometric driver API.

Motion & Actuation

6.7.9 Motion & Actuation

PWM, stepper motors, haptics, buzzers, and other actuators.

Haptics

Overview The haptics API allows for the control of haptic driver devices for the purposes of performing haptic feedback events.

During a haptic feedback event the haptic device drives a signal to an actuator. The source of the haptic event signal varies depending on the capabilities of the haptic device.

Some examples of haptic signal sources are analog signals, preprogrammed (ROM) wavetables, synthesized (RAM) wavetables, and digital audio streams.

Additionally, haptic driver devices often offer controls for adjusting and tuning the drive signal to meet the electrical requirements of their respective actuators.

API Reference

Haptics

Related code samples

- CS40L26 Haptic DriverDrive an LRA using the CS40L26/27 haptic driver chip.
- CS40L5x Haptic DriverDrive an LRA using the CS40L5x haptic driver chip.
- DRV2605 Haptic DriverDrive an LRA using the DRV2605 haptic driver chip.
- TM6605 Haptic DriverDrive an LRA using the Titan Micro Electronics TM6605 haptic driver chip.

Pulse Width Modulation (PWM)

Overview Pulse Width Modulation (PWM) is a technique for encoding information or controlling power delivery by varying the duty cycle of a digital signal. A PWM signal switches between high and low states at a fixed frequency; the fraction of each period spent in the high state is the *duty cycle*.

Common uses of PWM in embedded systems:

- **LED brightness control** — duty cycle maps directly to perceived brightness
- **DC motor speed control** — average voltage is proportional to duty cycle
- **Servo motor positioning** — pulse width encodes the desired angle
- **Audio tone generation** — varying frequency produces audible tones
- **Power conversion** — buck/boost converters and class-D amplifiers

PWM Signal Parameters A PWM signal is described by three values:

- **Period** — the total duration of one on/off cycle, in nanoseconds
- **Pulse width** — the duration of the high (active) portion, in nanoseconds
- **Duty cycle** — the ratio of pulse width to period (expressed as a percentage); a 50% duty cycle means the signal is high for half the period

```
|<-- pulse -->|
+--------------+            +-
|              |            |
+              +------------+
|<----------- period -------->|
```

Zephyr PWM Driver Model Each PWM controller exposes one or more independent *channels*. A channel is identified by a zero-based index and drives a single output pin.

All PWM drivers implement the same API defined in `include/zephyr/drivers/pwm.h`. The central function is `pwm_set()`, which programs the period and pulse width for a channel. Helper macros convert common time units to nanoseconds:

- `PWM_HZ` — frequency in Hz (e.g. `PWM_HZ(1000)` 1 ms period)
- `PWM_KHZ` — frequency in kHz
- `PWM_USEC` — period or pulse width in microseconds
- `PWM_MSEC` — period or pulse width in milliseconds

Device Tree Configuration PWM pins are described in devicetree under the `pwms` property. Each entry specifies the controller phandle, channel number, period in nanoseconds, and optional polarity flags:

```
/ {
    my_node {
        pwms = <&pwm0 0 PWM_MSEC(20) PWM_POLARITY_NORMAL>;
        pwm-names = "servo";
    };
};
```

In C, retrieve the spec with the `PWM_DT_SPEC_GET` family of macros:

```
static const struct pwm_dt_spec servo =
    PWM_DT_SPEC_GET(DT_NODELABEL(my_node));
```

Usage Examples Setting duty cycle using a `pwm_dt_spec`:

```
#include <zephyr/drivers/pwm.h>

static const struct pwm_dt_spec led_pwm =
    PWM_DT_SPEC_GET(DT_ALIAS(pwm_led0));

int set_led_brightness(uint8_t percent)
{
    if (!device_is_ready(led_pwm.dev)) {
        return -ENODEV;
    }

    /* pulse = period * duty_cycle / 100 */
    uint32_t pulse = led_pwm.period / 100 * percent;

    return pwm_set_dt(&led_pwm, led_pwm.period, pulse);
}
```

Setting a servo position (1–2 ms pulse within a 20 ms period):

```
#include <zephyr/drivers/pwm.h>

#define SERVO_PERIOD_NS   PWM_MSEC(20)
#define SERVO_MIN_PULSE   PWM_USEC(1000)
#define SERVO_MAX_PULSE   PWM_USEC(2000)

int set_servo_angle(const struct device *pwm_dev, uint32_t channel,
```

(continues on next page)

(continued from previous page)

```
                    uint8_t angle_deg)
{
    uint32_t pulse = SERVO_MIN_PULSE +
        (SERVO_MAX_PULSE - SERVO_MIN_PULSE) * angle_deg / 180;

    return pwm_set(pwm_dev, channel, SERVO_PERIOD_NS, pulse,
                   PWM_POLARITY_NORMAL);
}
```

PWM Capture Some PWM controllers support *input capture*, which measures the period and/or pulse width of an incoming signal. This is useful for decoding PWM signals from external sources such as RC receivers or sensor outputs.

Enable capture with `pwm_configure_capture()` and `pwm_enable_capture()`:

```
void pwm_capture_cb(const struct device *dev, uint32_t channel,
                    uint32_t period_cycles, uint32_t pulse_cycles,
                    int status, void *user_data)
{
    if (status != 0) {
        return;
    }
    /* convert cycles to nanoseconds using pwm_get_cycles_per_sec() */
}

/* configure for single-shot capture of both period and pulse */
pwm_configure_capture(dev, channel,
                      PWM_CAPTURE_TYPE_BOTH | PWM_CAPTURE_MODE_SINGLE,
                      pwm_capture_cb, NULL);
pwm_enable_capture(dev, channel);
```

Configuration Options Related configuration options:

- `CONFIG_PWM`
- `CONFIG_PWM_SHELL`
- `CONFIG_PWM_CAPTURE`
- `CONFIG_PWM_EVENT`

API Reference

PWM

Related code samples

- Fade LEDFade an LED using the PWM API.
- PWM BlinkyBlink an LED using the PWM API.
- PWM CaptureCapture a PWM signal.
- PWM EventEvents on a PWM signal.
- PWM RGB LEDDrive an RGB LED using the PWM API.
- ServomotorDrive a servomotor using the PWM API.
- SoundUse the piezo buzzer on the BBC micro:bit board.

- STM32 pwm mastermodeDevicetree configuration for utilizing master/slave capabilities to synchronize timer instances.

Fake PWM

Steppers

The stepper driver subsystem consists of two device driver APIs:

Stepper API The stepper driver API provides a common interface for stepper drivers.

- Configure **micro-stepping resolution** using `stepper_set_micro_step_res()` and `stepper_get_micro_step_res()`.
- **Enable** the stepper driver using `stepper_enable()`.
- **Disable** the stepper driver using `stepper_disable()`.
- Register an **event callback** using `stepper_set_event_cb()`.

Stepper Motion Controller API The stepper motion controller API provides a common interface for stepper motion controllers.

- Configure **reference position** in microsteps using `stepper_ctrl_set_reference_position()` and `stepper_ctrl_get_actual_position()`.
- Set **step interval** in nanoseconds between steps using `stepper_ctrl_set_microstep_interval()`
- **Move by** +/- micro-steps also known as **relative movement** using `stepper_ctrl_move_by()`.
- **Move to** a specific position also known as **absolute movement** using `stepper_ctrl_move_to()`.
- Run continuously with a **constant step interval** in a specific direction until a stop is detected using `stepper_ctrl_run()`.
- **Stop** the stepper using `stepper_ctrl_stop()`.
- Check if the stepper is **moving** using `stepper_ctrl_is_moving()`.
- Register an **event callback** using `stepper_ctrl_set_event_cb()`.

Device Tree In the context of stepper motion controllers, devicetree provides the initial hardware configuration for stepper drivers on a per device level. Each device must specify a device tree binding in Zephyr, and ideally, a set of hardware configuration options for things such as current settings, ramp parameters and furthermore. These can then be used in a boards devicetree to configure a stepper driver to its initial state.

Driver Composition Scenarios Below are two typical scenarios:

Integrated Stepper Motion Control and Driver A device composed of both motion controller and stepper driver blocks in a single IC is modeled as a multi-functional device in devicetree, and has two software drivers implementing Stepper Motion Controller and Stepper Hardware Driver APIs respectively. An example of such device is `adi,tmc50xx`.

```
/ {
    aliases {
        x_axis_stepper_ctrl = &tmc50xx_0_motion_controller;
        y_axis_stepper_ctrl =  &tmc50xx_1_motion_controller;
```

(continues on next page)

(continued from previous page)

```
        x_axis_stepper_driver = &tmc50xx_0_stepper_driver;
        y_axis_stepper_driver = &tmc50xx_1_stepper_driver;
    };
};

&spi0 {
    /* SPI bus options here, not shown */

    /* Dual controller/driver for up to two 2-phase bipolar stepper motors */
    tmc50xx: tmc50xx@0 {
        compatible = "adi,tmc50xx";
        reg = <0>;
        spi-max-frequency = <DT_FREQ_M(8)>; /* Maximum SPI bus frequency */

        poscmp-enable; test-mode; lock-gconf; /* ADI TMC Global configuration␣
↪flags */
        clock-frequency = <DT_FREQ_M(16)>; /* Internal/External Clock frequency */

        tmc50xx_0_stepper_driver: stepper-driver-0 {
            idx = <0>;
             compatible = "adi,tmc50xx-stepper-driver";
             micro-step-res = <256>;
             /* ADI TMC stallguard settings specific to TMC50XX */
             stallguard2-threshold=<30>;
         };

        tmc50xx_0_motion_controller: motion-controller-0 {
            idx = <0>;
            compatible = "adi,tmc50xx-stepper-ctrl";
            ...
            vmax = <900000>;
            amax = <50000>;
            ...
            activate-stallguard2;
            ...
         };

        tmc50xx_1_stepper_driver: stepper-driver-1 {
            idx = <1>;
             compatible = "adi,tmc50xx-stepper-driver";
             micro-step-res = <256>;
             /* ADI TMC stallguard settings specific to TMC50XX */
             stallguard2-threshold=<30>;
         };

        tmc50xx_1_motion_controller: motion-controller-1 {
            idx = <1>;
            compatible = "adi,tmc50xx-stepper-ctrl";
            ...
            vstart = <1000>;
            ...
            stallguard-threshold-velocity=<200000>;
         };
     };
};
```

Following the aforementioned configurations, the stepper driver subsystem can be used in the

application code as follows:

```
static const struct device *x_stepper_ctrl =␣
↪DEVICE_DT_GET(DT_ALIAS(x_axis_stepper_ctrl));
static const struct device *x_stepper_driver =␣
↪DEVICE_DT_GET(DT_ALIAS(x_axis_stepper_driver));
static const struct device *y_stepper_ctrl =␣
↪DEVICE_DT_GET(DT_ALIAS(y_axis_stepper_ctrl));
static const struct device *y_stepper_driver =␣
↪DEVICE_DT_GET(DT_ALIAS(y_axis_stepper_driver));
...
stepper_ctrl_move_to(x_stepper_ctrl, 200);
stepper_ctrl_stop(x_stepper_ctrl);
stepper_disable(x_stepper_driver);
stepper_disable(y_stepper_driver);
```

Individual Stepper Motion Controller and Driver Following is an example of a device tree configuration for a stepper driver with a dedicated stepper motion controller:

```
/ {
    aliases {
        stepper_driver = &tmc2209
        stepper_ctrl = &step_dir_motion_control;
    };

    tmc2209: tmc2209 {
        compatible = "adi,tmc2209";
        enable-gpios = <&gpioa 6 GPIO_ACTIVE_HIGH>;
        m0-gpios = <&gpiob 0 GPIO_ACTIVE_HIGH>;
        m1-gpios = <&gpioa 7 GPIO_ACTIVE_HIGH>;
    };

    step_dir_motion_control: step-dir-motion-control {
        compatible = "zephyr,gpio-step-dir-stepper-ctrl";
        step-gpios = <&gpioa 9 GPIO_ACTIVE_HIGH>;
        dir-gpios = <&gpioc 7 GPIO_ACTIVE_HIGH>;
        invert-direction;
        stepper-driver = <&tmc2209>;
    };
};
```

Following the aforementioned configurations, the stepper driver subsystem can be used in the application code as follows:

```
static const struct device *stepper_driver =␣
↪DEVICE_DT_GET(DT_ALIAS(stepper_driver));
static const struct device *stepper_ctrl = DEVICE_DT_GET(DT_ALIAS(stepper_ctrl));
...
stepper_ctrl_move_to(stepper_ctrl, 200);
stepper_ctrl_stop(stepper_ctrl);
stepper_disable(stepper_driver);
```

Stepper Motion Controller API Test Suite The stepper motion controller API test suite provides a set of tests that can be used to verify the functionality of stepper motion controllers.

```
# From the root of the zephyr repository
west build -b <board> --extra-dtc-overlay <path/to/board.overlay>␣
```

(continues on next page)

(continued from previous page)

```
↪tests/drivers/stepper/stepper_ctrl
west flash
```

Sample Output Below is a snippet of the test output for the h-bridge-stepper-ctrl.

```
===================================================================
TESTSUITE stepper succeeded

------ TESTSUITE SUMMARY START ------

SUITE PASS - 100.00% [stepper_ctrl]: pass = 10, fail = 0, skip = 0, total = 10␣
↪duration = 6.869 seconds
 - PASS - [stepper_ctrl.test_actual_position] duration = 0.001 seconds
 - PASS - [stepper_ctrl.test_move_by_negative_step_count] duration = 2.207 seconds
 - PASS - [stepper_ctrl.test_move_by_positive_step_count] duration = 2.202 seconds
 - PASS - [stepper_ctrl.test_move_to_negative_step_count] duration = 1.106 seconds
 - PASS - [stepper_ctrl.test_move_to_positive_step_count] duration = 1.102 seconds
 - PASS - [stepper_ctrl.test_move_zero_steps] duration = 0.006 seconds
 - PASS - [stepper_ctrl.test_run_negative_direction] duration = 0.115 seconds
 - PASS - [stepper_ctrl.test_run_positive_direction] duration = 0.124 seconds
 - PASS - [stepper_ctrl.test_set_micro_step_interval_invalid_zero] duration =␣
↪0.002 seconds
 - PASS - [stepper_ctrl.test_stop] duration = 0.004 seconds

------ TESTSUITE SUMMARY END ------

===================================================================
PROJECT EXECUTION SUCCESSFUL
```

API Reference A common set of functions which should be implemented by all stepper drivers.

Stepper Hardware Driver

A common set of functions which should be implemented by all stepper motion controllers.

Stepper Motion Controller

Stepper motion controller specific APIs

Trinamic

Trinamic Stepper Controller

Buzzer

The buzzer subsystem exposes an API to drive buzzer hardware uniformly, regardless of whether the underlying part is a passive piezo driven by a PWM channel or an active buzzer controlled by a single GPIO line.

Basic Operation Applications obtain a buzzer device through devicetree and drive it through the functions in include/zephyr/drivers/buzzer.h:

Buzzer API calls do not wait for the requested tone duration. Durations describe how long the hardware should keep emitting a tone, not how long the calling thread should sleep; use `BUZZER_DURATION_FOREVER` to play until an explicit stop.

- `buzzer_tone()` plays a tone at a specific frequency for a specific duration. The hardware keeps emitting the tone for the requested duration and the driver silences it automatically.
- `buzzer_beep()` plays the buzzer's natural operating frequency, encoded by the board file in the period cell of the `pwms` DT property (typically the piezo's mechanical resonance, which is the loudest tone the part can produce). On active buzzers the call is equivalent to driving the GPIO line on, since the hardware oscillator determines the actual pitch.
- `buzzer_set_volume()` adjusts the perceived loudness. Zero silences immediately; non-zero values are stored and applied on the next tone. Active buzzers, lacking analog volume control, map zero to silent and any non-zero value to their single audible level.
- `buzzer_stop()` cancels any in-progress tone immediately.

Backends Two devicetree-discoverable backends are provided:

- `pwm-buzzer` for passive piezo buzzers driven from a PWM channel. The PWM channel period sets the audio frequency and the duty cycle sets the perceived volume; the driver derives both from the application's tone and volume requests.
- `gpio-buzzer` for active buzzers driven through a single GPIO. The driver toggles the line on for any non-zero frequency and off for `BUZZER_FREQ_REST`.

API Reference
Buzzer

Related code samples

- Buzzer TonePlay a short beep and a few tones through the buzzer device aliased as buzzer0.

System & Diagnostics

6.7.10 System & Diagnostics

System information, synchronization, DMA, and diagnostics.

Hardware Information

Overview The HW Info API provides access to hardware information such as device identifiers and reset cause flags.

Reset cause flags can be used to determine why the device was reset; for example due to a watchdog timeout or due to power cycling. Different devices support different subset of flags. Use `hwinfo_get_supported_reset_cause()` to retrieve the flags that are supported by that device.

Most implementations are SoC specific, reading the identifiers from vendor registers or memory. The generic `zephyr,hwinfo-nvmem` backend sources the device ID, and optionally the EUI-64, from NVMEM cells (see *Non-Volatile Memory (NVMEM)*).

Configuration Options Related configuration options:

- `CONFIG_HWINFO`
- `CONFIG_HWINFO_NVMEM`

API Reference
Hardware Info

Hardware Spinlocks (HWSPINLOCK)

Overview An HWSPINLOCK device is a peripheral used to protect shared resources across clusters in the system. Each HWSPINLOCK instance is providing one or more spinlocks. The api is similar to regular zephyr spinlocks.

Spinlock APIs

Because we also want to protect the spinlock resource to be used by multiple cores in the same cluster, each HWSPINLOCK device include a regular zephyr spinlock, and use it to lock the access to HWSPINLOCK hardware.

API Reference
Hardware Spinlock

Related code samples

- HWSPINLOCKDemonstrate cross-core synchronization using the HWSPINLOCK API together with the MBOX API.

Entropy

Overview The entropy API provides functions to retrieve entropy values from entropy hardware present on the platform. The entropy APIs are provided for use by the random subsystem and cryptographic services. They are not suitable to be used as random number generation functions.

API Reference
Entropy

Error Detection And Correction (EDAC)

Error Detection And Correction is a mechanism used to detect and correct errors while storing or reading data.

In Band Error Correction Code (IBECC)

Overview The mechanism initially found in Intel Elkhart Lake SOCs and later boards is an integrated memory controller with IBECC.

The In-Band Error Correction Code (IBECC) improves reliability by providing error detection and correction. IBECC can work for all or for specific regions of physical memory space. The IBECC is useful for memory technologies that do not support the out-of-band ECC.

IBECC adds memory overhead of 1/32 of the memory. This memory is not accessible and used to store ECC syndrome data. IBECC converts read / write transactions to two separate transactions: one for actual data and another for cache line containing ECC value.

There is a debug feature IBECC Error Injection which helps to debug and verify IBECC functionality. ECC errors are injected on the write path and cause ECC errors on the read path.

IBECC Configuration There are three IBECC operation modes which can be selected by Bootloader. They are listed below:

- OPERATION_MODE = 0x0 sets functional mode to protect requests based on address range
- OPERATION_MODE = 0x1 sets functional mode to all requests not be protected and to ignore range checks

- OPERATION_MODE = 0x2 sets functional mode to protect all requests and ignore range checks

IBECC operational mode is configured through BIOS or Bootloader. For operation mode 0 there are more BIOS configuration options such as memory regions.

Due to high security risk Error Injection capability should not be enabled for production. Error Injection is only enabled for tests.

IBECC Logging IBECC logs the following fields:

- Error Address
- Error Syndrome
- Error Type
  - Correctable Error (CE) - error is detected and corrected by IBECC module.
  - Uncorrectable Error (UE) - error is detected by IBECC module and not automatically corrected.

The IBECC driver provides error type for the higher-level application to implement desired policy with respect for handling those memory errors. Error syndrome is not used in the IBECC driver but provided to higher-level application.

Usage notes Exceptional care needs to be taken with Non Maskable Interrupt (NMI). NMI will arrive at any time, even if the local CPU has disabled interrupts. That means that no locking mechanism can protect code against an NMI happening. Zephyr's IPC mechanisms universally use local IRQ locking as the base layer for all higher-level synchronization primitives. So, you cannot share anything that is "protected" by a lock with an NMI, because the protection does not work. The only tool you have available for synchronization in the Zephyr API that works against an NMI is the atomic layer. This also applies to callback function which is called by NMI handler.

Configuration option Related configuration option:

- `CONFIG_EDAC_IBECC`

Configuration option Related configuration option:

- `CONFIG_EDAC`

API Reference
EDAC

Related code samples

- EDAC shellTest error detection and correction (EDAC) using shell commands.

Coredump Device

Overview The coredump device is a pseudo-device driver with two types.A COREDUMP_TYPE_MEMCPY type exposes device tree bindings for memory address/size values to be included in any dump. And the driver exposes an API to add/remove dump memory regions at runtime. A COREDUMP_TYPE_CALLBACK device requires exactly one entry in the memory-regions array with a size of 0 and a desired size. The driver will statically allocate memory of the desired size and provide an API to register a callback function to fill that memory when a dump occurs.

Configuration Options Related configuration options:

- `CONFIG_COREDUMP_DEVICE`

API Reference
Coredump pseudo-device

Cyclic Redundancy Check (CRC)

Overview The Cyclic Redundancy Check (CRC) API provides functions for configuring and computing CRC values on hardware.

Configuration Options Related configuration options:

- `CONFIG_CRC_DRIVER`
- `CONFIG_CRC_DRIVER_INIT_PRIORITY`

API Reference
CRC driver APIs

Related code samples

- Cyclic Redundancy Check Drivers (CRC)Compute and verify a CRC checksum using the CRC driver API.

Platform Environment Control Interface (PECI)

Overview The Platform Environment Control Interface, abbreviated as PECI, is a thermal management standard introduced in 2006 with the Intel Core 2 Duo Microprocessors. The PECI interface allows external devices to read processor temperature, perform processor manageability functions, and manage processor interface tuning and diagnostics. The PECI bus driver APIs enable the interaction between Embedded Microcontrollers and CPUs.

Configuration Options Related configuration options:

- `CONFIG_PECI`

API Reference
PECI

Related code samples

- PECI interfaceMonitor CPU temperature using PECI.

Direct Memory Access (DMA)

Overview Direct Memory Access (Controller) is a commonly provided type of co-processor that can typically offload transferring data to and from peripherals and memory.

The DMA API is not a portable API and really cannot be as each DMA has unique memory requirements, peripheral interactions, and features. The API in effect provides a union of all useful DMA functionality drivers have needed in the tree. It can still be a good abstraction, with care, for peripheral devices for vendors where the DMA IP might be very similar but have slight variances.

The DMA drivers in general do not handle cache coherency; this is left up to the developer as requirements vary dramatically depending on the application. See *Caching Basics* for an overview of cache management in Zephyr.

Driver Implementation Expectations

Synchronization and Ownership From an API point of view, a DMA channel is a single-owner object, meaning the drivers should not attempt to wrap a channel with kernel synchronization primitives such as mutexes or semaphores. If DMA channels require mutating shared registers, those register updates should be wrapped in a spin lock.

This enables the entire API to be low-cost and callable from any call context, including ISRs where it may be very useful to start/stop/suspend/resume/reload a channel transfer.

Transfer Descriptor Memory Management Drivers should not attempt to use heap allocations of any kind. If object pools are needed for transfer descriptors then those should be setup in a way that does not break the promise of ISR-allowable calls. Many drivers choose to create a simple static descriptor array per channel with the size of the descriptor array adjustable using Kconfig.

Channel State Machine Expectations DMA channels should be viewed as state machines that the DMA API provides transition events for in the form of API calls. Every driver is expected to maintain its own channel state tracking. The busy state of the channel should be inspectable at any time with `dma_get_status()`.

A diagram, showing those expected possible state transitions and their API calls is provided here for reference.

API Reference

DMA

Related code samples

- NXP SmartDMA memory-to-memoryUse the NXP SmartDMA coprocessor to perform a memory-to-memory transfer through the Zephyr DMA API.

Multiplexer (MUX)

Overview The MUX subsystem provides a unified API for hardware signal multiplexers, it lets consumer drivers route an input signal to an output by referencing a MUX controller through standard devicetree `mux-controls` or `mux-states` phandle-array properties, without depending on any vendor-specific HAL.

The subsystem distinguishes two consumer-side patterns:

- `mux-controls` — the consumer references a controller and supplies the *state* (which input to route, or which output pattern to write) at runtime via `mux_control_set()`. Use this when the routing changes during normal operation.
- `mux-states` — the consumer references a controller *and* a fixed state in devicetree (the trailing cell of the specifier is the state value). The runtime call `mux_state_apply()` programs that fixed state in one line. Use this when the routing is decided at integration time and never changes after init.

Devicetree bindings

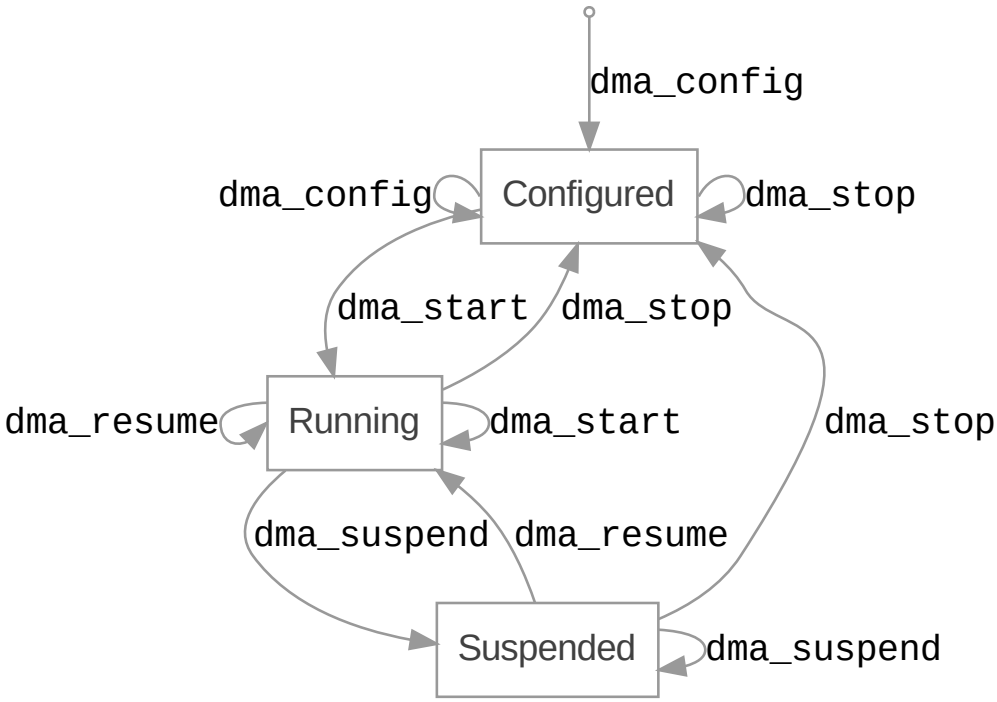


Fig. 2: DMA state finite state machine

Controller side Every MUX controller binding includes `mux-controller.yaml` (in `dts/bindings/mux/`) and declares two cell-count properties:

- `#mux-control-cells` — number of cells in a `mux-controls` specifier, used purely to address the desired control line within the controller.
- `#mux-state-cells` — number of cells in a `mux-states` specifier; equals `#mux-control-cells + 1`. The trailing cell carries the state value and is extracted by the framework into `mux_state::state` at compile time.

Backends are free to name the addressing cells after their hardware (`channel`, `output`, `index`, `device/input`, ...) and to name the trailing state cell after its hardware semantics (`state`, `input`, `connection`, `source`, ...).

Consumer side Consumers include `mux-consumer.yaml` and may set:

- `mux-controls` — list of phandle + addressing-cells tuples.
- `mux-control-names` — optional names matching `mux-controls` entries.
- `mux-states` — list of phandle + addressing-cells + trailing state cell tuples.
- `mux-state-names` — optional names matching `mux-states` entries.

Configuration Options Related configuration options:

- `CONFIG_MUX`

API Reference
MUX
Timing

6.7.11 Timing

Timers, clocks, watchdogs, and timekeeping primitives.

Clock Control

Overview The clock control API provides access to clocks in the system, including the ability to turn them on and off.

Configuration Options Related configuration options:

- `CONFIG_CLOCK_CONTROL`

API Reference
Clock Control

Related code samples

- AXI Clocking WizardEnable, reconfigure and read back an AXI Clocking Wizard output clock.
- LiteX clock control driverUse LiteX clock control driver to generate multiple clock signals.

Clock Monitor

Overview The clock monitor API provides access to hardware peripherals that observe a clock signal at runtime and report when its frequency drifts outside expected bounds or stops entirely. It is intended for functional-safety and diagnostic use cases — detecting failed oscillators, lost reference clocks, or out-of-spec frequency drift on critical clock trees.

Operating Modes Both modes share one lifecycle: configure with `clock_monitor_configure()`, begin operation with `clock_monitor_start()`, end it with `clock_monitor_stop()`.

The API exposes two operating modes:

`CLOCK_MONITOR_MODE_WINDOW`
: Continuous threshold check. The hardware compares the monitored clock's frequency against programmable high and low bounds derived from `clock_monitor_window_cfg.expected_hz` and `clock_monitor_window_cfg.tolerance_ppm`. Threshold crossings are delivered asynchronously through the user callback installed at configure time.

`CLOCK_MONITOR_MODE_MEASURE`
: One frequency measurement per `clock_monitor_start()`, delivered through the configure-time callback (`CLOCK_MONITOR_EVT_MEASURE_DONE` plus the measured value in Hz). The device automatically returns to the configured (stopped) state before the callback runs, so no `clock_monitor_stop()` is needed on the happy path. To repeat the measurement, call `clock_monitor_start()` again from the callback — both `clock_monitor_start()` and `clock_monitor_stop()` are ISR-safe (the same idiom as re-arming a counter alarm from its callback).

For MEASURE mode, the API provides no blocking wait: the application owns the timeout. The usual pattern is to wait on a semaphore given from the callback with an application-chosen timeout and call `clock_monitor_stop()` on the timeout path to abort the in-flight measurement. Alternatively, the most recent result can be polled with `clock_monitor_get_rate()` — also the retrieval path for user-mode threads, where callbacks may not be installed.

Events Events are delivered as a bitmask in `clock_monitor_event_data.events`:

- `CLOCK_MONITOR_EVT_FREQ_HIGH` — monitored frequency exceeded the upper threshold (WINDOW mode).
- `CLOCK_MONITOR_EVT_FREQ_LOW` — monitored frequency fell below the lower threshold (WINDOW mode).
- `CLOCK_MONITOR_EVT_CLOCK_LOST` — monitored clock stopped producing edges within the measurement window (MEASURE mode hardware failure).
- `CLOCK_MONITOR_EVT_MEASURE_DONE` — measurement completed successfully (MEASURE mode); `clock_monitor_event_data.measured_hz` holds the result.

The configure-time callback is the only event delivery path. For MEASURE mode the most recent completed result additionally remains retrievable by polling `clock_monitor_get_rate()`. WINDOW-mode user-mode observers, where callbacks may not be installed, receive events via a supervisor-side relay (e.g. `k_msgq` fed from the callback).

Configuration A clock monitor must be configured via `clock_monitor_configure()` before it is started. The configuration carries the operating mode, the mode-specific parameters (expected frequency, tolerance, measurement window) and an optional asynchronous callback. Configuring is only possible while the monitor is stopped; reconfiguration is done by calling `clock_monitor_configure()` again with a new mode/parameter set — there is no separate teardown operation. See `clock_monitor_configure()` in the *API Reference* for the full list of return codes.

Related configuration options:

- `CONFIG_CLOCK_MONITOR`

API Reference

Clock Monitor

Related code samples

- Clock Monitor Frequency CheckContinuously check a clock against programmable frequency thresholds.
- Clock Monitor Frequency MeterMeasure the frequency of a clock once and print the result.

Counter

Overview

API Reference

Counter

Related code samples

- Counter AlarmImplement an alarm application using the counter API.
- System Off (MCXE31B standby)Use standby (the only low-power mode) on FRDM-MCXE31B and wake via an RTC alarm.

Real-Time Clock (RTC)

Overview

Table 2: **Glossary**

Word	Definition
Real-time clock	Low power device tracking time using broken-down time
Real-time counter	Low power counter which can be used to track time
RTC	Acronym for real-time clock

An RTC is a low power device which tracks time using broken-down time. It should not be confused with low-power counters which sometimes share the same name, acronym, or both.

RTCs are usually optimized for low energy consumption and are usually kept running even when the system is in a low power state.

RTCs usually contain one or more alarms which can be configured to trigger at a given time. These alarms are commonly used to wake up the system from a low power state.

Devicetree bindings RTC bindings must include the `rtc-device.yaml` binding, which includes the `base.yaml` binding and the required `alarms-count` property.

```
include: rtc-device.yaml
```

Device driver design

Driver init RTCs are never powered off by the system. On init, drivers should expect the RTC is either:

- Powered, configured, and running.
- Powered, unconfigured, and stopped.

On init, drivers shall ensure the RTC is configured correctly, while preserving the time, alarms, and running status.

Time is set by calling `rtc_set_time()`, at which point the RTC will start running. Alarms pending status is cleared by `rtc_alarm_is_pending()` or by an alarm callback resulting from `rtc_alarm_set_callback()`.

GPIO routed interrupts RTCs with connected interrupt output pins shall configure and enable them on init. The host may enable and disable interrupts for its GPIOs, but RTC interrupt output pins must remain enabled.

This ensures consistent behavior between internally and externally connected RTCs, and allows for system wakeup through GPIO on any enabled RTC event like alarms or update.

> **Note**
>
> RTCs with unconnected interrupt output pins are not allowed to periodically poll the RTC to emulate it being connected. `rtc_alarm_set_callback()` and `rtc_update_set_callback()` shall return `-ENOTSUP` in this case.

Clock output If supported, the clock output configuration is defined in the devicetree. The output is configured by the driver on init.

Configuration Options Related configuration options:

- `CONFIG_RTC`
- `CONFIG_RTC_ALARM`
- `CONFIG_RTC_UPDATE`
- `CONFIG_RTC_CALIBRATION`

API Reference

RTC

> **Related code samples**
>
> - Real-Time Clock (RTC)Set and read the date/time from a Real-Time Clock.

Fake RTC

RTC device driver test suite The test suite validates the behavior of the RTC device driver. It is designed to be portable between boards. It uses the device tree alias `rtc` to designate the RTC device to test.

This test suite tests the following:

- Setting and getting the time.
- RTC Time incrementing correctly.
- Alarms if supported by hardware, with and without callback enabled

- Calibration if supported by hardware.

The calibration test tests a range of values which are printed to the console to be manually compared. The user must review the set and gotten values to ensure they are valid.

By default, only the mandatory setting and getting of time is enabled for testing. To test the optional alarms, update event callback and clock calibration, these must be enabled by selecting `CONFIG_RTC_ALARM`, `CONFIG_RTC_UPDATE` and `CONFIG_RTC_CALIBRATION`.

The following examples build the test suite for the `native_sim` board. To build the test suite for a different board, replace the `native_sim` board with your board.

To build the test application with the default configuration, testing only the mandatory features, the following command can be used for reference:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/rtc/rtc_api
```

To build the test with additional RTC features enabled, use menuconfig to enable the additional features by updating the configuration. The following command can be used for reference:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/rtc/rtc_api
west build -t menuconfig
```

Then build the test application using the following command:

```
# From the root of the zephyr repository
west build -b native_sim tests/drivers/rtc/rtc_api
```

To run the test suite, flash and run the application on your board, the output will be printed to the console.

> **Note**
>
> The tests take up to 30 seconds each if they are testing real hardware.

RTC emulated device The emulated RTC device fully implements the RTC API, and will behave like a real RTC device, with the following limitations:

- RTC time is not persistent across application initialization.
- RTC alarms are not persistent across application initialization.
- RTC time will drift over time.

Every time an application is initialized, the RTC's time and alarms are reset. Reading the time using `rtc_get_time()` will return `-ENODATA`, until the time is set using `rtc_set_time()`. The RTC will then behave as a real RTC, until the application is reset.

The emulated RTC device driver is built for the compatible `zephyr,rtc-emul` and will be included if `CONFIG_RTC` is selected.

History of RTCs in Zephyr RTCs have been supported before this API was created, using the *Counter* API. The unix timestamp was used to convert between broken-down time and the unix timestamp within the RTC drivers, which internally used the broken-down time representation.

The disadvantages of this approach were that hardware counters could not be set to a specific count, requiring all RTCs to use device specific APIs to set the time, converting from unix time to broken-down time, unnecessarily in some cases, and some common features missing, like input clock calibration and the update callback.

Watchdog

Overview

API Reference

Watchdog

Related code samples

- WatchdogUse the watchdog driver API to reset the board when it gets stuck in an infinite loop.

6.8 Pin Control

This is a high-level guide to pin control. See *Pin Control API* for API reference material.

6.8.1 Introduction

The hardware blocks that control pin multiplexing and pin configuration parameters such as pin direction, pull-up/down resistors, etc. are named **pin controllers**. The pin controller's main users are SoC hardware peripherals, since the controller enables exposing peripheral signals, like for example, map `I2C0 SDA` signal to pin `PX0`. Not only that, but it usually allows configuring certain pin settings that are necessary for the correct functioning of a peripheral, for example, the slew-rate depending on the operating frequency. The available configuration options are vendor/SoC dependent and can range from simple pull-up/down options to more advanced settings such as debouncing, low-power modes, etc.

The way pin control is implemented in hardware is vendor/SoC specific. It is common to find a *centralized* approach, that is, all pin configuration parameters are controlled by a single hardware block (typically named pinmux), including signal mapping. The figure below illustrates this approach. `PX0` can be mapped to `UART0_TX`, `I2C0_SCK` or `SPI0_MOSI` depending on the `AF` control bits. Other configuration parameters such as pull-up/down are controlled in the same block via `CONFIG` bits. This model is used by several SoC families, such as many from NXP and STM32.

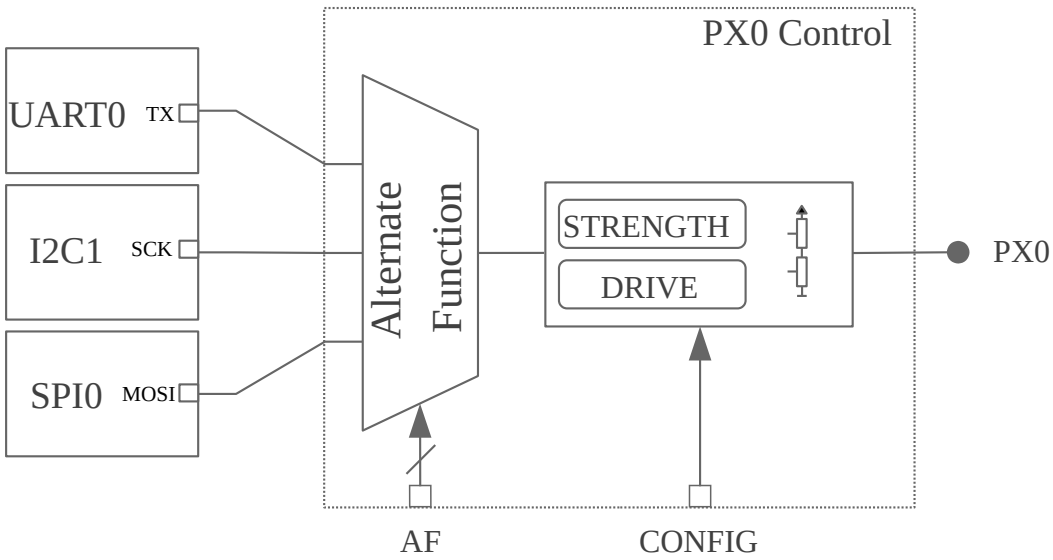


Fig. 3: Example of pin control centralized into a single per-pin block

Other vendors/SoCs use a *distributed* approach. In such case, the pin mapping and configuration are controlled by multiple hardware blocks. The figure below illustrates a distributed approach where pin mapping is controlled by peripherals, such as in Nordic nRF SoCs.

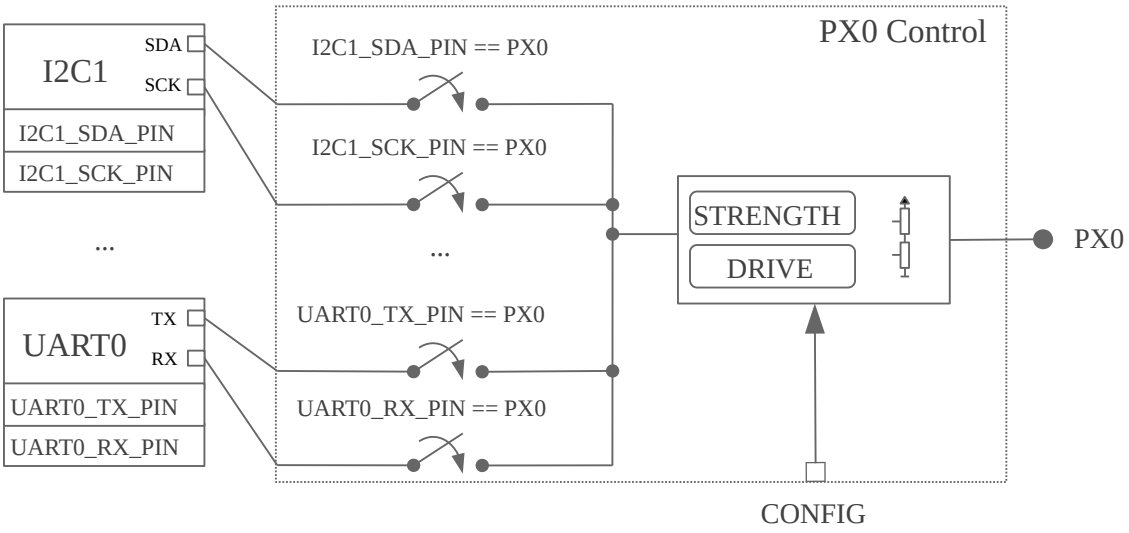


Fig. 4: Example pin control distributed between peripheral registers and per-pin block

From a user perspective, there is no difference in pin controller usage regardless of the hardware implementation: a user will always apply a state. The only difference lies in the driver implementation. In general, implementing a pin controller driver for a hardware that uses a distributed approach requires more effort, since the driver needs to gather knowledge of peripheral dependent registers.

Pin control vs. GPIO

Some functionality covered by a pin controller driver overlaps with GPIO drivers. For example, pull-up/down resistors can usually be enabled by both the pin control driver and the GPIO driver. In Zephyr context, the pin control driver purpose is to perform peripheral signal multiplexing and configuration of other pin parameters required for the correct operation of that peripheral. Therefore, the main users of the pin control driver are SoC peripherals. In contrast, GPIO drivers are for general purpose control of a pin, that is, when its logic level is read or controlled manually.

6.8.2 State model

For a device driver to operate correctly, a certain pin configuration needs to be applied. Some device drivers require a static configuration, usually set up at initialization time. Others need to change the configuration at runtime depending on the operating conditions, for example, to enable a low-power mode when suspending the device. Such requirements are modeled using **states**, a concept that has been adapted from the one in the Linux kernel. Each device driver owns a set of states. Each state has a unique name and contains a full pin configuration set (see the figure below). This effectively means that states are independent of each other, so they do not need to be applied in any specific order. Another advantage of the state model is that it isolates device drivers from pin configuration.

Table 3: Example pin configuration encoded using the states model

UART0 peripheral			
default state		sleep state	
TX	• Pin: PA0 • Pull: NONE • Low Power: NO	TX	• Pin: PA0 • Pull: NONE • Low Power: YES
RX	• Pin: PA1 • Pull: UP • Low Power: NO	RX	• Pin: PA1 • Pull: NONE • Low Power: YES

Standard states

The name assigned to pin control states or the number of them is up to the device driver requirements. In many cases a single state applied at initialization time will be sufficient, but in some other cases more will be required. In order to make things consistent, a naming convention has been established for the most common use cases. The figure below details the standardized states and its purpose.

Table 4: Standardized state names

State	Identifier	Purpose
default	PINCTRL_STATE_DEFAULT	State of the pins when the device is in operational state
sleep	PINCTRL_STATE_SLEEP	State of the pins when the device is in low power or sleep modes

Note that other standard states could be introduced in the future.

Custom states

Some device drivers may require using custom states beyond the standard ones. To achieve that, the device driver needs to have in its scope definitions for the custom state identifiers named as PINCTRL_STATE_{STATE_NAME}, where {STATE_NAME} is the capitalized state name. For example, if mystate has to be supported, a definition named PINCTRL_STATE_MYSTATE needs to be in the driver's scope.

> **Note**
>
> It is important that custom state identifiers start from PINCTRL_STATE_PRIV_START

If custom states need to be accessed from outside the driver, for example to perform dynamic pin control, custom identifiers should be placed in a header that is publicly accessible.

Skipping states

In most situations, the states defined in Devicetree will be the ones used in the compiled firmware. However, there are some cases where certain states will be conditionally used depending on a compilation flag. A typical case is the sleep state. This state is only used in practice if CONFIG_PM or CONFIG_PM_DEVICE is enabled. If a firmware variant without these power management configurations is needed, one should in theory remove the sleep state from Devicetree to not waste ROM space storing such unused state.

States can be skipped by the pinctrl Devicetree macros if a definition named PINCTRL_SKIP_{STATE_NAME} expanding to 1 is present when pin control configuration is defined.

In case of the `sleep` state, the `pinctrl` API already provides such definition conditional to the availability of device power management:

```
#if !defined(CONFIG_PM) && !defined(CONFIG_PM_DEVICE)
/** Out of power management configurations, ignore "sleep" state. */
#define PINCTRL_SKIP_SLEEP 1
#endif
```

6.8.3 Dynamic pin control

Dynamic pin control refers to the capability of changing pin configuration at runtime. This feature can be useful in situations where the same firmware needs to run onto slightly different boards, each having a peripheral routed at a different set of pins. This feature can be enabled by setting `CONFIG_PINCTRL_DYNAMIC`.

> **Note**
>
> Dynamic pin control should only be used on devices that have not been initialized. Changing pin configurations while a device is operating may lead to unexpected behavior. Since Zephyr does not support device de-initialization yet, this functionality should only be used during early boot stages.

One of the effects of enabling dynamic pin control is that `pinctrl_dev_config` will be stored in RAM instead of ROM (not states or pin configurations, though). The user can then use `pinctrl_update_states()` to update the states stored in `pinctrl_dev_config` with a new set. This effectively means that the device driver will apply the pin configurations stored in the updated states when it applies a state.

6.8.4 Devicetree representation

Because Devicetree is meant to describe hardware, it is the natural choice when it comes to storing pin control configuration. In the following sections you will find an overview on how states and pin configurations are represented in Devicetree.

States

Given a device, each of its pin control state is represented in Devicetree by `pinctrl-N` properties, being `N` the state index starting from zero. The `pinctrl-names` property is then used to assign a unique identifier for each state property by index, for example, `pinctrl-names` list entry 0 is the name for `pinctrl-0`.

```
periph0: periph@0 {
    ...
    /* state 0 ("default") */
    pinctrl-0 = <...>;
    ...
    /* state N ("mystate") */
    pinctrl-N = <...>;
    /* names for state 0 up to state N */
    pinctrl-names = "default", ..., "mystate";
    ...
};
```

Pin configuration

There are multiple ways to represent the pin configurations in Devicetree. However, all end up encoding the same information: the pin multiplexing and the pin configuration parameters. For example, `UART_RX` is mapped to `PX0` and pull-up is enabled. The representation choice largely depends on each vendor/SoC, so the Devicetree binding files for the pin control drivers are the best place to look for details.

A popular and versatile option is shown in the example below. One of the advantages of this choice is the grouping capability based on shared pin configuration. This allows to reduce the verbosity of the pin control definitions. Another advantage is that the pin configuration parameters for a particular state are enclosed in a single Devicetree node.

```
/* board.dts */
#include "board-pinctrl.dtsi"

&periph0 {
    pinctrl-0 = <&periph0_default>;
    pinctrl-names = "default";
};
```

```
/* vnd-soc-pkgxx.h
 * File with valid mappings for a specific package (may be autogenerated).
 * This file is optional, but recommended.
 */
...
#define PERIPH0_SIGA_PX0 VNDSOC_PIN(X, 0, MUX0)
#define PERIPH0_SIGB_PY7 VNDSOC_PIN(Y, 7, MUX4)
#define PERIPH0_SIGC_PZ1 VNDSOC_PIN(Z, 1, MUX2)
...
```

```
/* board-pinctrl.dtsi */
#include <vnd-soc-pkgxx.h>

&pinctrl {
    /* Node with pin configuration for default state */
    periph0_default: periph0_default {
        group1 {
            /* Mappings: PERIPH0_SIGA -> PX0, PERIPH0_SIGC -> PZ1 */
            pinmux = <PERIPH0_SIGA_PX0>, <PERIPH0_SIGC_PZ1>;
            /* Pins PX0 and PZ1 have pull-up enabled */
            bias-pull-up;
        };
        ...
        groupN {
            /* Mappings: PERIPH0_SIGB -> PY7 */
            pinmux = <PERIPH0_SIGB_PY7>;
        };
    };
};
```

Another popular model is based on having a node for each pin configuration and state. While this model may lead to shorter board pin control files, it also requires to have one node for each pin mapping and state, since in general, nodes can not be reused for multiple states. This method is discouraged if autogeneration is not an option.

Note

Because all Devicetree information is parsed into a C header, it is important to make sure its size is kept to a minimum. For this reason it is important to prefix pre-generated nodes with `/omit-if-no-ref/`. This prefix makes sure that the node is discarded when not used.

```
/* board.dts */
#include "board-pinctrl.dtsi"

&periph0 {
    pinctrl-0 = <&periph0_siga_px0_default &periph0_sigb_py7_default
                 &periph0_sigc_pz1_default>;
    pinctrl-names = "default";
};
```

```
/* vnd-soc-pkgxx.dtsi
 * File with valid nodes for a specific package (may be autogenerated).
 * This file is optional, but recommended.
 */

&pinctrl {
    /* Mapping for PERIPH0_SIGA -> PX0, to be used for default state */
    /omit-if-no-ref/ periph0_siga_px0_default: periph0_siga_px0_default {
        pinmux = <VNDSOC_PIN(X, 0, MUX0)>;
    };

    /* Mapping for PERIPH0_SIGB -> PY7, to be used for default state */
    /omit-if-no-ref/ periph0_sigb_py7_default: periph0_sigb_py7_default {
        pinmux = <VNDSOC_PIN(Y, 7, MUX4)>;
    };

    /* Mapping for PERIPH0_SIGC -> PZ1, to be used for default state */
    /omit-if-no-ref/ periph0_sigc_pz1_default: periph0_sigc_pz1_default {
        pinmux = <VNDSOC_PIN(Z, 1, MUX2)>;
    };
};
```

```
/* board-pinctrl.dts */
#include <vnd-soc-pkgxx.dtsi>

/* Enable pull-up for PX0 (default state) */
&periph0_siga_px0_default {
    bias-pull-up;
};

/* Enable pull-up for PZ1 (default state) */
&periph0_sigc_pz1_default {
    bias-pull-up;
};
```

Note

It is discouraged to add pin configuration defaults in pre-defined nodes. In general, pin configurations depend on the board design or on the peripheral working conditions, so the decision should be made by the board. For example, enabling a pull-up by default may not always be

desired because the board already has one or because its value depends on the operating bus speed. Another downside of defaults is that user may not be aware of them, for example:

```
/* not evident that "periph0_siga_px0_default" also implies "bias-pull-up" */
/omit-if-no-ref/ periph0_siga_px0_default: periph0_siga_px0_default {
    pinmux = <VNDSOC_PIN(X, 0, MUX0)>;
    bias-pull-up;
};
```

6.8.5 Implementation guidelines

Pin control drivers

Pin control drivers need to implement a single function: `pinctrl_configure_pins()`. This function receives an array of pin configurations that need to be applied. Furthermore, if `CONFIG_PINCTRL_STORE_REG` is set, it also receives the associated device register address for the given pins. This information may be required by some drivers to perform device specific actions.

The pin configuration is stored in an opaque type that is vendor/SoC dependent: `pinctrl_soc_pin_t`. This type needs to be defined in a header named `pinctrl_soc.h` file that is in the Zephyr's include path. It can range from a simple integer value to a struct with multiple fields. `pinctrl_soc.h` also needs to define a macro named `Z_PINCTRL_STATE_PINS_INIT` that accepts two arguments: a node identifier and a property name (`pinctrl-N`). With this information the macro needs to define an initializer for all pin configurations contained within the `pinctrl-N` property of the given node.

Regarding Devicetree pin configuration representation, vendors can decide which option is better for their devices. However, the following guidelines should be followed:

- Use `pinctrl-N` (N=0, 1, ...) and `pinctrl-names` properties to define pin control states. These properties are defined in `dts/bindings/pinctrl/pinctrl-device.yaml`.
- Use standard pin configuration properties as defined in `dts/bindings/pinctrl/pincfg-node.yaml`.

Representations not following these guidelines may be accepted if they are already used by the same vendor in other operating systems, e.g. Linux.

Device drivers

In this section you will find some tips on how a device driver should use the `pinctrl` API to successfully configure the pins it needs.

The device compatible needs to be modified in the corresponding binding so that the `pinctrl-device.yaml` is included. For example:

```
include: [base.yaml, pinctrl-device.yaml]
```

This file is needed to add `pinctrl-N` and `pinctrl-names` properties to the device.

From a device driver perspective there are two steps that need to be performed to be able to use the `pinctrl` API. First, the pin control configuration needs to be defined. This includes all states and pins. `PINCTRL_DT_DEFINE` or `PINCTRL_DT_INST_DEFINE` macros should be used for this purpose. Second, a reference to the device instance `pinctrl_dev_config` needs to be stored, since it is required to later use the API. This can be achieved using the `PINCTRL_DT_DEV_CONFIG_GET` and `PINCTRL_DT_INST_DEV_CONFIG_GET` macros.

It is worth to note that the only relationship between a device and its associated pin control configuration is based on variable naming conventions. The way an instance of `pinctrl_dev_config` is named for a corresponding device instance allows to later obtain a reference to it given the device's Devicetree node identifier. This allows to minimize ROM usage, since only devices requiring pin control will own a reference to a pin control configuration.

Once the driver has defined the pin control configuration and kept a reference to it, it is ready to use the API. The most common way to apply a state is by using `pinctrl_apply_state()`. It is also possible to use the lower level function `pinctrl_apply_state_direct()` to skip state lookup if it is cached in advance (e.g. at init time). Since state lookup time is expected to be fast, it is recommended to use `pinctrl_apply_state()`.

The example below contains a complete example of a device driver that uses the `pinctrl` API.

```
/* A driver for the "mydev" compatible device */
#define DT_DRV_COMPAT mydev

...
#include <zephyr/drivers/pinctrl.h>
...

struct mydev_config {
    ...
    /* Reference to mydev pinctrl configuration */
    const struct pinctrl_dev_config *pcfg;
    ...
};

...

static int mydev_init(const struct device *dev)
{
    const struct mydev_config *config = dev->config;
    int ret;
    ...
    /* Select "default" state at initialization time */
    ret = pinctrl_apply_state(config->pcfg, PINCTRL_STATE_DEFAULT);
    if (ret < 0) {
        return ret;
    }
    ...
}

#define MYDEV_DEFINE(i)                                                     \
    /* Define all pinctrl configuration for instance "i" */                 \
    PINCTRL_DT_INST_DEFINE(i);                                              \
    ...                                                                     \
    static const struct mydev_config mydev_config_##i = {                   \
        ...                                                                 \
        /* Keep a ref. to the pinctrl configuration for instance "i" */     \
        .pcfg = PINCTRL_DT_INST_DEV_CONFIG_GET(i),                          \
        ...                                                                 \
    };                                                                      \
    ...                                                                     \
                                                                            \
    DEVICE_DT_INST_DEFINE(i, mydev_init, NULL, &mydev_data##i,              \
                          &mydev_config##i, ...);

DT_INST_FOREACH_STATUS_OKAY(MYDEV_DEFINE)
```

6.8.6 Pin Control API

Pin Control

Related code samples

- Master Clock Output (MCO)Output an internal clock for external use by the application.

Dynamic pin control

Dynamic Pin Control

6.8.7 Other reference material

- Introduction to pin muxing and GPIO control under Linux

6.9 Porting

These pages document how to port Zephyr to new hardware.

6.9.1 Architecture Porting Guide

An architecture port is needed to enable Zephyr to run on an ISA (instruction set architecture) or an ABI (Application Binary Interface) that is not currently supported.

The following are examples of ISAs and ABIs that Zephyr supports:

- x86_32 ISA with System V ABI
- ARMv7-M ISA with Thumb2 instruction set and ARM Embedded ABI (aeabi)
- ARCv2 ISA

For information on Kconfig configuration, see *Setting Kconfig configuration values*. Architectures use a Kconfig configuration scheme similar to boards.

An architecture port can be divided in several parts; most are required and some are optional:

- **The early boot sequence**: each architecture has different steps it must take when the CPU comes out of reset (required).
- **Interrupt and exception handling**: each architecture handles asynchronous and unrequested events in a specific manner (required).
- **Thread context switching**: the Zephyr context switch is dependent on the ABI and each ISA has a different set of registers to save (required).
- **Thread creation and termination**: A thread's initial stack frame is ABI and architecture-dependent, and thread abortion possibly as well (required).
- **Device drivers**: most often, the system clock timer and the interrupt controller are tied to the architecture (some required, some optional).
- **Utility libraries**: some common kernel APIs rely on a architecture-specific implementation for performance reasons (required).
- **CPU idling/power management**: most architectures implement instructions for putting the CPU to sleep (partly optional, most likely very desired).
- **Fault management**: for implementing architecture-specific debug help and handling of fatal error in threads (partly optional).
- **Linker scripts and toolchains**: architecture-specific details will most likely be needed in the build system and when linking the image (required).
- **Memory Management and Memory Mapping**: for architecture-specific details on supporting memory management and memory mapping.

- **Stack Objects**: for architecture-specific details on memory protection hardware regarding stack objects.
- **User Mode Threads**: for supporting threads in user mode.
- **GDB Stub**: for supporting GDB stub to enable remote debugging.

Early Boot Sequence

The goal of the early boot sequence is to take the system from the state it is after reset to a state where is can run C code and thus the common kernel initialization sequence. Most of the time, very few steps are needed, while some architectures require a bit more work to be performed.

Common steps for all architectures:

- Setup an initial stack.
- If running an XIP (eXecute-In-Place) kernel, copy initialized data from ROM to RAM.
- If not using an ELF loader, zero the BSS section.
- Jump to `z_cstart()`, the early kernel initialization
  - `z_cstart()` is responsible for context switching out of the fake context running at startup into the main thread.

Some examples of architecture-specific steps that have to be taken:

- If given control in real mode on x86_32, switch to 32-bit protected mode.
- Setup the segment registers on x86_32 to handle boot loaders that leave them in an unknown or broken state.
- Initialize a board-specific watchdog on Cortex-M3/4.
- Switch stacks from MSP to PSP on Cortex-M.
- Use a different approach than calling into z_swap() on Cortex-M to prevent race conditions.
- Setup FIRQ and regular IRQ handling on ARCv2.

Early Boot Sequence Hooks Zephyr exposes several hooks (described in include/zephyr/platform/hooks.h) that allow execution of SoC- or board-specific code at precise moments of the boot process.

The kernel takes care of calling most of the hooks from architecture-agnostic code. However, some hooks must be called during the early boot sequence; since the sequence is implemented in architecture-specific code, the call to the hooks must also be done there. The following gives a rough overview of the early boot sequence and when hooks should be called by architecture-specific code:

1. Execution begins in an architecture-specific entry point whose name matches `CONFIG_KERNEL_ENTRY`.
2. Architecture-specific state is re-initialized immediately (if `CONFIG_INIT_ARCH_HW_AT_BOOT` is enabled).
3. `soc_early_reset_hook()` is called.

> **Note**
>
> It is not necessary to set up a valid stack before calling this hook. However, the hook is allowed to overwrite the stack pointer before returning. The architecture-specific code must not expect the stack pointer register value to be preserved across the call to `soc_early_reset_hook()`.

On architectures with multiple stack pointers, there is usually a *"primary"* stack pointer accessible directly and *"secondary"* stack pointer register(s). `soc_early_reset_hook()` implementations may overwrite the *"primary"* stack pointer must **not** read or modify the value of any *"secondary"* stack pointer. (This allows the architecture-specific code to set up any *"secondary"* stack pointer it desires before calling `soc_early_reset_hook()`)

For example, the ARM Cortex-A architecture defines several execution modes, each of which has its own stack pointer register `sp_mode`. When the processor is executing in mode `X`, operations involving the `sp` general-purpose register operate on `sp_X`. On this architecture, assuming that the processor is executing in mode `M` when `soc_early_reset_hook()` is called, the hook is allowed to overwrite `sp_M` (accessible through `sp`) but **must not** read or overwrite any other `sp_mode` (where `mode != M`).

`soc_early_reset_hook()` implementations are allowed to not return execution to the architecture-specific code, in which case they "take over" the system. Such hooks are not subject to the aforementioned rules and may read or overwrite any stack pointer. However, when such an implementation is provided, the rest of the early boot sequence obviously does not execute.

4. An initial stack is set up for next steps of the early boot sequence.
5. Architecture-specific *"resume from suspend-to-RAM"* logic is executed

Note

Refer to `CONFIG_PM_S2RAM` and the architecture-specific implementation for more details, but note that the rest of the early boot sequence is not executed if this logic determines that an exit from suspend-to-RAM is ongoing.

6. `soc_reset_hook()` is called.
7. *Architecture-specific operations (in assembly) are performed here...*
8. `z_prep_c()` is called. This architecture-specific function is implemented in C.
9. `z_prep_c()` immediately calls `soc_prep_hook()`.
10. *Architecture-specific operations (in C) are performed here...*
11. `z_cstart()` is called. Architecture-agnostic code begins executing.

Interrupt and Exception Handling

Each architecture defines interrupt and exception handling differently.

When a device wants to signal the processor that there is some work to be done on its behalf, it raises an interrupt. When a thread does an operation that is not handled by the serial flow of the software itself, it raises an exception. Both, interrupts and exceptions, pass control to a handler. The handler is known as an ISR (Interrupt Service Routine) in the case of interrupts. The handler performs the work required by the exception or the interrupt. For interrupts, that work is device-specific. For exceptions, it depends on the exception, but most often the core kernel itself is responsible for providing the handler.

The kernel has to perform some work in addition to the work the handler itself performs. For example:

- Prior to handing control to the handler:
  - Save the currently executing context.
  - Possibly getting out of power saving mode, which includes waking up devices.
  - Updating the kernel uptime if getting out of tickless idle mode.

- After getting control back from the handler:
  - Decide whether to perform a context switch.
  - When performing a context switch, restore the context being context switched in.

This work is conceptually the same across architectures, but the details are completely different:

- The registers to save and restore.
- The processor instructions to perform the work.
- The numbering of the exceptions.
- etc.

It thus needs an architecture-specific implementation, called the interrupt/exception stub.

Another issue is that the kernel defines the signature of ISRs as:

```
void (*isr)(void *parameter)
```

Architectures do not have a consistent or native way of handling parameters to an ISR. As such there are two commonly used methods for handling the parameter.

- Using some architecture defined mechanism, the parameter value is forced in the stub. This is commonly found in X86-based architectures.
- The parameters to the ISR are inserted and tracked via a separate table requiring the architecture to discover at runtime which interrupt is executing. A common interrupt handler demuxer is installed for all entries of the real interrupt vector table, which then fetches the device's ISR and parameter from the separate table. This approach is commonly used in the ARC and ARM architectures via the `CONFIG_GEN_ISR_TABLES` implementation. You can find examples of the stubs by looking at `_interrupt_enter()` in x86, `_isr_wrapper()` in ARM, or the full implementation description for ARC in arch/arc/core/isr_wrapper.S.

Each architecture also has to implement primitives for interrupt control:

- locking interrupts: `irq_lock()`, `irq_unlock()`.
- registering interrupts: `IRQ_CONNECT()`.
- programming the priority if possible `irq_priority_set()`.
- enabling/disabling interrupts: `irq_enable()`, `irq_disable()`.

Note

`IRQ_CONNECT` is a macro that uses assembler and/or linker script tricks to connect interrupts at build time, saving boot time and text size.

The vector table should contain a handler for each interrupt and exception that can possibly occur. The handler can be as simple as a spinning loop. However, we strongly suggest that handlers at least print some debug information. The information helps figuring out what went wrong when hitting an exception that is a fault, like divide-by-zero or invalid memory access, or an interrupt that is not expected (*spurious interrupt*). See the ARM implementation in arch/arm/core/cortex_m/fault.c for an example.

Thread Context Switching

Multi-threading is the basic purpose to have a kernel at all. Zephyr supports two types of threads: preemptible and cooperative. The rules for determining the next thread to schedule are handled by the kernel. However, it is up to the architecture port to implement the method of the context switch itself.

Zephyr provides two mutually exclusive interfaces for context switching. The preferred interface to use is `arch_switch` which is selected when `CONFIG_USE_SWITCH` is enabled. The alternative interface is `arch_swap`–selected when `CONFIG_USE_SWITCH` is disabled. When porting to a new architecture, only one of these needs to implemented; however, for SMP platforms it must be `arch_switch`.

A context switch can happen in several circumstances:

- When a thread executes a blocking operation, such as taking a semaphore that is currently unavailable.
- When a preemptible thread unblocks a thread of higher priority by releasing the object on which it was blocked.
- When an interrupt unblocks a thread of higher priority than the one currently executing, if the currently executing thread is preemptible.
- When a thread runs to completion.
- When a thread causes a fatal exception and is removed from the running threads. For example, referencing invalid memory,

Therefore, the context switching must thus be able to handle all these cases.

There are two types of context switches: *cooperative* and *preemptive*.

- A *cooperative* context switch happens when a thread willfully gives the control to another thread. There are two cases where this happens
  - When a thread explicitly yields.
  - When a thread tries to take an object that is currently unavailable and is willing to wait until the object becomes available.
- A *preemptive* context switch happens either because an ISR or a thread causes an operation that schedules a thread of higher priority than the one currently running, if the currently running thread is preemptible. An example of such an operation is releasing an object on which the thread of higher priority was waiting.

> **Note**
>
> Control is never taken from cooperative thread when one of them is the running thread.

A cooperative context switch is always done by having a thread call the internal kernel routine `z_swap` (or one of its variants). This in turn will call either `arch_switch` or `arch_swap` as appropriate. When these are called, no checks are done to determine if the context switch is to happen–the context switch must happen.

> **Note**
>
> On 32-bit x86, `arch_swap` is generic enough and the architecture flexible enough that it can be called when exiting an interrupt to provoke the context switch. This should not be taken as a rule, since neither the ARM Cortex-M nor ARCv2 port do this.

Since `z_swap` is cooperative, the caller-saved registers from the ABI are already on the stack. There is no need to save them in the k_thread structure.

A context switch can also be performed preemptively. This happens upon exiting an ISR, in the kernel interrupt exit stub:

- `_interrupt_enter` on x86 after the handler is called.
- `z_arm_exc_exit` and `z_arm_int_exit` on ARM.

- `_firq_exit` and `_rirq_exit` on ARCv2.

The decision logic to invoke the context switch is simple and is only performed when exiting a non-nested interrupt:

When `CONFIG_USE_SWITCH` is enabled ...

- The interrupt exit code shall call `z_get_next_switch_handle()`, and return to the thread context identified by the returned switch handle

When `CONFIG_USE_SWITCH` is not enabled ...

- The interrupt exit code shall fetch the cached thread from the ready queue, and:
  - If the cached thread is not the current thread, invoke the context switch.
  - Otherwise do not invoke it.

This is simple, but crucial: if this is not implemented correctly, the kernel will not function as intended and will experience bizarre crashes, mostly due to stack corruption.

Thread Creation and Termination

To start a new thread, a stack frame must be constructed so that the context switch can pop it the same way it would pop one from a thread that had been context switched out. This is to be implemented in an architecture-specific `_new_thread` internal routine.

The thread entry point is also not to be called directly, i.e. it should not be set as the PC (program counter) for the new thread. Rather it must be wrapped in `_thread_entry`. This means that the PC in the stack frame shall be set to `_thread_entry`, and the thread entry point shall be passed as the first parameter to `_thread_entry`. The specifics of this depend on the ABI.

The need for an architecture-specific thread termination implementation depends on the architecture. There is a generic implementation, but it might not work for a given architecture.

One reason that has been encountered for having an architecture-specific implementation of thread termination is that aborting a thread might be different if aborting because of a graceful exit or because of an exception. This is the case for ARM Cortex-M, where the CPU has to be taken out of handler mode if the thread triggered a fatal exception, but not if the thread gracefully exits its entry point function.

This means implementing an architecture-specific version of `k_thread_abort()`, and setting the Kconfig option `CONFIG_ARCH_HAS_THREAD_ABORT` as needed for the architecture (e.g. see arch/arm/core/cortex_m/Kconfig).

Thread Local Storage

To enable thread local storage on a new architecture:

1. Implement `arch_tls_stack_setup()` to setup the TLS storage area in stack. Refer to the toolchain documentation on how the storage area needs to be structured. Some helper functions can be used:
   - Function `z_tls_data_size()` returns the size needed for thread local variables (excluding any extra data required by toolchain and architecture).
   - Function `z_tls_copy()` prepares the TLS storage area for thread local variables. This only copies the variable themselves and does not do architecture and/or toolchain specific data.
2. In the context switching, grab the `tls` field inside the new thread's `struct k_thread` and put it into an appropriate register (or some other variable) for access to the TLS storage area. Refer to toolchain and architecture documentation on which registers to use.
3. In kconfig, add `select ARCH_HAS_THREAD_LOCAL_STORAGE` to kconfig related to the new architecture.

4. Run the `tests/kernel/threads/tls` to make sure the new code works.

Device Drivers

The kernel requires very few hardware devices to function. In theory, the only required device is the interrupt controller, since the kernel can run without a system clock. In practice, to get access to most, if not all, of the sanity check test suite, a system clock is needed as well. Since these two are usually tied to the architecture, they are part of the architecture port.

Interrupt Controllers There can be significant differences between the interrupt controllers and the interrupt concepts across architectures.

For example, x86 has the concept of an IDT and different interrupt controllers. The position of an interrupt in the IDT determines its priority.

On the other hand, the ARM Cortex-M has the NVIC (Nested Vectored Interrupt Controller) as part of the architecture definition. There is no need for an IDT-like table that is separate from the NVIC vector table. The position in the table has nothing to do with priority of an IRQ: priorities are programmable per-entry.

The ARCv2 has its interrupt unit as part of the architecture definition, which is somewhat similar to the NVIC. However, where ARC defines interrupts as having a one-to-one mapping between exception and interrupt numbers (i.e. exception 1 is IRQ1, and device IRQs start at 16), ARM has IRQ0 being equivalent to exception 16 (and weirdly enough, exception 1 can be seen as IRQ-15).

All these differences mean that very little, if anything, can be shared between architectures with regards to interrupt controllers.

System Clock x86 has APIC timers and the HPET as part of its architecture definition. ARM Cortex-M has the SYSTICK exception. Finally, ARCv2 has the timer0/1 device.

Kernel timeouts are handled in the context of the system clock timer driver's interrupt handler.

Console Over Serial Line There is one other device that is almost a requirement for an architecture port, since it is so useful for debugging. It is a simple polling, output-only, serial port driver on which to send the console (`printk`, `printf`) output.

It is not required, and a RAM console (`CONFIG_RAM_CONSOLE`) can be used to send all output to a circular buffer that can be read by a debugger instead.

Utility Libraries

The kernel depends on a few functions that can be implemented with very few instructions or in a lock-less manner in modern processors. Those are thus expected to be implemented as part of an architecture port.

- Atomic operators.
  - If instructions do exist for a given architecture, the implementation is configured using the `CONFIG_ATOMIC_OPERATIONS_ARCH` Kconfig option.
  - If instructions do not exist for a given architecture, a generic version that wraps `irq_lock()` or `irq_unlock()` around non-atomic operations exists. It is configured using the `CONFIG_ATOMIC_OPERATIONS_C` Kconfig option.
- Find-least-significant-bit-set and find-most-significant-bit-set.
  - If instructions do not exist for a given architecture, it is always possible to implement these functions as generic C functions.

It is possible to use compiler built-ins to implement these, but be careful they use the required compiler barriers.

CPU Idling/Power Management

The kernel provides support for CPU power management with two functions: `arch_cpu_idle()` and `arch_cpu_atomic_idle()`.

`arch_cpu_idle()` can be as simple as calling the power saving instruction for the architecture with interrupts unlocked, for example `hlt` on x86, `wfi` or `wfe` on ARM, `sleep` on ARC. This function can be called in a loop within a context that does not care if it get interrupted or not by an interrupt before going to sleep. There are basically two scenarios when it is correct to use this function:

- In a single-threaded system, in the only thread when the thread is not used for doing real work after initialization, i.e. it is sitting in a loop doing nothing for the duration of the application.
- In the idle thread.

`arch_cpu_atomic_idle()`, on the other hand, must be able to atomically re-enable interrupts and invoke the power saving instruction. It can thus be used in real application code, again in single-threaded systems.

Normally, idling the CPU should be left to the idle thread, but in some very special scenarios, these APIs can be used by applications.

Both functions must exist for a given architecture. However, the implementation can be simply the following steps, if desired:

1. unlock interrupts
2. NOP

However, a real implementation is strongly recommended.

Fault Management

In the event of an unhandled CPU exception, the architecture code must call into `z_fatal_error()`. This function dumps out architecture-agnostic information and makes a policy decision on what to do next by invoking `k_sys_fatal_error()`. This function can be overridden to implement application-specific policies that could include locking interrupts and spinning forever (the default implementation) or even powering off the system (if supported).

Toolchain and Linking

Toolchain support has to be added to the build system.

Some architecture-specific definitions are needed in include/zephyr/toolchain/gcc.h. See what exists in that file for currently supported architectures.

Each architecture also needs its own linker script, even if most sections can be derived from the linker scripts of other architectures. Some sections might be specific to the new architecture, for example the SCB section on ARM and the IDT section on x86.

Memory Management and Memory Mapping

If the target platform enables paging and requires drivers to memory-map their I/O regions, `CONFIG_MMU` needs to be enabled and the following API implemented:

- `arch_mem_map()`
- `arch_mem_unmap()`
- `arch_page_phys_get()`

Stack Objects

The presence of memory protection hardware affects how stack objects are created. All architecture ports must specify the required alignment of the stack pointer, which is some combination of CPU and ABI requirements. This is defined in architecture headers with ARCH_STACK_PTR_ALIGN and is typically something small like 4, 8, or 16 bytes.

Two types of thread stacks exist:

- "kernel" stacks defined with K_KERNEL_STACK_DEFINE() and related APIs, which can host kernel threads running in supervisor mode or used as the stack for interrupt/exception handling. These have significantly relaxed alignment requirements and use less reserved data. No memory is reserved for privilege elevation stacks.
- "thread" stacks which typically use more memory, but are capable of hosting thread running in user mode, as well as any use-cases for kernel stacks.

If CONFIG_USERSPACE is not enabled, "thread" and "kernel" stacks are equivalent.

Additional macros may be defined in the architecture layer to specify the alignment of the base of stack objects, any reserved data inside the stack object not used for the thread's stack buffer, and how to round up stack sizes to support user mode threads. In the absence of definitions some defaults are assumed:

- ARCH_KERNEL_STACK_RESERVED: default no reserved space
- ARCH_THREAD_STACK_RESERVED: default no reserved space
- ARCH_KERNEL_STACK_OBJ_ALIGN: default align to ARCH_STACK_PTR_ALIGN
- ARCH_THREAD_STACK_OBJ_ALIGN: default align to ARCH_STACK_PTR_ALIGN
- ARCH_THREAD_STACK_SIZE_ALIGN: default round up to ARCH_STACK_PTR_ALIGN

All stack creation macros are defined in terms of these.

Stack objects all have the following layout, with some regions potentially zero-sized depending on configuration. There are always two main parts: reserved memory at the beginning, and then the stack buffer itself. The bounds of some areas can only be determined at runtime in the context of its associated thread object. Other areas are entirely computable at build time.

Some architectures may need to carve-out reserved memory at runtime from the stack buffer, instead of unconditionally reserving it at build time, or to supplement an existing reserved area (as is the case with the ARM FPU). Such carve-outs will always be tracked in thread.stack_info.start. The region specified by thread.stack_info.start and thread.stack_info.size is always fully accessible by a user mode thread. thread.stack_info.delta denotes an offset which can be used to compute the initial stack pointer from the very end of the stack object, taking into account storage for TLS and ASLR random offsets.

```
+---------------------+ <- thread.stack_obj
| Reserved Memory     | } K_(THREAD|KERNEL)_STACK_RESERVED
+---------------------+
| Carved-out memory   |
|.....................| <- thread.stack_info.start
| Unused stack buffer |
|                     |
|.....................| <- thread's current stack pointer
| Used stack buffer   |
|                     |
|.....................| <- Initial stack pointer. Computable
| ASLR Random offset  |      with thread.stack_info.delta
+---------------------| <- thread.userspace_local_data
| Thread-local data   |
+---------------------+ <- thread.stack_info.start + thread.stack_info.size
```

At present, Zephyr does not support stacks that grow upward.

No Memory Protection If no memory protection is in use, then the defaults are sufficient.

HW-based stack overflow detection This option uses hardware features to generate a fatal error if a thread in supervisor mode overflows its stack. This is useful for debugging, although for a couple reasons, you can't reliably make any assertions about the state of the system after this happens:

- The kernel could have been inside a critical section when the overflow occurs, leaving important global data structures in a corrupted state.
- For systems that implement stack protection using a guard memory region, it's possible to overshoot the guard and corrupt adjacent data structures before the hardware detects this situation.

To enable the `CONFIG_HW_STACK_PROTECTION` feature, the system must provide some kind of hardware-based stack overflow protection, and enable the `CONFIG_ARCH_HAS_STACK_PROTECTION` option.

Two forms of HW-based stack overflow detection are supported: dedicated CPU features for this purpose, or special read-only guard regions immediately preceding stack buffers.

`CONFIG_HW_STACK_PROTECTION` only catches stack overflows for supervisor threads. This is not required to catch stack overflow from user threads; `CONFIG_USERSPACE` is orthogonal.

This feature only detects supervisor mode stack overflows, including stack overflows when handling system calls. It doesn't guarantee that the kernel has not been corrupted. Any stack overflow in supervisor mode should be treated as a fatal error, with no assertions about the integrity of the overall system possible.

Stack overflows in user mode are recoverable (from the kernel's perspective) and require no special configuration; `CONFIG_HW_STACK_PROTECTION` only applies to catching overflows when the CPU is in supervisor mode.

CPU-based stack overflow detection If we are detecting stack overflows in supervisor mode via special CPU registers (like ARM's SPLIM), then the defaults are sufficient.

Guard-based stack overflow detection We are detecting supervisor mode stack overflows via special memory protection region located immediately before the stack buffer that generates an exception on write. Reserved memory will be used for the guard region.

`ARCH_KERNEL_STACK_RESERVED` should be defined to the minimum size of a memory protection region. On most ARM CPUs this is 32 bytes. `ARCH_KERNEL_STACK_OBJ_ALIGN` should also be set to the required alignment for this region.

MMU-based systems should not reserve RAM for the guard region and instead simply leave an non-present virtual page below every stack when it is mapped into the address space. The stack object will still need to be properly aligned and sized to page granularity.

```
+----------------------------+ <- thread.stack_obj
| Guard reserved memory      | } K_KERNEL_STACK_RESERVED
+----------------------------+
| Guard carve-out            |
|............................| <- thread.stack_info.start
| Stack buffer               |
.                            .
```

Guard carve-outs for kernel stacks are uncommon and should be avoided if possible. They tend to be needed for two situations:

- The same stack may be re-purposed to host a user thread, in which case the guard is unnecessary and shouldn't be unconditionally reserved. This is the case when privilege elevation stacks are not inside the stack object.
- The required guard size is variable and depends on context. For example, some ARM CPUs have lazy floating point stacking during exceptions and may decrement the stack pointer by a large value without writing anything, completely overshooting a minimally-sized guard and corrupting adjacent memory. Rather than unconditionally reserving a larger guard, the extra memory is carved out if the thread uses floating point.

User mode enabled Enabling user mode activates two new requirements:

- A separate fixed-sized privilege mode stack, specified by `CONFIG_PRIVILEGED_STACK_SIZE`, must be allocated that the user thread cannot access. It is used as the stack by the kernel when handling system calls. If stack guards are implemented, a stack guard region must be able to be placed before it, with support for carve-outs if necessary.
- The memory protection hardware must be able to program a region that exactly covers the thread's stack buffer, tracked in `thread.stack_info`. This implies that `ARCH_THREAD_STACK_SIZE_ADJUST()` will need to round up the requested stack size so that a region may cover it, and that `ARCH_THREAD_STACK_OBJ_ALIGN()` is also specified per the granularity of the memory protection hardware.

This becomes more complicated if the memory protection hardware requires that all memory regions be sized to a power of two, and aligned to their own size. This is common on older MPUs and is known with `CONFIG_MPU_REQUIRES_POWER_OF_TWO_ALIGNMENT`.

`thread.stack_info` always tracks the user-accessible part of the stack object, it must always be correct to program a memory protection region with user access using the range stored within.

Non power-of-two memory region requirements On systems without power-of-two region requirements, the reserved memory area for threads stacks defined by `K_THREAD_STACK_RESERVED` may be used to contain the privilege mode stack. The layout could be something like:

```
+-----------------------------+ <- thread.stack_obj
| Other platform data         |
+-----------------------------+
| Guard region (if enabled)   |
+-----------------------------+
| Guard carve-out (if needed) |
|.............................|
| Privilege elevation stack   |
+-----------------------------| <- thread.stack_obj +
| Stack buffer                |      K_THREAD_STACK_RESERVED =
.                             .      thread.stack_info.start
```

The guard region, and any carve-out (if needed) would be configured as a read-only region when the thread is created.

- If the thread is a supervisor thread, the privilege elevation region is just extra stack memory. An overflow will eventually crash into the guard region.
- If the thread is running in user mode, a memory protection region will be configured to allow user threads access to the stack buffer, but nothing before or after it. An overflow in user mode will crash into the privilege elevation stack, which the user thread has no access to. An overflow when handling a system call will crash into the guard region.

On an MMU system there should be no physical guards; the privilege mode stack will be mapped into kernel memory, and the stack buffer in the user part of memory, each with non-present virtual guard pages below them to catch runtime stack overflows.

Other platform data may be stored before the guard region, but this is highly discouraged if such data could be stored in `thread.arch` somewhere.

`ARCH_THREAD_STACK_RESERVED` will need to be defined to capture the size of the reserved region containing platform data, privilege elevation stacks, and guards. It must be appropriately sized such that an MPU region to grant user mode access to the stack buffer can be placed immediately after it.

Power-of-two memory region requirements Thread stack objects must be sized and aligned to the same power of two, without any reserved memory to allow efficient packing in memory. Thus, any guards in the thread stack must be completely carved out, and the privilege elevation stack must be allocated elsewhere.

`ARCH_THREAD_STACK_SIZE_ADJUST()` and `ARCH_THREAD_STACK_OBJ_ALIGN()` should both be defined to `Z_POW2_CEIL()`. `K_THREAD_STACK_RESERVED` must be 0.

For the privilege stacks, the `CONFIG_GEN_PRIV_STACKS` must be, enabled. For every thread stack found in the system, a corresponding fixed-size kernel stack used for handling system calls is generated. The address of the privilege stacks can be looked up quickly at runtime based on the thread stack address using `z_priv_stack_find()`. These stacks are laid out the same way as other kernel-only stacks.

```
+-----------------------------+ <- z_priv_stack_find(thread.stack_obj)
| Reserved memory             | } K_KERNEL_STACK_RESERVED
+-----------------------------+
| Guard carve-out (if needed) |
|.............................|
| Privilege elevation stack   |
|                             |
+-----------------------------+ <- z_priv_stack_find(thread.stack_obj) +
                                     K_KERNEL_STACK_RESERVED +
                                     CONFIG_PRIVILEGED_STACK_SIZE

+-----------------------------+ <- thread.stack_obj
| MPU guard carve-out         |
| (supervisor mode only)      |
|.............................| <- thread.stack_info.start
| Stack buffer                |
.                             .
```

The guard carve-out in the thread stack object is only used if the thread is running in supervisor mode. If the thread drops to user mode, there is no guard and the entire object is used as the stack buffer, with full access to the associated user mode thread and `thread.stack_info` updated appropriately.

User Mode Threads

To support user mode threads, several kernel-to-arch APIs need to be implemented, and the system must enable the `CONFIG_ARCH_HAS_USERSPACE` option. Please see the documentation for each of these functions for more details:

- `arch_buffer_validate()` to test whether the current thread has access permissions to a particular memory region
- `arch_user_mode_enter()` which will irreversibly drop a supervisor thread to user mode privileges. The stack must be wiped.
- `arch_syscall_oops()` which generates a kernel oops when system call parameters can't be validated, in such a way that the oops appears to be generated from where the system call was invoked in the user thread

- arch_syscall_invoke0() through arch_syscall_invoke6() invoke a system call with the appropriate number of arguments which must all be passed in during the privilege elevation via registers.
- arch_is_user_context() return nonzero if the CPU is currently running in user mode
- arch_mem_domain_max_partitions_get() which indicates the max number of regions for a memory domain. MMU systems have an unlimited amount, MPU systems have constraints on this.

Some architectures may need to update software memory management structures or modify hardware registers on another CPU when memory domain APIs are invoked. If so, CONFIG_ARCH_MEM_DOMAIN_SYNCHRONOUS_API must be selected by the architecture and some additional APIs must be implemented. This is common on MMU systems and uncommon on MPU systems:

- arch_mem_domain_thread_add()
- arch_mem_domain_thread_remove()
- arch_mem_domain_partition_add()
- arch_mem_domain_partition_remove()

Please see the doxygen documentation of these APIs for details.

In addition to implementing these APIs, there are some other tasks as well:

- _new_thread() needs to spawn threads with K_USER in user mode
- On context switch, the outgoing thread's stack memory should be marked inaccessible to user mode by making the appropriate configuration changes in the memory management hardware.. The incoming thread's stack memory should likewise be marked as accessible. This ensures that threads can't mess with other thread stacks.
- On context switch, the system needs to switch between memory domains for the incoming and outgoing threads.
- Thread stack areas must include a kernel stack region. This should be inaccessible to user threads at all times. This stack will be used when system calls are made. This should be fixed size for all threads, and must be large enough to handle any system call.
- A software interrupt or some kind of privilege elevation mechanism needs to be established. This is closely tied to how the _arch_syscall_invoke macros are implemented. On system call, the appropriate handler function needs to be looked up in _k_syscall_table. Bad system call IDs should jump to the K_SYSCALL_BAD handler. Upon completion of the system call, care must be taken not to leak any register state back to user mode.

GDB Stub

To enable GDB stub for remote debugging on a new architecture:

1. Create a new gdbstub.h header file under appropriate architecture include directory (include/zephyr/arch/<arch>/gdbstub.h).
    - Create a new struct struct gdb_ctx as the GDB context.
        - Must define a member named exception of type unsigned int to store the GDB exception reason. This value needs to be set before entering z_gdb_main_loop().
        - Architecture can define as many members as needed for GDB stub to function.
        - Pointer to this struct needs to be passed to z_gdb_main_loop(), where this pointer will be passed to other GDB stub functions.
2. Functions for entering and exiting GDB stub main loop.

- If the architecture relies on interrupts to service breakpoints, interrupt service routines (ISR) need to be implemented, which will serve as the entry point to GDB stub main loop.
- These functions need to save and restore context so code execution can continue as if no breakpoints have been encountered.
- These functions need to call `z_gdb_main_loop()` after saving execution context to go into the GDB stub main loop to receive commands from GDB.
- Before calling `z_gdb_main_loop()`, `gdb_ctx.exception` must be set to specify the exception reason.

3. Implement necessary functions to support GDB stub functionality:
   - `arch_gdb_init()`
     - This needs to initialize necessary bits to support GDB stub functionality, for example, setting up the GDB context and connecting debug interrupts.
     - This must stop code execution via architecture specific method (e.g. raising debug interrupts). This allows GDB to connect during boot.
   - `arch_gdb_continue()`
     - This function is called when GDB sends a `c` or `continue` command to continue code execution.
   - `arch_gdb_step()`
     - This function is called when GDB sends a `si` or `stepi` command to execute one machine instruction, before returning to GDB prompt.
   - Hardware register read/write functions:
     - Since the GDB stub is running on the target, manipulation of hardware registers need to cached to avoid affecting the execution of GDB stub. Think of it as context switching, where the execution context is changed to the GDB stub. So that the register values of the running thread before context switch need to be stored. Manipulation of register values must only be done to this cached copy. The updated values will then be written to hardware registers before switching back to the previous running thread.
     - `arch_gdb_reg_readall()`
       * This collects all hardware register values that would appear in a `g`/`G` packets which will be sent back to GDB. The format of the G-packet is architecture specific. Consult GDB on what is expected.
       * Note that, for most architectures, a valid G-packet must be returned and sent to GDB. If a packet without incorrect length is sent to GDB, GDB will abort the debugging session.
     - `arch_gdb_reg_writeall()`
       * This takes a G-packet sent by GDB and populates the hardware registers with values from the G-packet.
     - `arch_gdb_reg_readone()`
       * This reads the value of one hardware register and sends the result to GDB.
     - `arch_gdb_reg_writeone()`
       * This writes the value of one hardware register received from GDB.
   - Breakpoints:
     - `arch_gdb_add_breakpoint()` and `arch_gdb_remove_breakpoint()`

  - GDB may decide to use software breakpoints which modifies the memory at the breakpoint locations to replace the instruction with software breakpoint or trap instructions. GDB will then restore the memory content once execution reaches the breakpoints. GDB supports this by default and there is usually no need to handle software breakpoints in the architecture code (where breakpoint type is `0`).
  - Hardware breakpoints (type `1`) are required if the code is in ROM or flash that cannot be modified at runtime. Consult the architecture datasheet on how to enable hardware breakpoints.
  - If hardware breakpoints are not supported by the architecture, there is no need to implement these in architecture code. GDB will then rely on software breakpoints.

4. For architecture where certain memory regions are not accessible, an array named `gdb_mem_region_array` of type `gdb_mem_region` needs to be defined to specify regions that are accessible. For each array item:
   - `gdb_mem_region.start` specifies the start of a memory region.
   - `gdb_mem_region.end` specifies the end of a memory region.
   - `gdb_mem_region.attributes` specifies the permission of a memory region.
     - `GDB_MEM_REGION_RO`: region is read-only.
     - `GDB_MEM_REGION_RW`: region is read-write.
   - `gdb_mem_region.alignment` specifies read/write alignment of a memory region. Use `0` if there is no alignment requirement and read/write can be done byte-by-byte.

API Reference

Timing
Architecture timing APIs

Threads
Architecture thread APIs

Architecture-specific Thread Local Storage APIs

Power Management
Architecture-specific power management APIs

Symmetric Multi-Processing
Architecture-specific SMP APIs

Interrupts
Architecture-specific IRQ APIs

Userspace
Architecture-specific userspace APIs

Memory Management
Architecture-specific memory-mapping APIs

Miscellaneous Architecture APIs
Miscellaneous architecture APIs

GDB Stub APIs
Architecture-specific gdbstub APIs

6.9.2 SoC Porting Guide

This page describes how to add support for a new *SoC* in Zephyr, be it in the upstream Zephyr project or locally in your own repository.

SoC Definitions

It is expected that you are familiar with the board concept in Zephyr. A high level overview of the hardware support hierarchy and terms used in the Zephyr documentation can be seen in *Hardware support hierarchy*.

For SoC porting, the most important terms are:

- SoC: the exact system on a chip the board's CPU is part of.
- SoC series: a group of tightly related SoCs.
- SoC family: a wider group of SoCs with similar characteristics.
- CPU cluster: a cluster of one or more CPU cores.
- CPU core: a particular CPU instance of a given architecture.
- Architecture: an instruction set architecture.

Architecture See *Architecture Porting Guide*.

Create your SoC directory

Each SoC must have a unique name. Use the official name given by the SoC vendor and check that it's not already in use. In some cases someone else may have contributed a SoC with identical name. If the SoC name is already in use, then you should probably improve the existing SoC instead of creating a new one. The script `list_hardware` can be used to retrieve a list of all SoCs known in Zephyr, for example `./scripts/list_hardware.py --soc-root=. --socs` from the Zephyr base directory for a list of names that are already in use.

Start by creating the directory `zephyr/soc/<VENDOR>/soc1`, where `<VENDOR>` is your vendor subdirectory.

> **Note**
> A `<VENDOR>` subdirectory is mandatory if contributing your SoC to Zephyr, but if your SoC is placed in a local repo, then any folder structure under `<your-repo>/soc` is permitted. The `<VENDOR>` subdirectory must match a vendor defined in the list in dts/bindings/vendor-prefixes.txt. If the SoC vendor does not have a prefix in that list, then one must be created.

> **Note**
> The SoC directory name does not need to match the name of the SoC. Multiple SoCs can even be defined in one directory. In Zephyr, SoCs are often organized in sub-folders in a common SoC Family or SoC Series tree.

Your SoC directory should look like this:

```
soc/<VENDOR>/<soc-name>
├── soc.yml
├── soc.h
├── CMakeLists.txt
├── Kconfig
├── Kconfig.soc
└── Kconfig.defconfig
```

Replace `<soc-name>` with your SoC's name.

The mandatory files are:

1. `soc.yml`: a YAML file describing the high-level meta data of the SoC such as:
   - SoC name: the name of the SoC
   - CPU clusters: CPU clusters if the SoC contains one or more clusters
   - SoC series: the SoC series to which the SoC belong
   - SoC family: the SoC family to which the series belong
2. `soc.h`: a header file which can be used to describe or provide configuration macros for the SoC. The `soc.h` will often be included in drivers, sub-systems, boards, and other source code found in Zephyr.
3. `Kconfig.soc`: the base SoC configuration which defines a Kconfig SoC symbol in the form of `config SOC_<soc-name>` and provides the SoC name to the Kconfig `SOC` setting. If the `soc.yml` describes a SoC family and series, then those must also be defined in this file. Kconfig settings outside of the SoC tree must not be selected. To select general Zephyr Kconfig settings the `Kconfig` file must be used.
4. `CMakeLists.txt`: CMake file loaded by the Zephyr build system. This CMake file can define additional include paths and/or source files to be used when a build targets the SoC. Also the base line linker script to use must be defined.

The optional files are:

- `Kconfig`, `Kconfig.defconfig` software configuration in *Configuration System (Kconfig)* format. These select the architecture and peripherals available.

Write your SoC YAML

The SoC YAML file describes the SoC family, SoC series, and SoC at a high level.

Detailed configurations, such as hardware description and configuration are done in devicetree and Kconfig.

The skeleton of a simple SoC YAML file containing just one SoC is:

```yaml
socs:
  - name: <soc1>
```

It is possible to have multiple SoC located in the SoC folder. For example if they belong to a common family or series it is recommended to locate such SoC in a common tree. Multiple SoCs and SoC series in a common folder can be described in the `soc.yml` file as:

```yaml
family:
  - name: <family-name>
    series:
      - name: <series-1-name>
        socs:
          - name: <soc1>
            cpuclusters:
```

(continues on next page)

(continued from previous page)

```
            - name: <coreA>
            - name: <coreB>
              ...
        - name: <soc2>
    - name: <series-2-name>
      ...
```

Write your SoC devicetree

SoC devicetree include files are located in the `<zephyr-repo>/dts` folder under a corresponding `<ARCH>/<VENDOR>`.

The SoC `dts/<ARCH>/<VENDOR>/<soc>.dtsi` describes your SoC hardware in the Devicetree Source (DTS) format and must be included by any boards which use the SoC.

If a highlevel `<arch>.dtsi` file exists, then a good starting point is to include this file in your `<soc>.dtsi`.

In general, `<soc>.dtsi` should look like this:

```
#include <arch>/<arch>.dtsi

/ {
        chosen {
                /* common chosen settings for your SoC */
        };

        cpus {
                #address-cells = <m>;
                #size-cells = <n>;

                cpu@0 {
                device_type = "cpu";
                compatible = "<compatibles>";
                /* ... your CPU definitions ... */
        };

        soc {
                /* Your SoC definitions and peripherals */
                /* such as ram, clock, buses, peripherals. */
        };
};
```

Hint

It is possible to structure multiple `<VENDOR>/<soc>.dtsi` files in sub-directories for a cleaner file system structure. For example organized pre SoC series, like this: `<VENDOR>/<SERIES>/<soc>.dtsi`.

Multiple CPU clusters Devicetree reflects the hardware. The memory space and peripherals available to one CPU cluster can be very different from another CPU cluster, therefore each CPU cluster will often have its own `.dtsi` file.

CPU cluster `.dtsi` files should follow the naming scheme `<soc>_<cluster>.dtsi`. A `<soc>_<cluster>.dtsi` file will look similar to a SoC `.dtsi` without CPU clusters.

Write Kconfig files

Zephyr uses the Kconfig language to configure software features. Your SoC needs to provide some Kconfig settings before you can compile a Zephyr application for it.

Setting Kconfig configuration values is documented in detail in *Setting Kconfig configuration values*.

There is one mandatory Kconfig file in the SoC directory, and two optional files for a SoC:

```
soc/<vendor>/<your soc>
├── Kconfig.soc
├── Kconfig
└── Kconfig.defconfig
```

`Kconfig.soc`
: A shared Kconfig file which can be sourced both in Zephyr Kconfig and sysbuild Kconfig trees.

  This file selects the SoC family and series in the Kconfig tree and potential other SoC related Kconfig settings. In some cases a SOC_PART_NUMBER. This file must not select anything outside the reusable Kconfig SoC tree.

  A `Kconfig.soc` may look like this:

```
config SOC_FAMILY_<SOC_FAMILY_NAME>
        bool

config SOC_SERIES_<SOC_SERIES_NAME>
        bool
        select SOC_FAMILY_<SOC_FAMILY_NAME>

config SOC_<SOC_NAME>
        bool
        select SOC_SERIES_<SOC_SERIES_NAME>

config SOC_FAMILY
        default "<soc_family_name>" if SOC_FAMILY_<SOC_FAMILY_NAME>

config SOC_SERIES
        default "<soc_series_name>" if SOC_SERIES_<SOC_SERIES_NAME>

config SOC
        default "<soc_name>" if SOC_<SOC_NAME>
```

  Notice that `SOC_NAME` is a pure uppercase version of the SoC name, `SOC_SERIES_NAME` is a pure uppercase version of SoC series name and `SOC_FAMILY_NAME` is a pure uppercase version of the SoC family name. If these fields do not appear in the `soc.yml` file then they should not be present in the `Kconfig.soc` file.

  The Kconfigs `SOC`, `SOC_SERIES` and `SOC_FAMILY` settings are globally defined as strings and therefore the `Kconfig.soc` file shall only define the default string values and not the types. Notice that the string values must match the values used in the `soc.yml` file.

Note

The build system supports any variation of case for `soc_name`, `soc_series_name` and `soc_family_mame`, but when submitting boards for inclusion in Zephyr itself, these must be purely lowercase versions of the Kconfig names

`Kconfig`
: Included by soc/Kconfig.

This file can add Kconfig settings which are specific to the current SoC.

The `Kconfig` will often indicate given hardware support using a setting of the form `HAS_<support>`.

```
config SOC_<SOC_NAME>
        select ARM
        select CPU_HAS_FPU
```

If the setting name is identical to an existing Kconfig setting in Zephyr and only modifies the default value of said setting, then `Kconfig.defconfig` should be used instead.

`Kconfig.defconfig`
: SoC specific default values for Kconfig options.

Not all SoCs have a `Kconfig.defconfig` file.

The entire file should be inside a pair of `if SOC_<SOC_NAME>` / `endif` or `if SOC_SERIES_<SERIES_NAME>` / `endif`, like this:

```
if SOC_<SOC_NAME>

config NUM_IRQS
        default 32

endif # SOC_<SOC_NAME>
```

Multiple CPU clusters CPU clusters must provide additional Kconfig settings in the `Kconfig.soc` file. This will usually be in the form of `SOC_<SOC_NAME>_<CLUSTER>` so for a given `soc1` with two clusters `clusterA` and `clusterB`, then this will look like:

SoC's When a SoC defines CPU cluster

```
config SOC_SOC1_CLUSTERA
        bool
        select SOC_SOC1

config SOC_SOC1_CLUSTERB
        bool
        select SOC_SOC1
```

6.9.3 Board Porting Guide

To add Zephyr support for a new *board*, you at least need a *board directory* with various files in it. Files in the board directory inherit support for at least one SoC and all of its features. Therefore, Zephyr must support your *SoC* as well.

Transition to the current hardware model

Shortly after Zephyr 3.6.0 was released, a new hardware model was introduced to Zephyr. This new model overhauls the way both SoCs and boards are named and defined, and adds support for features that had been identified as important over the years. Among them:

- Support for multi-core, multi-arch AMP (Asymmetrical Multi Processing) SoCs
- Support for multi-SoC boards
- Support for reusing the SoC and board Kconfig trees outside of the Zephyr build system

- Support for advanced use cases with *Sysbuild (System build)*
- Removal of all existing arbitrary and inconsistent uses of Kconfig and folder names

All the documentation in this page refers to the current hardware model. Please refer to the documentation in Zephyr v3.6.0 (or earlier) for information on the previous, now obsolete, hardware model.

More information about the rationale, development and concepts behind the new model can be found in the original issue, the original Pull Request and, for a complete set of changes introduced, the hardware model v2 commit.

Some non-critical features, enhancements and improvements of the new hardware model are still in development. Check the hardware model v2 enhancements issue for a complete list.

The transition from the previous hardware model to the current one (commonly referred to as "hardware model v2") requires modifications to all existing board and SoC definitions. A decision was made not to provide direct backwards compatibility for the previous model, which leaves users transitioning from a previous version of Zephyr to one including the new model (v3.7.0 and onwards) with two options if they have an out-of-tree board (or SoC):

1. Convert the out-of-tree board to the current hardware model (recommended)
2. Take the SoC definition from Zephyr v3.6.0 and copy it to your downstream repository (ensuring that the build system can find it via a *zephyr module* or `SOC_ROOT`). This will allow your board, defined in the previous hardware model, to continue to work

When converting your board from the previous to the current hardware model, we recommend first reading through this page to understand the model in detail. You can then use the example-application conversion Pull Request as an example on how to port a simple board. Additionally, a conversion script is available and works reliably in many cases (though multi-core SoCs may not be handled entirely). Finally, the hardware model v2 commit contains the full conversion of all existing boards from the old to the current model, so you can use it as a complete conversion reference.

Hardware support hierarchy

Zephyr's hardware support is based on a series of hierarchical abstractions. Primarily, each *board* has one or more *SoC*. Each SoC can be optionally classed into an *SoC series*, which in turn may optionally belong to an *SoC family*. Each SoC has one or more *CPU cluster*, each containing one or more *CPU core* of a particular *architecture*.

You can visualize the hierarchy in the diagram below:

Below are some examples of the hierarchy described in this section, in the form of a *board* per row with its corresponding hierarchy entries. Notice how the *SoC series* and *SoC family* levels are not always used.

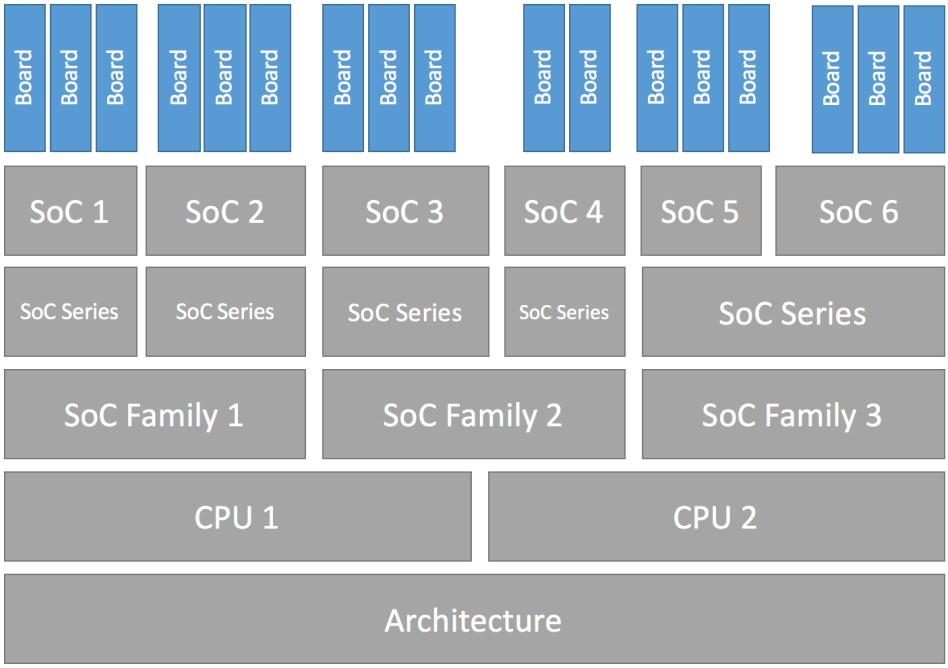


Fig. 5: Hardware support Hierarchy

board name	*board qualifiers*	*SoC*	*SoC Series*	*SoC family*	CPU core	*architecture*
nrf52dk	nrf52832	nRF52832	nRF52	Nordic nRF	Arm Cortex-M4	ARMv7-M
frdm_k64f	mk64f12	MK64F12	Kinetis K6x	NXP Kinetis	Arm Cortex-M4	ARMv7-M
rv32m1_vega	openisa_rv32m1/r	RV32M1	(Not used)	(Not used)	RI5CY	RISC-V RV32
nrf5340dk	nrf5340/cpuapp	nRF5340	nRF53	Nordic nRF	Arm Cortex-M33	ARMv8-M
	nrf5340/cpunet	nRF5340	nRF53	Nordic nRF	Arm Cortex-M33	ARMv8-M
mimx8mp_e	mimx8ml8/a53	i.MX8M Plus	i.MX8M	NXP i.MX	Arm Cortex-A53	ARMv8-A
	mimx8ml8/m7	i.MX8M Plus	i.MX8M	NXP i.MX	Arm Cortex-M7	ARMv7-M
	mimx8ml8/adsp	i.MX8M Plus	i.MX8M	NXP i.MX	Cadence HIFI4	Xtensa LX6

Additional details about terminology can be found in the next section.

Board terminology

The previous section introduced the hierarchical manner in which Zephyr classifies and implements hardware support. This section focuses on the terminology used around hardware support, and in particular when defining and working with boards and SoCs.

The overall set of terms used around the concept of board in Zephyr is depicted in the image below, which uses the bl5340_dvk board as reference.

The diagram shows the different terms that are used to describe boards:

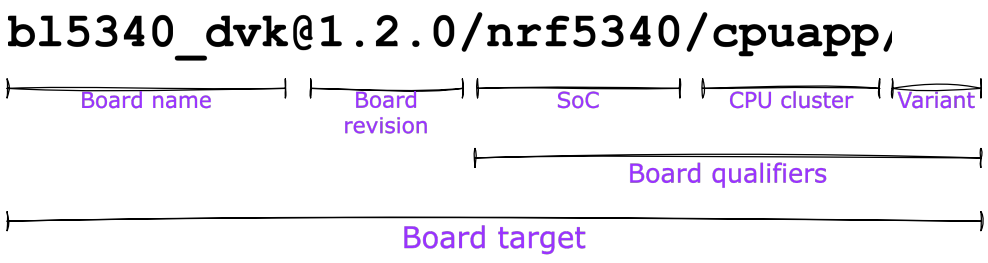


Fig. 6: Board terminology diagram

- The *board name*: `bl5340_dvk`
- The optional *board revision*: `1.2.0`
- The *board qualifiers*, that optionally describe the *SoC*, *CPU cluster* and *variant*: `nrf5340/cpuapp/ns`
- The *board target*, which uniquely identifies a combination of the above and can be used to specify the hardware to build for when using the tooling provided by Zephyr: `bl5340_dvk@1.2.0/nrf5340/cpuapp/ns`

Formally this can also be seen as *board name*[@*revision*][/*board qualifiers*], which can be extended to *board name*[@*revision*][/*SoC*[/*CPU cluster*][/*variant*]].

If a board contains only one single-core SoC, then the SoC can be omitted from the board target. This implies that if the board does not define any board qualifiers, the board name can be used as a board target. Conversely, if board qualifiers are part of the board definition, then the SoC can be omitted by leaving it out but including the corresponding forward-slashes: `//`.

Continuing with the example above, The board bl5340_dvk is a single SoC board where the SoC defines two CPU clusters: `cpuapp` and `cpunet`. One of the CPU clusters, `cpuapp`, additionally defines a non-secure board variant, `ns`.

The board qualifiers `nrf5340/cpuapp/ns` can be read as:

- `nrf5340`: The SoC, which is a Nordic nRF5340 dual-core SoC
- `cpuapp`: The CPU cluster `cpuapp`, which consists of a single Cortex-M33 CPU core. The number of cores in a CPU cluster cannot be determined from the board qualifiers.
- `ns`: a variant, in this case `ns` is a common variant name in Zephyr denoting a non-secure build for boards supporting *Trusted Firmware-M (TF-M)*.

Not all SoCs define CPU clusters or variants. For example a simple board like the thingy52 contains a single SoC with no CPU clusters and no variants. For `thingy52` the board target `thingy52/nrf52832` can be read as:

- `thingy52`: board name.
- `nrf52832`: The board qualifiers, in this case identical to the SoC, which is a Nordic nRF52832.

Make sure your SoC is supported

Start by making sure your SoC is supported by Zephyr. If it is, it's time to *Create your board directory*. If you don't know, try:

- checking boards for names that look relevant, and reading individual board documentation to find out for sure.
- asking your SoC vendor

If you need to add a SoC, CPU cluster, or even architecture support, this is the wrong page, but here is some general advice.

Architecture See *Architecture Porting Guide*.

CPU Core CPU core support files go in `core` subdirectories under arch, e.g. arch/x86/core.

See *Install a Toolchain* for information about toolchains (compiler, linker, etc.) supported by Zephyr. If you need to support a new toolchain, *Build and Configuration Systems* is a good place to start learning about the build system. Please reach out to the community if you are looking for advice or want to collaborate on toolchain support.

SoC Zephyr SoC support files are in architecture-specific subdirectories of soc. They are generally grouped by SoC family.

When adding a new SoC family or series for a vendor that already has SoC support within Zephyr, please try to extract common functionality into shared files to avoid duplication. If there is no support for your vendor yet, you can add it in a new directory `zephyr/soc/<VENDOR>/<YOUR-SOC>`; please use self-explanatory directory names.

Create your board directory

Once you've found an existing board that uses your SoC, you can usually start by copy/pasting its board directory and changing its contents for your hardware.

You need to give your board a unique name. Run `west boards` for a list of names that are already taken, and pick something new. Let's say your board is called `plank` (please don't actually use that name).

Start by creating the board directory `zephyr/boards/<VENDOR>/plank`, where `<VENDOR>` is your vendor subdirectory. (You don't have to put your board directory in the zephyr repository, but it's the easiest way to get started. See *Custom Board, Devicetree and SOC Definitions* for documentation on moving your board directory to a separate repository once it's working.)

> **Note**
>
> A `<VENDOR>` subdirectory is mandatory if contributing your board to Zephyr, but if your board is placed in a local repo, then any folder structure under `<your-repo>/boards` is permitted. If the vendor is defined in the list in dts/bindings/vendor-prefixes.txt then you must use that vendor prefix as `<VENDOR>`. `others` may be used as vendor prefix if the vendor is not defined.

> **Note**
>
> The board directory name does not need to match the name of the board. Multiple boards can even be defined in one directory.

Your board directory should look like this:

```
boards/<VENDOR>/plank
├── board.yml
├── board.cmake
├── CMakeLists.txt
├── doc
│   ├── plank.webp
│   └── index.rst
├── Kconfig.plank
├── Kconfig.defconfig
├── plank_<qualifiers>_defconfig
```

(continues on next page)

(continued from previous page)

```
├── plank_<qualifiers>.dts
└── plank_<qualifiers>.yaml
```

Replace `plank` with your board's name, of course.

The mandatory files are:

1. `board.yml`: a YAML file describing the high-level meta data of the boards such as the boards names, their SoCs, and variants. CPU clusters for multi-core SoCs are not described in this file as they are inherited from the SoC's YAML description.
2. `plank_<qualifiers>.dts`: a hardware description in *devicetree* format. This declares your SoC, connectors, and any other hardware components such as LEDs, buttons, sensors, or communication peripherals (USB, Bluetooth controller, etc).
3. `Kconfig.plank`: the base software configuration for selecting SoC and other board and SoC related settings. Kconfig settings outside of the board and SoC tree must not be selected. To select general Zephyr Kconfig settings the `Kconfig` file must be used.

The optional files are:

- `Kconfig`, `Kconfig.defconfig` software configuration in *Configuration System (Kconfig)* formats. This provides default settings for software features and peripheral drivers.
- `plank_defconfig` and `plank_<qualifiers>_defconfig`: software configuration in Kconfig `.conf` format.
- `board.cmake`: used for *Flash and debug support*
- `CMakeLists.txt`: if you need to add additional source files to your build.
- `doc/index.rst`, `doc/plank.webp`: documentation for and a picture of your board. You only need this if you're *Contributing your board* to Zephyr.
- `plank_<qualifiers>.yaml`: a YAML file with miscellaneous metadata used by the *Test Runner (Twister)*.

Board qualifiers of the form `<soc>/<cpucluster>/<variant>` are normalized so that / is replaced with _ when used for filenames, for example: `soc1/foo` becomes `soc1_foo` when used in filenames.

Write your board YAML

The board YAML file describes the board at a high level. This includes the SoC, board variants, and board revisions.

Detailed configurations, such as hardware description and configuration are done in devicetree and Kconfig.

The skeleton of the board YAML file is:

```yaml
board:
  name: <board-name>
  full_name: <board-full-name>
  vendor: <board-vendor>
  revision:
    format: <major.minor.patch|letter|number|custom>
    default: <default-revision-value>
    exact: <true|false>
    revisions:
    - name: <revA>
    - name: <revB>
      ...
```

(continues on next page)

(continued from previous page)

```
  socs:
  - name: <soc-1>
    variants:
    - name: <variant-1>
    - name: <variant-2>
      variants:
      - name: <sub-variant-2-1>
        ...
  - name: <soc-2>
    ...
```

It is possible to have multiple boards located in the board folder. If multiple boards are placed in the same board folder, then the file `board.yml` must describe those in a list as:

```
boards:
- name: <board-name-1>
  vendor: <board-vendor>
  full_name: <board-full-name>
  ...
- name: <board-name-2>
  vendor: <board-vendor>
  full_name: <board-full-name>
  ...
...
```

Write your devicetree

The devicetree file `boards/<vendor>/plank/plank_<qualifiers>.dts` describes your board hardware in the Devicetree Source (DTS) format (as usual, change `plank` to your board's name). If you're new to devicetree, see *Introduction to devicetree*.

In general, `plank_<qualifiers>.dts` should look like this:

```
/dts-v1/;
#include <your_soc_vendor/your_soc.dtsi>

/ {
        model = "A human readable name";
        compatible = "yourcompany,plank";

        chosen {
                zephyr,console = &your_uart_console;
                zephyr,sram = &your_memory_node;
                /* other chosen settings  for your hardware */
        };

        /*
         * Your board-specific hardware: buttons, LEDs, sensors, etc.
         */

        leds {
                compatible = "gpio-leds";
                led0: led_0 {
                        gpios = </* GPIO your LED is hooked up to */>;
                        label = "LED 0";
                };
```

(continues on next page)

(continued from previous page)

```
                /* ... other LEDs ... */
        };

        buttons {
                compatible = "gpio-keys";
                /* ... your button definitions ... */
        };

        /* These aliases are provided for compatibility with samples */
        aliases {
                led0 = &led0; /* now you support the blinky sample! */
                /* other aliases go here */
        };
};

&some_peripheral_you_want_to_enable { /* like a GPIO or SPI controller */
        status = "okay";
};

&another_peripheral_you_want {
        status = "okay";
};
```

In the case a board has only a single SoC, without any board variants then the dts file can be named `<plank>.dts` instead, however this is not recommended due to the file silently be unused if a variant or other SoC is added to the board.

If you're in a hurry, simple hardware can usually be supported by copy/paste followed by trial and error. If you want to understand details, you will need to read the rest of the devicetree documentation and the devicetree specification.

Example: FRDM-K64F and Hexiwear K64 This section contains concrete examples related to writing your board's devicetree.

The FRDM-K64F and Hexiwear K64 board devicetrees are defined in frdm_k64fs.dts and hexiwear_k64.dts respectively. Both boards have NXP SoCs from the same Kinetis SoC family, the K6X.

Common devicetree definitions for K6X are stored in nxp_k6x.dtsi, which is included by both board `.dts` files. nxp_k6x.dtsi in turn includes armv7-m.dtsi, which has common definitions for Arm v7-M cores.

Since nxp_k6x.dtsi is meant to be generic across K6X-based boards, it leaves many devices disabled by default using `status` properties. For example, there is a CAN controller defined as follows (with unimportant parts skipped):

```
can0: can@40024000 {
     ...
     status = "disabled";
     ...
};
```

It is up to the board `.dts` or application overlay files to enable these devices as desired, by setting `status = "okay"`. The board `.dts` files are also responsible for any board-specific configuration of the device, such as adding nodes for on-board sensors, LEDs, buttons, etc.

For example, FRDM-K64 (but not Hexiwear K64) `.dts` enables the CAN controller and sets the bus speed:

```
&can0 {
     status = "okay";
};
```

The `&can0 { ... };` syntax adds/overrides properties on the node with label `can0`, i.e. the `can@4002400` node defined in the `.dtsi` file.

Other examples of board-specific customization is pointing properties in `aliases` and `chosen` to the right nodes (see *Aliases and chosen nodes*), and making GPIO/pinmux assignments.

Write Kconfig files

Zephyr uses the Kconfig language to configure software features. Your board needs to provide some Kconfig settings before you can compile a Zephyr application for it.

Setting Kconfig configuration values is documented in detail in *Setting Kconfig configuration values*.

There is one mandatory Kconfig file in the board directory, and several optional files for a board named `plank`:

```
boards/<vendor>/plank
├── Kconfig
├── Kconfig.plank
├── Kconfig.defconfig
└── plank_<qualifiers>_defconfig
```

`Kconfig.plank`
: A shared Kconfig file which can be sourced both in Zephyr Kconfig and sysbuild Kconfig trees.

  This file selects the SoC in the Kconfig tree and potential other SoC related Kconfig settings. This file must not select anything outside the reusable Kconfig board and SoC trees.

  A `Kconfig.plank` may look like this:

  ```
  config BOARD_PLANK
          select SOC_SOC1
  ```

  The Kconfig symbols `BOARD_`*board* and `BOARD_`*normalized_board_target* are constructed by the build system, therefore no type shall be defined in above code snippet.

`Kconfig`
: Included by boards/Kconfig.

  This file can add Kconfig settings which are specific to the current board.

  Not all boards have a `Kconfig` file.

  A board specific setting should be defining a custom setting and usually with a prompt, like this:

  ```
  config BOARD_FEATURE
          bool "Board specific feature"
  ```

  If the setting name is identical to an existing Kconfig setting in Zephyr and only modifies the default value of said setting, then `Kconfig.defconfig` should be used instead.

`Kconfig.defconfig`
: Board-specific default values for Kconfig options.

  Not all boards have a `Kconfig.defconfig` file.

  The entire file should be inside an `if BOARD_PLANK` / `endif` pair of lines, like this:

```
if BOARD_PLANK

config FOO
        default y

if NETWORKING

config SOC_ETHERNET_DRIVER
        default y

endif # NETWORKING

endif # BOARD_PLANK
```

plank_<qualifiers>_defconfig (or plank_defconfig in limited circumstances)
A Kconfig fragment that is merged as-is into the final build directory `.config` whenever an application is compiled for your board.

`plank_defconfig` can only be used with boards that have no qualifiers, no variants and a single SoC present, though this style of naming is not recommended due to samples/tests or downstream usage breaking suddenly without warning if a new SoC or board variant/qualifier is added to an board in upstream Zephyr.

Note

Multiple files are not merged and there is no fallback mechanism for files, this means if there is a board with 2 different SoCs and each one has 2 board variants, a `plank_defconfig` file would be wholly unused, for the first qualifier and variant `plank_<soc1>_<variant1>_defconfig` will be used, it will not include other file.

The `_defconfig` should contain mandatory settings for your UART, console, etc. The results are architecture-specific, but typically look something like this:

```
CONFIG_GPIO=y
CONFIG_CONSOLE=y
CONFIG_UART_CONSOLE=y
CONFIG_SERIAL=y
```

plank_x_y_z_defconfig / plank_<qualifiers>_x_y_z_defconfig
A Kconfig fragment that is merged as-is into the final build directory `.config` whenever an application is compiled for your board revision `x.y.z`.

Build, test, and fix

Now it's time to build and test the application(s) you want to run on your board until you're satisfied.

For example:

```
west build -b plank samples/hello_world
west flash
```

For `west flash` to work, see *Flash and debug support* below. You can also just flash `build/zephyr/zephyr.elf`, `zephyr.hex`, or `zephyr.bin` with any other tools you prefer.

Before submitting a board upstream, verify that every board target you add can pass the project's minimum open source test suite using only code from the mainline Zephyr repository and its modules. The suite currently consists of:

- `samples/philosophers`

- `tests/kernel`

For example, build the suite for a board target with:

```
west twister -p plank -T samples/philosophers -T tests/kernel
```

For boards with multiple SoCs, CPU clusters, variants, or revisions, repeat the test suite for each new board target. A hello_world build is also recommended as a quick smoke check, for example:

```
west build -p always -b plank/soc1/foo samples/hello_world
west build -p always -b plank@1.0.0/soc1/foo samples/hello_world
```

Use *Sysbuild (System build)* if the board target requires it. When using board testing metadata, such as `testing: only_tags` in the board target YAML file, make sure the target is still validated against the minimum test suite in local testing or CI.

General recommendations

For consistency and to make it easier for users to build applications which remain board agnostic, please follow these guidelines when porting a board you intend to contribute to Zephyr:

Enable valuable components in Devicetree
Devicetree nodes for valuable onboard components (LEDs, buttons, sensors, onboard USB/Ethernet/BLE/Wi-Fi, etc.) must be **enabled by default** and have correct pin control and driver configuration so that they work out of the box.

Keep subsystems disabled by default (Kconfig)
Do not enable subsystems in the board defconfig unless they are strictly required for basic board operation, or are explicitly listed as exceptions in these recommendations.

Configure system clock and tick source
Set up a functioning system clock and tick source.

Provide a default console
Use the `zephyr,console` chosen node to point to the UART controller used for console output.

Boards with built-in debug or a USB-to-UART adapter should set the console to the UART controller connected to that adapter.

USB-only boards without any debug adapter must include the common USB CDC-ACM Kconfig and DTS fragments to enable CDC-ACM UART as a default backend for logging and shell.

Add *shield interface* definitions
For boards exposing standard expansion headers, add connector nodes and pin-muxing. Enable only the peripherals needed for the expected/standard connector functionality.

Configure pins and peripheral instances
Map peripherals to the correct pins (e.g., SPI on Arduino SPI pins) and provide default pinmux entries supporting the board's features.

Enable networking interfaces
If networking hardware is present, configure default interfaces for each supported technology so that networking samples work out of the box.

Enable GPIO controllers
All GPIO ports connected to onboard components or expansion headers should be enabled.

Enable MPU and stack protection
It is recommended to enable the MPU when available (unless memory resources are too limited). When the MPU is enabled, it is recommended to also enable hardware stack protection (`CONFIG_HW_STACK_PROTECTION`) to ease debugging by allowing the kernel to detect stack overflows.

Flash and debug support

Zephyr supports *Building, Flashing and Debugging* via west extension commands.

To add `west flash` and `west debug` support for your board, you need to create a `board.cmake` file in your board directory. This file's job is to configure a "runner" for your board. (There's nothing special you need to do to get `west build` support for your board.)

"Runners" are Zephyr-specific Python classes that wrap *flash and debug host tools* and integrate with west and the zephyr build system to support `west flash` and related commands. Each runner supports flashing, debugging, or both. You need to configure the arguments to these Python scripts in your `board.cmake` to support those commands like this example `board.cmake`:

```
board_runner_args(jlink "--device=nrf52" "--speed=4000")
board_runner_args(pyocd "--target=nrf52" "--frequency=4000000")

include(${ZEPHYR_BASE}/boards/common/nrfutil.board.cmake)
include(${ZEPHYR_BASE}/boards/common/nrfjprog.board.cmake)
include(${ZEPHYR_BASE}/boards/common/jlink.board.cmake)
include(${ZEPHYR_BASE}/boards/common/pyocd.board.cmake)
```

This example configures the `nrfutil`, `nrfjprog`, `jlink`, and `pyocd` runners.

Warning

Runners usually have names which match the tools they wrap, so the `jlink` runner wraps Segger's J-Link tools, and so on. But the runner command line options like `--speed` etc. are specific to the Python scripts.

Note

Runners and board configuration should be created without being targeted to a single operating system if the tool supports multiple operating systems, nor should it rely upon special system setup/configuration. For example; do not assume that a user will have prior knowledge/configuration or (if using Linux) special udev rules installed, do not assume one specific `/dev/X` device for all platforms as this will not be compatible with Windows or macOS, and allow for overriding of the selected device so that multiple boards can be connected to a single system and flashed/debugged at the choice of the user.

For more details:

- Run `west flash --context` to see a list of available runners which support flashing, and `west flash --context -r <RUNNER>` to view the specific options available for an individual runner.
- Run `west debug --context` and `west debug --context <RUNNER>` to get the same output for runners which support debugging.
- Run `west flash --help` and `west debug --help` for top-level options for flashing and debugging.
- See *Configuration Options* for Python APIs.
- Look for `board.cmake` files for other boards similar to your own for more examples.

To see what a `west flash` or `west debug` command is doing exactly, run it in verbose mode:

```
west --verbose flash
west --verbose debug
```

Verbose mode prints any host tool commands the runner uses.

The order of the `include()` calls in your `board.cmake` matters. The first `include` sets the default runner if it's not already set. For example, including `nrfjprog.board.cmake` first means that `nrfjprog` is the default flash runner for this board. Since `nrfjprog` does not support debugging, `jlink` is the default debug runner.

Multiple board revisions

See *Building for a board revision* for basics on this feature from the user perspective.

Board revisions are described in the `revision` entry of the `board.yml`.

```
board:
  revision:
    format: <major.minor.patch|letter|number|custom>
    default: <default-revision-value>
    exact: <true|false>
    revisions:
    - name: <revA>
    - name: <revB>
```

Zephyr natively supports the following revision formats:

- `major.minor.patch`: match a three digit revision, such as `1.2.3`.
- `number`: matches integer revisions
- `letter`: matches single letter revisions from `A` to `Z` only

Fuzzy revision matching Fuzzy revision matching is enabled per default.

If the user selects a revision between those available, the closest revision number that is not larger than the user's choice is used. For example, if the board `plank` defines revisions `0.5.0`, and `1.5.0` and the user builds for `plank@0.7.0`, the build system will target revision `0.5.0`.

The build system will print this at CMake configuration time:

```
-- Board: plank, Revision: 0.7.0 (Active: 0.5.0)
```

This allows you to only create revision configuration files for board revision numbers that introduce incompatible changes.

Similarly for `letter` revision format, if revisions `A`, `D`, and `F` are defined and the user builds for `plank@E`, the build system will target revision `D`.

Exact revision matching Exact revision matching is enabled when `exact: true` is specified in the revision section in `board.yml`.

When exact is defined then building for `plank@0.7.0` in the above example will result in the following error message:

```
Board revision `0.7.0` not found.  Please specify a valid board revision.
```

Board revision configuration adjustment When the user builds for board `plank@<revision>` it is possible to make adjustments to the board's normal configuration.

As described in the *Write your devicetree* and *Write Kconfig files* sections the board default configuration is created from the files `<board>.dts` / `<board>_<qualifiers>.dts` and `<board>_defconfig` / `<board>_<qualifiers>_defconfig`. When building for a specific board revision, the above files are used as a starting point and the following board files will be used in addition:

- <board>_<qualifiers>_<revision>_defconfig: a specific revision defconfig which is only used for the board and SOC / variants identified by <board>_<qualifiers>.
- <board>_<qualifiers>_<revision>.overlay: a specific revision dts overlay which is only used for the board and SOC / variants identified by <board>_<qualifiers>.

This split allows boards with multiple SoCs, multi-core SoCs, or variants to place common revision adjustments which apply to all SoCs and variants in a single file, while still providing the ability to place SoC or variant specific adjustments in a dedicated revision file.

Using the plank board from previous sections, then we could have the following revision adjustments:

```
boards/zephyr/plank
├── plank_soc1_foo_1_5_0.overlay   # DTS overlay for plank board when building for␣
↪soc1 variant foo on revision 1.5.0
└── plank_soc1_foo_1_5_0_defconfig # Kconfig adjustment for plank board when␣
↪building for soc1 variant foo on revision 1.5.0
```

Custom revision.cmake files

Some boards may not use board revisions supported natively by Zephyr. For example string revisions.

One reason why Zephyr doesn't support string revisions is that strings can take many forms and it's not always clear if the given strings are just strings, such as blue, green, red, etc. or if they provide an order which can be matched against higher or lower revisions, such as alpha, beta`, gamma.

Due to the sheer number of possibilities with strings, including the possibility of doing regex matches internally, then string revisions must be done using custom revision type.

To indicate to the build system that custom revisions are used, the format field in the revision section of the board.yml must be written as:

```yaml
board:
  revision:
    format: custom
```

When using custom revisions then a revision.cmake must be created in the board directory.

The revision.cmake will be included by the build system when building for the board and it is the responsibility of the file to validate the revision specified by the user.

The **BOARD_REVISION** variable holds the revision value specified by the user.

To signal to the build system that it should use a different revision than the one specified by the user, revision.cmake can set the CMake variable *ACTIVE_BOARD_REVISION* to the revision to use instead. The corresponding Kconfig files and devicetree overlays must be named <board>_<ACTIVE_BOARD_REVISION>_defconfig and <board>_<ACTIVE_BOARD_REVISION>.overlay.

Contributing your board

If you want to contribute your board to Zephyr, first – thanks!

There are some extra things you'll need to do:

1. Make sure you've followed all the *General recommendations*. They are requirements for boards included with Zephyr.
2. Add documentation for your board using the template file doc/templates/board.tmpl. See *Documentation Generation* for information on how to build your documentation before submitting your pull request.

3. Prepare a pull request adding your board which follows the *Contribution Guidelines*.

Board extensions

The board hardware model in Zephyr allows you to extend an existing board with new board variants. Such board extensions can be done in your custom repository and thus outside of the Zephyr repository.

Extending an existing board with an extra variant allows you to adjust an existing board and thereby during build to select building for the existing, unmodified board, or the new variant.

To extend an existing board, first create a `board.yml` in your extended board. Make sure to use the directory structure described in *Create your board directory*.

The skeleton of the board YAML file for extending a board is:

```yaml
board:
  extend: <existing-board-name>
  variants:
    - name: <new-variant>
      qualifier: <existing-qualifier>
```

When extending a board, your board directory should look like:

```
boards/<VENDOR>/plank
├── board.yml
├── plank_<new-qualifiers>_defconfig
└── plank_<new-qualifiers>.dts
```

Replace `plank` with the real name of the board you extend.

In some cases you might want to also adjust additional settings, like the `Kconfig.defconfig` or `Kconfig.`*board*. Therefore it is also possible to provide the following in addition when extending a board.

```
boards/<VENDOR>/plank
├── board.cmake
├── Kconfig
├── Kconfig.plank
├── Kconfig.defconfig
└── plank_<new-qualifiers>.yaml
```

6.9.4 Shields

Shields, also known as "add-on" or "daughter boards", attach to a board to extend its features and services for easier and modularized prototyping. In Zephyr, the shield feature provides Zephyr-formatted shield descriptions for easier compatibility with applications.

Shield activation

Activate support for one or more shields by adding the matching `--shield` arguments to the west command:

```
west build -b your_board --shield x_nucleo_idb05a1 --shield␣
↪x_nucleo_iks01a1 your_app
```

Alternatively, it could be set by default in a project's CMakeLists.txt:

```
set(SHIELD x_nucleo_iks01a1)
```

Shield interfaces

A shield is defined by two key characteristics:

1. **Physical connectors** - the mechanical interface
2. **Electrical signals** - what each pin actually does

The connection between a shield and a board happens through Devicetree files:

- Board side: The board's devicetree file uses a *GPIO nexus node* to map connector pins to the microcontroller's actual GPIO pins. It also defines labels for buses exposed through the connector (like `arduino_i2c`, `arduino_spi`, `arduino_uart`).
- Shield side: The shield's .overlay file references these same labels to describe how its components connect to the board.

At build time, the board's devicetree and the shield's overlay are combined to create a complete picture of the hardware setup.

For example, let's say you have a board with an Arduino connector but no built-in accelerometer. You can add one using an Arduino shield:

1. The board's Devicetree defines an `arduino_i2c` label: it is the I2C bus made available on the Arduino connector
2. The accelerometer shield's overlay file also references `arduino_i2c` to indicate it uses that same I2C bus. If it needs to use GPIO pins from the connector, it references the GPIO nexus node defined by the board's Devicetree (e.g. `arduino_header`).

When you then build for this board with this shield, Zephyr automatically "wires them" together.

> **Note**
>
> Some boards and shields may only support a limited set of features of a shield hardware interface. Refer to their documentation for more details.

Arduino MKR This is the form factor of the Arduino MKR boards.

Relevant devicetree node labels:

- `arduino_mkr_header` See `arduino-mkr-header` for details on GPIO pin definitions and includes for use in devicetree files.
- `arduino_mkr_i2c`
- `arduino_mkr_spi`
- `arduino_mkr_serial`

Arduino Nano This is the form factor of the Arduino Nano boards.

Relevant devicetree node labels:

- `arduino_nano_header` See `arduino-nano-header` for details on GPIO pin definitions and includes for use in devicetree files.
- `arduino_nano_i2c`
- `arduino_nano_spi`
- `arduino_nano_serial`

Fig. 7: Arduino MKR Zero, an example of a board with the Arduino MKR shield interface

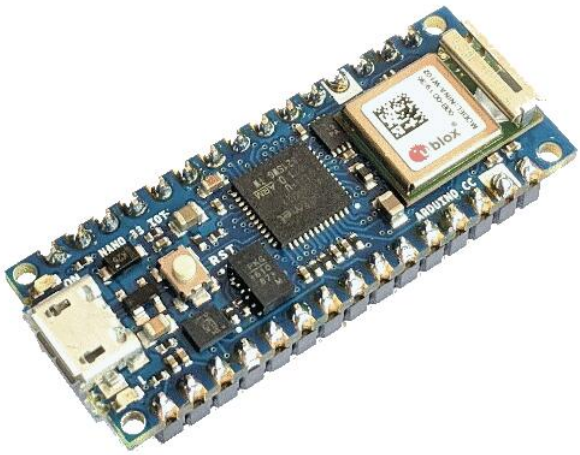


Fig. 8: Arduino Nano 33 IOT, an example of a board with the Arduino Nano shield interface

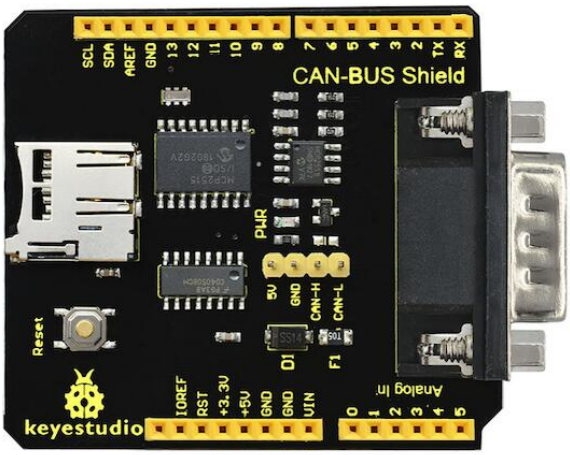


Fig. 9: Keyestudio CAN-BUS, an example of an Arduino shield (Credit: Keyestudio)

Arduino Uno R3 This is the form factor of the Arduino Uno R3 board.

Relevant devicetree node labels:

- `arduino_header` See `arduino-header-r3` for details on GPIO pin definitions and includes for use in devicetree files.
- `arduino_adc` See `arduino,uno-adc`
- `arduino_pwm` See `arduino-header-pwm`
- `arduino_serial`
- `arduino_i2c`
- `arduino_spi`

For technical details, see Arduino Uno R3 pinout.

Camera and display connectors These describe connections to cameras and displays (strictly speaking not shields).

- `arducam,dvp-20pin-connector`
- `nxp,cam-44pins-connector`
- `nxp,parallel-lcd-connector`
- `raspberrypi,csi-connector`
- `st,dsi-lcd-qsh-030-connector`
- `st,dvp-cam-zif-30-connector`
- `weact,dcmi-camera-connector`

ESP-01 This is an 8-pin header for ESP-01 Wi-Fi modules.

Relevant devicetree node labels:

- `esp_01_header` See `esp-01-header` for details on GPIO pin definitions and includes for use in devicetree files.
- `esp_01_serial`

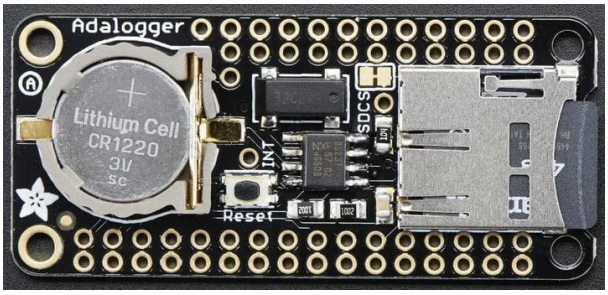


Fig. 10: Adafruit Adalogger, an example of a Featherwing (Credit: Adafruit)

Feather This is the form factor of the Adafruit Feather series of boards. Shields intended for Feather boards are called Featherwings.

Relevant devicetree node labels:

- `feather_header` See `adafruit-feather-header` for GPIO pin definitions.
- `feather_adc`
- `feather_i2c`
- `feather_serial`
- `feather_spi`

Microbit This is for the edge connector of the Microbit boards.

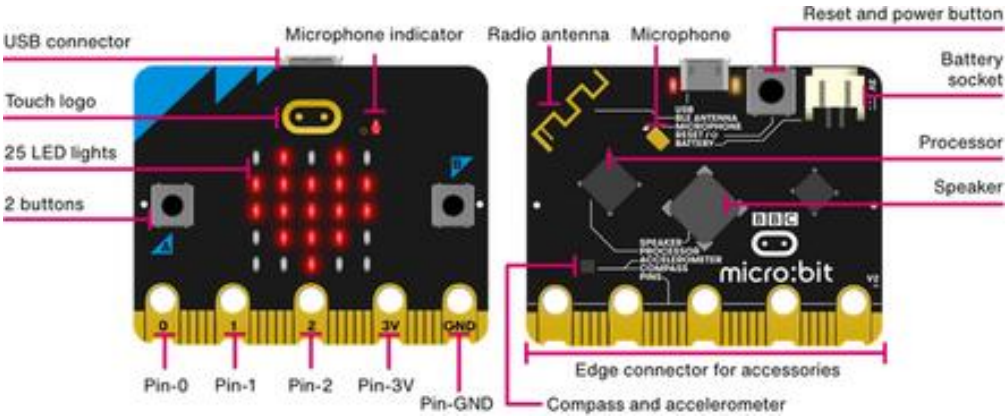


Fig. 11: Microbit V2 board uses the Microbit shield interface

See `microbit,edge-connector` for GPIO pin definitions and links to technical requirements.

mikroBUS™ This is an interface standard for add-on boards, developed by Mikroe.

Relevant devicetree node labels:

- `mikrobus_header` See `mikro-bus` for GPIO pin definitions and links to technical specifications.
- `mikrobus_adc`
- `mikrobus_i2c`
- `mikrobus_pwm`
- `mikrobus_spi`
- `mikrobus_serial`

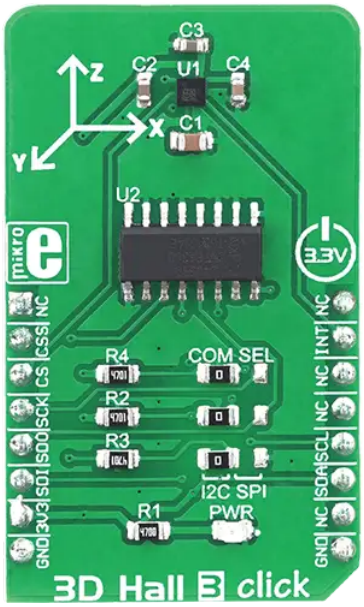


Fig. 12: 3D Hall 3 Click, an example of a mikroBUS™ shield

Note that boards with several mikroBUS™ connectors might define for example `mikrobus_2_spi`.

Pico This is the form factor of the Raspberry Pi Pico boards.

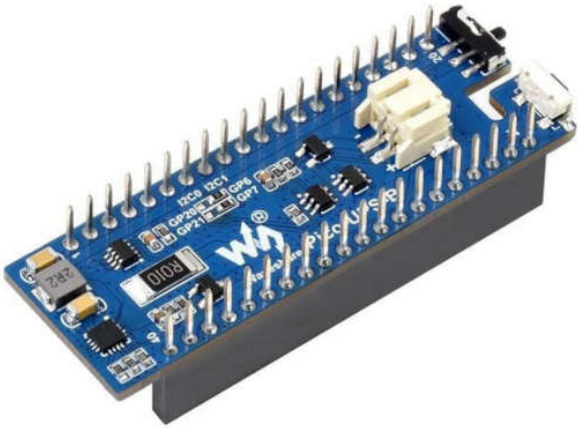


Fig. 13: Waveshare Pico UPS-B, an example of a Pico shield

Relevant devicetree node labels:

- `pico_header` See `raspberrypi,pico-header` for GPIO pin definitions.
- `pico_i2c` A node label that refers to the same node as either the node label `pico_i2c0` or `pico_i2c1`. It references the node that should be used with priority or as default.
- `pico_i2c0`
- `pico_i2c1`
- `pico_serial`
- `pico_spi`

ST Morpho Development boards from ST Microelectronics often uses the ST Morpho shield interface.

Fig. 14: X-NUCLEO-GFX01M2, an example of an ST Morpho shield

Relevant devicetree node labels:

- `st_morpho_header` See `st-morpho-header` for details on GPIO pin definitions and includes for use in devicetree files.
- `st_morpho_lcd_spi`
- `st_morpho_flash_spi`

ST Zio STM32 Nucleo-144 development boards from ST Microelectronics expose the ST Zio connector, an extension of the Arduino Uno V3 connector that gives access to more of the STM32 I/Os through four headers (CN7, CN8, CN9 and CN10).

Relevant devicetree node labels:

- `st_zio_header` See `st-zio-header` for details on GPIO pin definitions and includes for use in devicetree files.

STMod+ This is a 20-pin expansion connector found on some STMicroelectronics Discovery and Evaluation boards.

Relevant devicetree node labels:

- `stmod_plus_connector` See `st,stmod-plus-connector` for details on GPIO pin definitions and includes for use in devicetree files.
- `stmod_adc`
- `stmod_i2c`
- `stmod_pwm`
- `stmod_serial`
- `stmod_spi`

Boards may expose additional interface labels when a peripheral is wired to the STMod+ connector and enabled in the board devicetree.

WisBlock This is a modular interface standard for add-on boards, developed by RAKwireless. It defines Core, Sensor, I/O and Power Slots, each accepting a module of the matching class through a 24-pin or 40-pin WisConnector.

Relevant devicetree node labels:

- `wisblock_io` See `wisblock-io-slot` for GPIO pin definitions.

Fig. 15: RAK19007, an example of a WisBlock Base Board (Credit: RAKwireless)

- `wisblock_sensor_a` See `wisblock-sensor-slot` for GPIO pin definitions.
- `wisblock_adc`
- `wisblock_pwm`
- `wisblock_i2c1`
- `wisblock_i2c2`
- `wisblock_i2s`
- `wisblock_pdm`
- `wisblock_spi`
- `wisblock_uart0`
- `wisblock_uart1`

Note that Sensor Slots are labeled `wisblock_sensor_a` through `wisblock_sensor_f` after the Slot they implement.

Xiao This is the form factor of the Seeeduino XIAO boards.

Relevant devicetree node labels:

- `xiao_d` See `seeed,xiao-gpio` for GPIO pin definitions.
- `xiao_spi`
- `xiao_i2c`
- `xiao_serial`
- `xiao_adc`
- `xiao_dac`

zephyr_i2c / Stemma QT / Quiic These are four-pin I2C connectors. SparkFun calls these connectors "Qwiic", and Adafruit calls them "Stemma QT". The I2C connectors have four pins; GND, +3.3 Volt, I2C data and I2C clock. The most common physical connector is the 1.0 mm pitch JST-SH.

Fig. 16: Seeed Studio XIAO Expansion Board, an example of a Xiao shield (Credit: Seeed Studio)

Due to the different brand names, the interface is labeled “zephyr_i2c”.

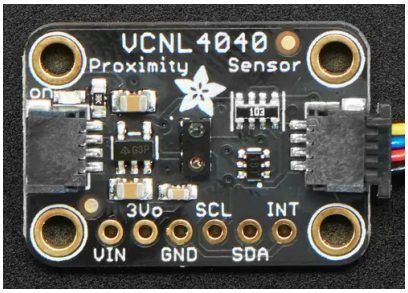


Fig. 17: Adafruit VCNL4040, an example of a zephyr_i2c shield (Credit: Adafruit)

See `stemma-qt-connector` and `grove-header` for descriptions and links to further details.

Relevant devicetree node labels:

- `zephyr_i2c`

ST M.2 serial memory connector Some STMicroelectronics Nucleo-144 development boards expose an ST-specific M.2 serial memory connector used to interface with external serial memories through XSPI, together with auxiliary signals such as I2C and connector GPIOs.

Fig. 18: B-M2MEM-PACK1, an example of an ST M.2 memory shield.

Relevant devicetree node labels:

- `m2mem_connector` See `st,m2-memory-connector` for details on GPIO pin definitions and includes for use in devicetree files.
- `m2mem_i2c`
- `m2mem_xspi`

Shield porting and configuration

Shield configuration files are available in the board directory under boards/shields:

```
boards/shields/<shield>
├── shield.yml
├── <shield>.overlay
├── Kconfig.shield
├── Kconfig.defconfig
└── pre_dt_shield.cmake
```

These files provides shield configuration as follows:

- **shield.yml**: This file provides metadata about the shield in YAML format. It must contain the following fields:
  - `name`: Name of the shield used in Kconfig and build system (required)
  - `full_name`: Full commercial name of the shield (required)
  - `vendor`: Manufacturer/vendor of the shield (required)
  - `supported_features`: List of hardware features the shield supports (optional). In order to help users identify the features a shield supports without having to dig into its overlay file, the `supported_features` field can be used to list the types of features the shield supports. The values should be the same as the ones defined in the dts/bindings/binding-types.txt file.

  Example:

  ```yaml
  name: foo_shield
  full_name: Foo Shield for Arduino
  vendor: acme
  supported_features:
    - display
    - input
  ```

- **<shield>.overlay**: This file provides a shield description in devicetree format that is merged with the board's *devicetree* before compilation.
- **Kconfig.shield**: This file defines shield Kconfig symbols that will be used for default shield configuration. To ease use with applications, the default shield configuration here should be consistent with those in the *Write your devicetree*.
- **Kconfig.defconfig**: This file defines the default shield configuration. It is made to be consistent with the *Write your devicetree*. Hence, shield configuration should be done by keeping in mind that features activation is application responsibility.
- **pre_dt_shield.cmake**: This optional file can be used to pass additional arguments to the devicetree compiler `dtc`.

Besides, in order to avoid name conflicts with devices that may be defined at board level, it is advised, specifically for shields devicetree descriptions, to provide a device nodelabel is the form <device>_<shield>, for instance:

```
sdhc_myshield: sdhc@1 {
        reg = <1>;
        ...
};
```

Adding Source Code

It is possible to add source code to shields, as a way to meet configuration requirements that are specific to the shield (e.g: initialization routines, timing constraints, etc), in order to enable it for proper operation with the different Zephyr components.

> **Note**
>
> Source code in shields shall not be used for purposes other than the one described above. Generic functionalities that could be reused among shields (and/or targets) shall not be captured here.

To effectively incorporate source code: add a `CMakeLists.txt` file, as well as the corresponding source files (referenced in CMake similar to other areas of Zephyr, e.g: boards).

Board compatibility

Hardware shield-to-board compatibility depends on the use of well-known connectors used on popular boards (such as Arduino and 96boards). For software compatibility, boards must also provide a configuration matching their supported connectors.

This should be done at two different level:

- Pinmux: Connector pins should be correctly configured to match shield pins
- Devicetree: A board *devicetree* file, `BOARD.dts` should define an alternate nodelabel for each connector interface. For example, for Arduino I2C:

```
arduino_i2c: &i2c1 {};
```

Board specific shield configuration If modifications are needed to fit a shield to a particular board or board revision, you can override a shield description for a specific board by adding board or board revision overriding files to a shield, as follows:

```
boards/shields/<shield>
└── boards
    ├── <board>_<revision>.overlay
    ├── <board>.overlay
    ├── <board>.defconfig
    ├── <board>_<revision>.conf
    └── <board>.conf
```

Shield variants

Some shields may support several variants or revisions. In that case, it is possible to provide multiple version of the shields description:

```
boards/shields/<shield>
├── <shield_v1>.overlay
├── <shield_v1>.defconfig
├── <shield_v2>.overlay
└── <shield_v2>.defconfig
```

In this case, a shield-particular revision name can be used:

```
west build -b None --shield shield_v2 your_app
```

You can also provide a board-specific configuration to a specific shield revision:

```
boards/shields/<shield>
├── <shield_v1>.overlay
├── <shield_v1>.defconfig
├── <shield_v2>.overlay
├── <shield_v2>.defconfig
└── boards
    └── <shield_v2>
        ├── <board>.overlay
        └── <board>.defconfig
```

GPIO nexus nodes

GPIOs accessed by the shield peripherals must be identified using the shield GPIO abstraction, for example from the `arduino-header-r3` compatible. Boards that provide the header must map the header pins to SOC-specific pins. This is accomplished by including a nexus node that looks like the following into the board devicetree file:

```
arduino_header: connector {
        compatible = "arduino-header-r3";
        #gpio-cells = <2>;
        gpio-map-mask = <0xffffffff 0xffffffc0>;
        gpio-map-pass-thru = <0 0x3f>;
        gpio-map = <0 0 &gpioa 0 0>,    /* A0 */
                   <1 0 &gpioa 1 0>,    /* A1 */
                   <2 0 &gpioa 4 0>,    /* A2 */
                   <3 0 &gpiob 0 0>,    /* A3 */
                   <4 0 &gpioc 1 0>,    /* A4 */
                   <5 0 &gpioc 0 0>,    /* A5 */
                   <6 0 &gpioa 3 0>,    /* D0 */
                   <7 0 &gpioa 2 0>,    /* D1 */
                   <8 0 &gpioa 10 0>,   /* D2 */
                   <9 0 &gpiob 3 0>,    /* D3 */
                   <10 0 &gpiob 5 0>,   /* D4 */
                   <11 0 &gpiob 4 0>,   /* D5 */
                   <12 0 &gpiob 10 0>,  /* D6 */
                   <13 0 &gpioa 8 0>,   /* D7 */
                   <14 0 &gpioa 9 0>,   /* D8 */
                   <15 0 &gpioc 7 0>,   /* D9 */
                   <16 0 &gpiob 6 0>,   /* D10 */
                   <17 0 &gpioa 7 0>,   /* D11 */
                   <18 0 &gpioa 6 0>,   /* D12 */
                   <19 0 &gpioa 5 0>,   /* D13 */
                   <20 0 &gpiob 9 0>,   /* D14 */
                   <21 0 &gpiob 8 0>;   /* D15 */
};
```

This specifies how Arduino pin references like `<&arduino_header 11 0>` are converted to SOC gpio pin references like `<&gpiob 4 0>`.

In Zephyr GPIO specifiers generally have two parameters (indicated by `#gpio-cells = <2>`): the pin number and a set of flags. The low 6 bits of the flags correspond to features that can be configured in devicetree. In some cases it's necessary to use a non-zero flag value to tell the driver how a particular pin behaves, as with:

```
drdy-gpios = <&arduino_header 11 GPIO_ACTIVE_LOW>;
```

After preprocessing this becomes `<&arduino_header 11 1>`. Normally the presence of such a flag would cause the map lookup to fail, because there is no map entry with a non-zero flags value. The `gpio-map-mask` property specifies that, for lookup, all bits of the pin and all but the low 6 bits of the flags are used to identify the specifier. Then the `gpio-map-pass-thru` specifies that the low 6 bits of the flags are copied over, so the SOC GPIO reference becomes `<&gpiob 4 1>` as intended.

See nexus node for more information about this capability.

6.10 Virtualization

These pages document Zephyr virtualization facilities.

6.10.1 Virtual I/O (VIRTIO)

Overview

Virtual I/O (VIRTIO) is a protocol used for communication with various devices, typically used in virtualized environments. Its main goal is to provide an efficient and standardized mechanism for interfacing with virtual devices from within a virtual machine. The communication relies on virtqueues and standard transfer methods like PCI or MMIO.

Concepts

Virtio defines various components used during communication and initialization. It specifies both the host (named "device" in the specification) and guest (named "driver" in the specification) sides. Currently Zephyr can only work as a guest. On top of the facilities exposed by the Virtio driver, a driver for a specific device (e.g. network card) can be implemented.

A high-level overview of a system with a Virtio device is shown below.

Configuration space Each device provides configuration space, used for initialization and configuration. It allows selection of device and driver features, enabling specific virtqueues and setting their addresses. Once the device is configured, most of its configuration cannot be changed without resetting the device. The exact layout of the configuration space depends on the transfer method.

Driver and device features The configuration space provides a way to negotiate feature bits, determining some non-mandatory capabilities of the devices. The exact available feature bits depend on the device and platform.

Device-specific configuration Some of the devices offer device-specific configuration space, providing additional configuration options.

Virtqueues The main mechanism used for transferring data between host and guest is a virtqueue. Specific devices have different numbers of virtqueues, for example devices supporting bidirectional transfer usually have one or more tx/rx virtqueue pairs. Virtio specifies two types of virtqueues: split virtqueues and packed virtqueues. Zephyr currently supports only split virtqueues.

Split virtqueues A split virtqueue consists of three parts: descriptor table, available ring and used ring.

The descriptor table holds descriptors of buffers, that is their physical addresses, lengths and flags. Each descriptor is either device writeable or driver writeable. The descriptors can be

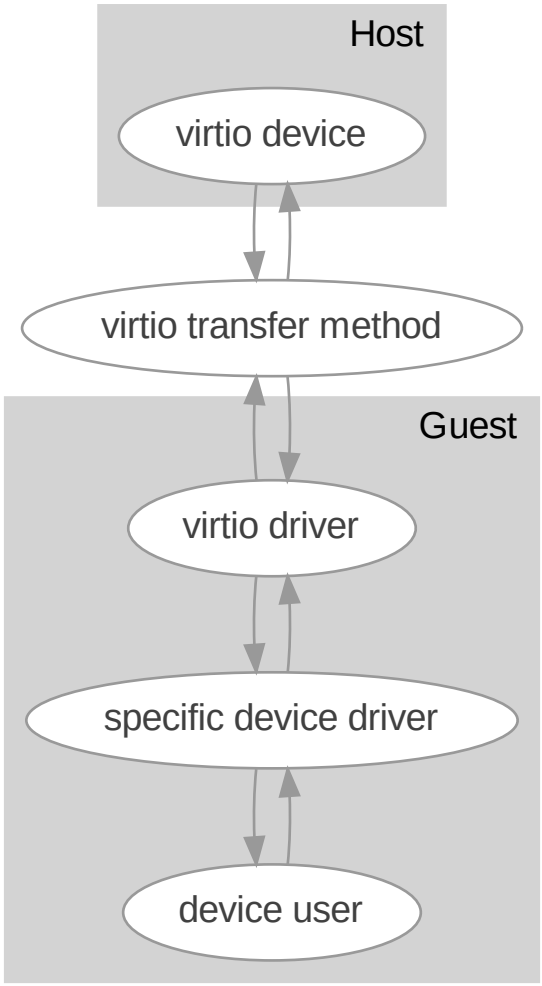


Fig. 19: Virtual I/O overview

chained, creating descriptor chains. Typically a chain begins with descriptors containing the data for the device to read and ends with the device writeable part, where the device places its response.

The main part of the available ring is a circular buffer of references (in the form of indexes) to the descriptors in the descriptor table. Once the guest decides to send the data to the host, it adds the index of the head of the descriptor chain to the top of the available ring.

The used ring is similar to the available ring, but it's used by the host to return descriptors to the guest. In addition to storing descriptor indexes, it also provides information about the amount of data written to them.

Common Virtio libraries

Zephyr provides an API for interfacing with Virtio devices and virtqueues, which allows performing necessary operations over the lifetime of the Virtio device.

Device initialization Once the Virtio driver finishes performing low-level initialization common to the all devices using a given transfer method, like finding device on the bus and mapping Virtio structures, the device specific driver steps in and performs the next stages of initialization with the help of the Virtio API.

The first thing the device-specific driver does is feature bits negotiation. It uses `virtio_read_device_feature_bit()` to determine which features the device offers, and then selects the ones it needs using `virtio_write_driver_feature_bit()`. After all required features have been selected, the device-specific driver calls `virtio_commit_feature_bits()`. Then, virtqueues are initialized with `virtio_init_virtqueues()`. This function enumerates the virtqueues, invoking the provided callback `virtio_enumerate_queues` to determine the required size of each virtqueue. Initialization process is finalized by calling `virtio_finalize_init()`. From this point, if none of the functions returned errors, the virtqueues are operational. If the specific device provides one, the device-specific config can be obtained by calling `virtio_get_device_specific_config()`.

Virtqueue operation Once the virtqueues are operational, they can be used to send and receive data. To do so, the pointer to the nth virtqueue has to be acquired using `virtio_get_virtqueue()`. To send data consisting of a descriptor chain, `virtq_add_buffer_chain()` has to be used. Along the descriptor chain, it takes pointer to the callback that will be invoked once the device returns the given descriptor chain. After that, the virtqueue has to be notified using `virtio_notify_virtqueue()` from the Virtio API.

Guest-side Virtio drivers

Currently Zephyr provides drivers for Virtio over PCI and Virtio over MMIO and drivers for three devices using virtio - virtiofs, used to access the filesystem of the host, virtio-entropy, used as an entropy source, and virtio-blk, used to access a block device.

Virtiofs This driver provides support for virtiofs - a filesystem allowing a virtual machine guest to access a directory on the host. It uses FUSE messages to communicate between the host and the guest in order to perform filesystem operations such as opening and reading files. Every time the guest wants to perform some filesystem operation it places in the virtqueue a descriptor chain starting with the device readable part, containing the FUSE input header and input data, and ending it with the device writeable part, with place for the FUSE output header and output data.

Virtio-entropy This driver allows using virtio-entropy as an entropy source in Zephyr. The operation of this device is simple - the driver places a buffer in the virtqueue and receives it back, filled with random data.

Virtio-blk This driver exposes a virtio-blk block device to Zephyr's disk access layer, so it can be used through the *Disk Access API* and, on top of it, a filesystem. It keeps a single request in flight, submitting a descriptor chain made of a request header, the caller's data buffer as one or more scatter-gather segments, and a status byte. See *VirtIO Block* for details.

Virtio samples

A sample showcasing the use of a driver relying on Virtio is provided in virtiofs. If you wish to check code interfacing directly with the Virtio driver, you can check the virtiofs driver, especially `virtiofs_init()` for initialization and `virtiofs_send_receive()` with the `virtiofs_recv_cb()` for data transfer to/from the Virtio device.

API Reference

VIRTIO

Virtqueue Interface

Chapter 7

Contributing to Zephyr

Contributions from the community are the backbone of the project. Whether it is by submitting code, improving documentation, or proposing new features, your efforts are highly appreciated. This page lists useful resources and guidelines to help you in your contribution journey.

7.1 General Guidelines

7.1.1 Contribution Guidelines

As an open-source project, we welcome and encourage the community to submit patches directly to the project. In our collaborative open source environment, standards and methods for submitting changes help reduce the chaos that can result from an active development community.

This document explains how to participate in project conversations, log bugs and enhancement requests, and submit patches to the project so your patch will be accepted quickly in the codebase.

Prerequisites

As a contributor, you'll want to be familiar with the Zephyr project, how to configure, install, and use it as explained in the Zephyr Project website and how to set up your development environment as introduced in the Zephyr *Getting Started Guide*.

You should be familiar with common developer tools such as Git and CMake, and platforms such as GitHub.

If you haven't already done so, you'll need to create a (free) GitHub account on https://github.com and have Git tools available on your development system.

> **Note**
>
> The Zephyr development workflow supports all 3 major operating systems (Linux, macOS, and Windows) but some of the tools used in the sections below are only available on Linux and macOS. On Windows, instead of running these tools yourself, you will need to rely on the Continuous Integration (CI) service using Github Actions, which runs automatically on GitHub when you submit your Pull Request (PR). You can see any failure results in the workflow details link near the end of the PR conversation list. See *Continuous Integration* for more information

Licensing

Licensing is very important to open source projects. It helps ensure the software continues to be available under the terms that the author desired.

Zephyr uses the Apache 2.0 license (as found in the LICENSE file in the project's GitHub repo) to strike a balance between open contribution and allowing you to use the software however you would like to. The Apache 2.0 license is a permissive open source license that allows you to freely use, modify, distribute and sell your own products that include Apache 2.0 licensed software. (For more information about this, check out articles such as Why choose Apache 2.0 licensing and Top 10 Apache License Questions Answered).

A license tells you what rights you have as a developer, as provided by the copyright holder. It is important that the contributor fully understands the licensing rights and agrees to them. Sometimes the copyright holder isn't the contributor, such as when the contributor is doing work on behalf of a company.

Components using other Licenses There are some imported or reused components of the Zephyr project that use other licensing, as described in *Licensing of Zephyr Project components*.

Importing code into the Zephyr OS from other projects that use a license other than the Apache 2.0 license needs to be fully understood in context and approved by the Zephyr governing board.

By carefully reviewing potential contributions and also enforcing a *Developer Certification of Origin (DCO)* for contributed code, we can ensure that the Zephyr community can develop products with the Zephyr Project without concerns over patent or copyright issues.

See *Contributing External Components* for more information about this contributing and review process for imported components.

Licensing of Zephyr Project components Zephyr as a whole is licensed under the Apache 2.0 License. It does, however, import or reuse a small number of packages, scripts and other files that are covered by other licenses. In some cases there is no way to add a license header to those files, so their licensing is declared centrally, in a machine-readable form, in the REUSE.toml file at the root of the repository (following the REUSE specification).

The sections below are **generated automatically** from that metadata, so they always reflect the actual state of the tree. To add, update or remove an entry, edit the corresponding `[[annotations]]` block in REUSE.toml rather than this page (see *Contributing External Components*).

> **Note**
>
> This page lists licensing *exceptions* only. It does **not** define the license of the Zephyr project itself, which is Apache 2.0 as specified in the LICENSE file at the root of the repository.

Documented components

- *Bootstrap JavaScript and CSS Files*
- *Bosch BMI08x Sensor Configuration File*
- *Coccinelle Scripts*
- *Continuous Integration Scripts*
- *Doxygen Awesome CSS Theme Files*
- *ENE KB1200_EVB Board OpenOCD Configuration*
- *FUSE Interface Definition Header File*

- *GCOV Coverage Header File*
- *getopt Command-line Parsing Files*
- *Godot Documentation Theme Files*
- *HTTP Parser Files*
- *Kconfiglib Configuration Library*
- *littlefs Configuration Files*
- *Minimal libc atoi File*
- *Minimal libc BSD String and Conversion Files*
- *Modern CMake Sphinx Domain*
- *noUiSlider Library*
- *OP-TEE Message Interface Headers*
- *OpenThread Spinel HDLC RCP Host Interface Files*
- *Popper.js Library*
- *POSIX fnmatch Files*
- *POSIX sys/stat.h Header*
- *Python Devicetree Library Files*
- *Python Devicetree Library Test Files*
- *RISC-V Control and Status Register Header*
- *RV32M1 VEGA Board OpenOCD Configuration*
- *TF-M Secure Partition Sample Files*
- *Thread-Metric RTOS Test Suite Source Files*
- *Tippy.js Library*
- *UF2 Conversion Script*
- *USB Device Stack Headers*
- *WireGuard VPN Files*
- *Xen Hypervisor Interface Files*
- *xoshiro128++ PRNG*
- *Xtensa Startup Assembly Files*

Bootstrap JavaScript and CSS Files

Origin
getbootstrap.com

License
MIT

Used by the dashboard tool only and never linked into a Zephyr firmware image.

Files (2):

- scripts/dashboard/static/css/bootstrap-chop.css
- scripts/dashboard/static/js/bootstrap-chop.js

Bosch BMI08x Sensor Configuration File

Origin
github.com/BoschSensortec/BMI08x-Sensor-API

License
BSD-3-Clause

Enabled by
`CONFIG_BMI08X`

Sensor configuration data compiled into the firmware only when the BMI08x driver is enabled.

Files (1):

- drivers/sensor/bosch/bmi08x/bmi08x_config_file.h

Coccinelle Scripts

Origin
coccinelle.gitlabpages.inria.fr/website

License
GPL-2.0

Used by Coccinelle, a tool for transforming C code, and never linked into the firmware.

Files (8):

- scripts/coccicheck
- scripts/coccinelle/array_size.cocci
- scripts/coccinelle/deref_null.cocci
- scripts/coccinelle/mini_lock.cocci
- scripts/coccinelle/noderef.cocci
- scripts/coccinelle/returnvar.cocci
- scripts/coccinelle/semicolon.cocci
- scripts/coccinelle/unsigned_lesser_than_zero.cocci

Continuous Integration Scripts

Origin
www.kernel.org

License
GPL-2.0

Used in Continuous Integration (CI) and never linked into the firmware.

Files (3):

- scripts/checkpatch.pl
- scripts/checkstack.pl
- scripts/spelling.txt

Doxygen Awesome CSS Theme Files

Origin
github.com/jothepro/doxygen-awesome-css

License
MIT

Style the Doxygen API documentation and are never linked into a firmware.

Files (4):

- doc/_doxygen/doxygen-awesome-darkmode-toggle.js
- doc/_doxygen/doxygen-awesome-sidebar-only-darkmode-toggle.css
- doc/_doxygen/doxygen-awesome-sidebar-only.css
- doc/_doxygen/doxygen-awesome.css

ENE KB1200_EVB Board OpenOCD Configuration

Origin
openocd.org

License
GPL-2.0-or-later

Used by OpenOCD when programming and debugging the kb1200_evb board. Never linked into the firmware.

Files (1):

- boards/ene/kb1200_evb/support/openocd.cfg

FUSE Interface Definition Header File

License
`GPL-2.0 WITH Linux-syscall-note OR BSD-2-Clause`

Enabled by
`CONFIG_FUSE_CLIENT`

Used under the BSD-2-Clause option of its dual license. Only built when the gating option is enabled.

Files (1):

- subsys/fs/fuse_client/fuse_abi.h

GCOV Coverage Header File

Origin
gcc.gnu.org

License
`GPL-3.0-or-later WITH GCC-exception-3.1`

Enabled by
`CONFIG_COVERAGE_GCOV`

Linked into the firmware only when GCOV coverage is enabled; covered by the GCC Runtime Library Exception.

Files (1):

- subsys/testsuite/coverage/coverage.h

getopt Command-line Parsing Files

Origin
www.netbsd.org

License
BSD-3-Clause

Enabled by
`CONFIG_GETOPT`

Compiled into the firmware only when getopt() support is enabled.

Files (3):

- lib/utils/getopt/README
- lib/utils/getopt/getopt.c
- lib/utils/getopt/getopt_long.c

Godot Documentation Theme Files

Origin
github.com/godotengine/godot-docs

License
CC-BY-3.0

Customize the Sphinx theme used to render the documentation and are never linked into a firmware.

Files (4):

- doc/_static/css/custom.css
- doc/_static/css/dark.css
- doc/_static/css/light.css
- doc/_static/js/custom.js

HTTP Parser Files

Origin
github.com/nodejs/http-parser

License
MIT

Enabled by
`CONFIG_HTTP_PARSER`

Compiled into the firmware only when the HTTP parser is enabled.

Files (5):

- include/zephyr/net/http/parser.h
- include/zephyr/net/http/parser_state.h
- include/zephyr/net/http/parser_url.h
- subsys/net/lib/http/http_parser.c
- subsys/net/lib/http/http_parser_url.c

Kconfiglib Configuration Library

Origin
github.com/ulfalizer/Kconfiglib

License
ISC

Used to process Kconfig files at build time and never linked into the firmware.

Files (4):

- scripts/kconfig/guiconfig.py
- scripts/kconfig/kconfig.py
- scripts/kconfig/kconfiglib.py
- scripts/kconfig/menuconfig.py

littlefs Configuration Files

Origin
github.com/littlefs-project/littlefs

License
BSD-3-Clause

Enabled by
`CONFIG_FILE_SYSTEM_LITTLEFS`

Compiled into the firmware only when the littlefs file system is enabled.

Files (2):

- modules/littlefs/zephyr_lfs_config.h
- modules/littlefs/zephyr_lfs_crc.c

Minimal libc atoi File

Origin
musl.libc.org

License
MIT

Enabled by
`CONFIG_MINIMAL_LIBC`

Compiled into the firmware only when the minimal C library is selected.

Files (1):

- lib/libc/minimal/source/stdlib/atoi.c

Minimal libc BSD String and Conversion Files

License
BSD-3-Clause

Enabled by
`CONFIG_MINIMAL_LIBC`

Compiled into the firmware only when the minimal C library is selected.

Files (5):

- lib/libc/minimal/source/stdlib/strtol.c
- lib/libc/minimal/source/stdlib/strtoll.c
- lib/libc/minimal/source/stdlib/strtoul.c
- lib/libc/minimal/source/stdlib/strtoull.c
- lib/libc/minimal/source/string/strstr.c

Modern CMake Sphinx Domain

Origin
gitlab.kitware.com/cmake/cmake/-/tree/master/Utilities/Sphinx

License
BSD-3-Clause

Provide the CMake domain used to render the documentation and are never linked into a firmware.

Files (3):

- doc/_extensions/moderncmakedomain/__init__.py
- doc/_extensions/moderncmakedomain/cmake.py
- doc/_extensions/moderncmakedomain/colors.py

noUiSlider Library

Origin
refreshless.com/nouislider

License
MIT

Provides the flash/RAM range sliders in the board catalog and is never linked into a firmware.

Files (2):

- doc/_extensions/zephyr/domain/static/css/nouislider.min.css
- doc/_extensions/zephyr/domain/static/js/nouislider.min.js

OP-TEE Message Interface Headers

Origin
www.op-tee.org

License
BSD-2-Clause

Enabled by
`CONFIG_OPTEE`

Compiled into the firmware only when the OP-TEE driver is enabled.

Files (3):

- drivers/tee/optee/optee_msg.h
- drivers/tee/optee/optee_rpc_cmd.h
- drivers/tee/optee/optee_smc.h

OpenThread Spinel HDLC RCP Host Interface Files

Origin
openthread.io

License
BSD-3-Clause

Enabled by
`CONFIG_HDLC_RCP_IF`

Linked into the firmware only when the Spinel HDLC RCP host interface is enabled.

Files (3):

- modules/openthread/platform/hdlc_interface.cpp
- modules/openthread/platform/hdlc_interface.hpp
- modules/openthread/platform/radio_spinel.cpp

Popper.js Library

Origin
popper.js.org

License
MIT

Used by Tippy.js to position the documentation's Doxygen tooltips and is never linked into a firmware.

Files (1):

- doc/_extensions/zephyr/doxytooltip/static/tippy/popper.min.js

POSIX fnmatch Files

License
BSD-3-Clause

Enabled by
`CONFIG_POSIX_C_LIB_EXT`

Compiled into the firmware only when the POSIX C library extensions are enabled.

Files (2):

- include/zephyr/posix/fnmatch.h
- subsys/portability/posix/c_lib_ext/fnmatch.c

POSIX sys/stat.h Header

License
BSD-3-Clause

Provides the POSIX file-status definitions; compiled into the firmware wherever it is included.

Files (1):

- include/zephyr/posix/sys/stat.h

Python Devicetree Library Files

Origin
github.com/zephyrproject-rtos/python-devicetree

License
BSD-3-Clause

Parse devicetree sources at build time and are never linked into the firmware.

Files (8):

- scripts/dts/gen_defines.py
- scripts/dts/python-devicetree/src/devicetree/_private.py
- scripts/dts/python-devicetree/src/devicetree/dtlib.py
- scripts/dts/python-devicetree/src/devicetree/edtlib.py
- scripts/dts/python-devicetree/tests/test-multidir.dts

- scripts/dts/python-devicetree/tests/test.dts
- scripts/dts/python-devicetree/tests/test_dtlib.py
- scripts/dts/python-devicetree/tests/test_edtlib.py

Python Devicetree Library Test Files

Origin
github.com/zephyrproject-rtos/python-devicetree

License
BSD-3-Clause

Used only when testing the python-devicetree library and never linked into the firmware.

Files (104):

- scripts/dts/python-devicetree/tests/test-bindings-2/multidir.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/allow-and-blocklist-child.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/allow-and-blocklist-multilevel.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/allow-and-blocklist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/allow-not-list.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/allowlist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/block-not-list.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/blocklist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-bad-name.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-base-1.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-base-2.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-child-base.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-child-union.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-dup.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/class-union.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/empty-allowlist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/empty-blocklist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/filter-child-bindings.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/include-2.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/include-invalid-keys.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/include-invalid-type.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/include-no-list.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/include-no-name.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/include.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/intermixed.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/simple.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/simple_filter_allowlist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-include/simple_filter_blocklist.yaml

- scripts/dts/python-devicetree/tests/test-bindings-include/simple_inherit.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/base.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/base_amend.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/base_inherit.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/base_multi.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/compat_desc.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/compat_desc_base.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/compat_desc_multi.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/default_base.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/default_override.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/default_override_include.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/default_required.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/diamond.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/filter_allows_notblocked.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/filter_among_allowed.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/filter_among_notblocked.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_child_propconst.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_child_propdefault.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_child_propenum.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_child_propreq.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_child_proptype.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_grandchild_propconst.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_grandchild_propdefault.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_grandchild_propenum.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_grandchild_propreq.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_grandchild_proptype.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_propconst.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_propdefault.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_propenum.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_propreq.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/invalid_proptype.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/simple.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/simple_allowlist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/simple_blocklist.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/simple_inherit.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/thing.yaml
- scripts/dts/python-devicetree/tests/test-bindings-init/vnd,thing.yaml
- scripts/dts/python-devicetree/tests/test-bindings/bar-bus.yaml
- scripts/dts/python-devicetree/tests/test-bindings/child-binding-with-compat.yaml

- scripts/dts/python-devicetree/tests/test-bindings/child-binding.yaml
- scripts/dts/python-devicetree/tests/test-bindings/child.yaml
- scripts/dts/python-devicetree/tests/test-bindings/defaults.yaml
- scripts/dts/python-devicetree/tests/test-bindings/dependency-mode.yaml
- scripts/dts/python-devicetree/tests/test-bindings/deprecated.yaml
- scripts/dts/python-devicetree/tests/test-bindings/device-on-any-bus.yaml
- scripts/dts/python-devicetree/tests/test-bindings/device-on-bar-bus.yaml
- scripts/dts/python-devicetree/tests/test-bindings/device-on-foo-bus.yaml
- scripts/dts/python-devicetree/tests/test-bindings/enums.yaml
- scripts/dts/python-devicetree/tests/test-bindings/false-positive.yaml
- scripts/dts/python-devicetree/tests/test-bindings/foo-bus.yaml
- scripts/dts/python-devicetree/tests/test-bindings/foo-optional.yaml
- scripts/dts/python-devicetree/tests/test-bindings/foo-required.yaml
- scripts/dts/python-devicetree/tests/test-bindings/gpio-dst.yaml
- scripts/dts/python-devicetree/tests/test-bindings/gpio-src.yaml
- scripts/dts/python-devicetree/tests/test-bindings/grandchild-1.yaml
- scripts/dts/python-devicetree/tests/test-bindings/grandchild-2.yaml
- scripts/dts/python-devicetree/tests/test-bindings/grandchild-3.yaml
- scripts/dts/python-devicetree/tests/test-bindings/interrupt-1-cell.yaml
- scripts/dts/python-devicetree/tests/test-bindings/interrupt-2-cell.yaml
- scripts/dts/python-devicetree/tests/test-bindings/interrupt-3-cell.yaml
- scripts/dts/python-devicetree/tests/test-bindings/min-max-len.yaml
- scripts/dts/python-devicetree/tests/test-bindings/min-max.yaml
- scripts/dts/python-devicetree/tests/test-bindings/multidir.yaml
- scripts/dts/python-devicetree/tests/test-bindings/order-1.yaml
- scripts/dts/python-devicetree/tests/test-bindings/order-2.yaml
- scripts/dts/python-devicetree/tests/test-bindings/parent.yaml
- scripts/dts/python-devicetree/tests/test-bindings/phandle-array-controller-0.yaml
- scripts/dts/python-devicetree/tests/test-bindings/phandle-array-controller-1.yaml
- scripts/dts/python-devicetree/tests/test-bindings/phandle-array-controller-2.yaml
- scripts/dts/python-devicetree/tests/test-bindings/props.yaml
- scripts/dts/python-devicetree/tests/test-bindings/test-cpu-fallback.yaml
- scripts/dts/python-devicetree/tests/test-wrong-bindings/wrong-address-cells-default.yaml
- scripts/dts/python-devicetree/tests/test-wrong-bindings/wrong-phandle-array-name.yaml
- scripts/dts/python-devicetree/tests/test-wrong-bindings/wrong-size-cells-default.yaml
- scripts/dts/python-devicetree/tests/test-wrong-bindings/wrong-specifier-space-type.yaml
- scripts/dts/python-devicetree/tests/test-wrong-bindings/wrong-status-default.yaml

RISC-V Control and Status Register Header

License
SHL-0.51

Enabled by
`CONFIG_RISCV`

RISC-V control and status register access helpers, also available under Apache-2.0; compiled into the firmware only on RISC-V targets.

Files (1):

- include/zephyr/arch/riscv/csr.h

RV32M1 VEGA Board OpenOCD Configuration

License
BSD-3-Clause

Used by OpenOCD when programming and debugging the rv32m1_vega board. Never linked into the firmware.

Files (2):

- boards/openisa/rv32m1_vega/support/openocd_rv32m1_vega_ri5cy.cfg
- boards/openisa/rv32m1_vega/support/openocd_rv32m1_vega_zero_riscy.cfg

TF-M Secure Partition Sample Files

Origin
www.trustedfirmware.org/projects/tf-m

License
BSD-3-Clause

Used only when building the TF-M secure partition integration sample.

Files (2):

- samples/tfm_integration/tfm_secure_partition/dummy_partition/tfm_dummy_partition.yaml.in
- samples/tfm_integration/tfm_secure_partition/dummy_partition/tfm_manifest_list.yaml.in

Thread-Metric RTOS Test Suite Source Files

Origin
github.com/eclipse-threadx/threadx

License
MIT

Linked only into the Thread-Metric RTOS Test Suite test firmware.

Files (12):

- tests/benchmarks/thread_metric/src/tm_api.h
- tests/benchmarks/thread_metric/src/tm_basic_processing_test.c
- tests/benchmarks/thread_metric/src/tm_cooperative_scheduling_test.c
- tests/benchmarks/thread_metric/src/tm_interrupt_preemption_processing_test.c
- tests/benchmarks/thread_metric/src/tm_interrupt_processing_test.c
- tests/benchmarks/thread_metric/src/tm_memory_allocation_test.c
- tests/benchmarks/thread_metric/src/tm_message_processing_test.c

- tests/benchmarks/thread_metric/src/tm_porting_layer.h
- tests/benchmarks/thread_metric/src/tm_porting_layer_zephyr.c
- tests/benchmarks/thread_metric/src/tm_preemptive_scheduling_test.c
- tests/benchmarks/thread_metric/src/tm_synchronization_processing_test.c
- tests/benchmarks/thread_metric/thread_metric_readme.txt

Tippy.js Library

Origin
atomiks.github.io/tippyjs

License
MIT

Renders the tooltips of the documentation's Doxygen tooltip extension and is never linked into a firmware.

Files (1):

- doc/_extensions/zephyr/doxytooltip/static/tippy/tippy-bundle.umd.min.js

UF2 Conversion Script

Origin
github.com/microsoft/uf2

License
MIT

Converts firmware images to UF2 at build time and is never linked into the firmware.

Files (1):

- scripts/build/uf2conv.py

USB Device Stack Headers

License
BSD-3-Clause

Enabled by
`CONFIG_USB_DEVICE_STACK`

Part of the legacy USB device stack, compiled into the firmware only when it is enabled.

Files (2):

- include/zephyr/usb/class/usb_dfu.h
- include/zephyr/usb/usb_device.h

WireGuard VPN Files

Origin
github.com/smartalock/wireguard-lwip

License
BSD-3-Clause

Enabled by
`CONFIG_WIREGUARD`

Compiled into the firmware only when WireGuard is enabled.

Files (7):

- subsys/net/lib/wireguard/crypto/crypto.c
- subsys/net/lib/wireguard/crypto/crypto.h
- subsys/net/lib/wireguard/crypto/refc/blake2s.c
- subsys/net/lib/wireguard/crypto/refc/blake2s.h
- subsys/net/lib/wireguard/crypto/refc/hchacha20.c
- subsys/net/lib/wireguard/crypto/refc/hchacha20.h
- subsys/net/lib/wireguard/wg_crypto.c

Xen Hypervisor Interface Files

Origin
xenproject.org

License
MIT

Enabled by
`CONFIG_XEN`

Compiled into the firmware only when Xen guest support is enabled.

Files (1):

- drivers/xen/gnttab.c

xoshiro128++ PRNG

Origin
prng.di.unimi.it

License
CC0-1.0

Enabled by
`CONFIG_XOSHIRO_RANDOM_GENERATOR`

Compiled into the firmware only when the xoshiro128++ PRNG is selected.

Files (1):

- subsys/random/random_xoshiro128.c

Xtensa Startup Assembly Files

License
MIT

Enabled by
`CONFIG_XTENSA_RESET_VECTOR`

Compiled into the firmware only when the Xtensa reset vector is enabled.

Files (2):

- arch/xtensa/core/startup/memctl_default.S
- arch/xtensa/core/startup/memerror_vector.S

Copyright and License Notices Zephyr follows SPDX/REUSE-style file headers. Add a machine-readable copyright notice and a license identifier at the top of each file so tooling can detect them (for example *west spdx*, which uses the REUSE tool).

The Zephyr Project follows the Community Best Practice for copyright notices from the Linux Foundation, therefore we recommend using the following copyright notice:

```
SPDX-FileCopyrightText: Copyright The Zephyr Project Contributors
```

Include the license identifier alongside it:

```
SPDX-License-Identifier: Apache-2.0
```

Practical guidance:

- Place both lines at the very top of the file using the file's native comment syntax.
- If you authored substantial, original content, you *may* add an additional line for yourself or your organization.

Developer Certification of Origin (DCO)

To make a good faith effort to ensure licensing criteria are met, the Zephyr project requires the Developer Certificate of Origin (DCO) process to be followed.

The DCO is an attestation attached to every contribution made by every developer. In the commit message of the contribution, (described more fully later in this document), the developer simply adds a `Signed-off-by` statement and thereby agrees to the DCO.

When a developer submits a patch, it is a commitment that the contributor has the right to submit the patch per the license. The DCO agreement is shown below and at https://developercertificate.org/.

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the
    best of my knowledge, is covered under an appropriate open
    source license and I have the right under that license to
    submit that work with modifications, whether created in whole
    or in part by me, under the same open source license (unless
    I am permitted to submit under a different license), as
    Indicated in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including
    all personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
    consistent with this project or the open source license(s)
    involved.
```

DCO Sign-Off The "sign-off" in the DCO is a "Signed-off-by:" line in each commit's log message. The Signed-off-by: line must be in the following format:

```
Signed-off-by: Your Name <your.email@example.com>
```

For your commits, replace:

- `Your Name` with your legal name (pseudonyms, hacker handles, and the names of groups are not allowed)
- `your.email@example.com` with the real email address you are using to author the commit. Pseudo or anonymized emails such as `you-id+your-username@users.noreply.github.com` are not allowed. The email must match the one you use to author the commit (CI will fail if there is no match).

You can automatically add the Signed-off-by: line to your commit body using `git commit -s`. Use other commits in the zephyr git history as examples. See *Git Setup* for instructions on configuring user and email settings in Git.

Additional requirements:

- If you are altering an existing commit created by someone else, you must add your Signed-off-by: line without removing the existing one.

AI Coding Assistants

This section provides guidance for contributors using AI tools and assistants when contributing to the Zephyr project.

Licensing and Legal Requirements All contributions must comply with the licensing requirements of the project and be compatible with Zephyr's licensing (e.g. Apache-2.0, see *Licensing* for more details).

Signed-off-by and Developer Certificate of Origin AI agents **must not** add `Signed-off-by` tags. Only humans can legally certify the *Developer Certification of Origin (DCO)*. The human submitter is responsible for:

- Reviewing all AI-generated code.
- Ensuring compliance with licensing requirements.
- Adding their own Signed-off-by tag to certify the DCO.
- Taking full responsibility for the contribution.

Usage disclosure and attribution When AI tools are being used to help write a contribution, proper attribution helps track the evolving role of AI in the development process. Contributions should include an `Assisted-by:` tag in the following format:

```
Assisted-by: [Agent Name]:[Model Version] [Tool1] [Tool2]
```

Where:

- `[Agent Name]` is the name of the AI tool or framework.
- `[Model Version]` is the specific model version used.
- `[Tool1] [Tool2]` are optional specialized analysis tools used.

Basic development tools (git, gcc, make, editors) should not be listed.

Example:

```
Assisted-by: Claude:claude-opus-4.6 coccinelle
```

Source Tree Structure

To clone the main Zephyr Project repository use the instructions in *Get Zephyr and install Python dependencies*.

This section describes the main repository's source tree. In addition to the Zephyr kernel itself, you'll also find the sources for technical documentation, sample code, supported board configurations, and a collection of subsystem tests. All of these are available for developers to contribute to and enhance.

Understanding the Zephyr source tree can help locate the code associated with a particular Zephyr feature.

At the top of the tree, several files are of importance:

`CMakeLists.txt`
: The top-level file for the CMake build system, containing a lot of the logic required to build Zephyr.

`Kconfig`
: The top-level Kconfig file, which refers to the file `Kconfig.zephyr` also found in the top-level directory.

  See *the Kconfig section of the manual* for detailed Kconfig documentation.

`west.yml`
: The *West (Zephyr's meta-tool)* manifest, listing the external repositories managed by the west command-line tool.

The Zephyr source tree also contains the following top-level directories, each of which may have one or more additional levels of subdirectories not described here.

`arch`
: Architecture-specific kernel and system-on-chip (SoC) code. Each supported architecture (for example, x86 and ARM) has its own subdirectory, which contains additional subdirectories for the following areas:

  - architecture-specific kernel source files
  - architecture-specific kernel include files for private APIs

`soc`
: SoC related code and configuration files.

`boards`
: Board related code and configuration files.

`doc`
: Zephyr technical documentation source files and tools used to generate the https://docs.zephyrproject.org web content.

`drivers`
: Device driver code.

`dts`
: *devicetree* source files used to describe non-discoverable board-specific hardware details.

`include`
: Include files for all public APIs, except those defined under `lib`.

`kernel`
: Architecture-independent kernel code.

`lib`
: Library code, including the minimal standard C library.

`misc`
: Miscellaneous code that doesn't belong to any of the other top-level directories.

`samples`
: Sample applications that demonstrate the use of Zephyr features.

`scripts`
: Various programs and other files used to build and test Zephyr applications.

`cmake`
: Additional build scripts needed to build Zephyr.

`subsys`
: Subsystems of Zephyr, including:

  - USB device stack code
  - Networking code, including the Bluetooth stack and networking stacks
  - File system code
  - Bluetooth host and controller

`tests`
: Test code and benchmarks for Zephyr features.

`share`
: Additional architecture independent data. It currently contains Zephyr's CMake package.

Pull Requests and Issues

Before starting on a patch, first check in our issues Zephyr Project Issues system to see what's been reported on the issue you'd like to address. Have a conversation on the Zephyr devel mailing list (or the Zephyr Discord Server) to see what others think of your issue (and proposed solution). You may find others that have encountered the issue you're finding, or that have similar ideas for changes or additions. Send a message to the Zephyr devel mailing list to introduce and discuss your idea with the development community.

It's always a good practice to search for existing or related issues before submitting your own. When you submit an issue (bug or feature request), the triage team will review and comment on the submission, typically within a few business days.

You can find all open pull requests on GitHub and open Zephyr Project Issues in Github issues.

Git Setup

We need to know who you are, and how to contact you. To add this information to your Git installation, set the Git configuration variables `user.name` to your full name, and `user.email` to your email address.

For example, if your name is `Zephyr Developer` and your email address is `z.developer@example.com`:

```
git config --global user.name "Zephyr Developer"
git config --global user.email "z.developer@example.com"
```

Note

`user.name` must be your full name (first and last at minimum), not a pseudonym or hacker handle. The email address that you use in your Git configuration must match the email ad-

dress you use to sign your commits. If they don't match, the CI system will fail your pull request.

If you intend to edit commits using the Github.com UI, ensure that your github profile `email address` and profile `name` also match those used in your git configuration (`user.name` & `user.email`).

Pull Request Guidelines

When opening a new Pull Request, adhere to the following guidelines to ensure compliance with Zephyr standards and facilitate the review process.

If in doubt, it's advisable to explore existing Pull Requests within the Zephyr repository. Use the search filters and labels to locate PRs related to changes similar to the ones you are proposing.

Note

GitHub's default code UI uses 4-character tabs. However, Zephyr follows the Linux kernel coding style, which uses 8-character tabs.

To ensure your view of the code is consistent with other developers, please go to your user preferences on GitHub and change the tab width to 8 spaces.

Commit Message Guidelines Changes are submitted as Git commits. Each commit has a *commit message* describing the change. Acceptable commit messages look like this:

```
[area]: [summary of change]

[Commit message body (must be non-empty)]

Signed-off-by: [Your Full Name] <[your.email@address]>
```

You need to change text in square brackets (`[like this]`) above to fit your commit.

Here is an example of a good commit message.

```
drivers: sensor: abcd1234: fix bus I/O error handling

The abcd1234 sensor driver is failing to check the flags field in
the response packet from the device which signals that an error
occurred. This can lead to reading invalid data from the response
buffer. Fix it by checking the flag and adding an error path.

Signed-off-by: Zephyr Developer <z.developer@example.com>
```

[area]: [summary of change] This line is called the commit's *title*. Titles must be:

- one line
- less than 72 characters long
- followed by a completely blank line

[area]

The `[area]` prefix usually identifies the area of code being changed. It can also identify the change's wider context if multiple areas are affected.

Here are some examples:

- `doc:` ... for documentation changes

- `drivers: foo:` for foo driver changes
- `Bluetooth: Shell:` for changes to the Bluetooth shell
- `net: ethernet:` for Ethernet-related networking changes
- `dts:` for treewide devicetree changes
- `style:` for code style changes

If you're not sure what to use, try running `git log FILE`, where `FILE` is a file you are changing, and using previous commits that changed the same file as inspiration.

[summary of change]
The `[summary of change]` part should be a quick description of what you've done. Here are some examples:

- `doc: update wiki references to new site`
- `drivers: sensor: sensor_shell: fix channel name collision`

Warning

An empty commit message body is not permitted. Even for trivial changes, please include a descriptive commit message body. Your pull request will fail CI checks if you do not.

Commit Message Body This part of the commit should explain what your change does, and why it's needed. Be specific. A body that says `"Fixes stuff"` will be rejected. Be sure to include the following as relevant:

- **what** the change does,
- **why** you chose that approach,
- **what** assumptions were made, and
- **how** you know it works – for example, which tests you ran.

Each line in your commit message should usually be 75 characters or less. Use newlines to wrap longer lines. Exceptions include lines with long URLs, email addresses, etc.

For examples of accepted commit messages, you can refer to the Zephyr GitHub changelog.

Tip

You should have set your *Git Setup* already. Create your commit with `git commit -s` to add the Signed-off-by: line automatically using this information.

Signed-off-by: ... For open source licensing reasons, your commit must include a Signed-off-by: line that looks like this:

```
Signed-off-by: [Your Full Name] <[your.email@address]>
```

For example, if your full name is `Zephyr Developer` and your email address is `z.developer@example.com`:

```
Signed-off-by: Zephyr Developer <z.developer@example.com>
```

This means that you have personally made sure your change complies with the *Developer Certification of Origin (DCO)*. For this reason, you must use your legal name. Pseudonyms or "hacker aliases" are not permitted.

Your name and the email address you use must match the name and email in the Git commit's `Author:` field.

See the *Contributor Expectations* for a more complete discussion of contributor and reviewer expectations.

Adding Links If your change addresses a specific GitHub issue, include a reference in the pull request description using the following format:

```
Fixes zephyrproject-rtos/zephyr#[issue number]
```

For pull requests to the Zephyr project only, the short form can also be used, for example:

```
Fixes #[issue number]
```

Replace [issue number] with the relevant GitHub issue number. For example:

```
Fixes zephyrproject-rtos/zephyr#1234
```

This syntax ensures that the issue is automatically closed when the pull request is merged. Always specify the full repository path (zephyrproject-rtos/zephyr) to avoid ambiguity, especially when working across multiple repositories.

The same format can also be used in commit messages.

For linking to additional external resources—such as related issues, datasheets, or technical reference manuals—use the `Link:` tag:

```
Link: https://github.com/zephyrproject-rtos/zephyr/issues/<issue number>
```

Continuous Integration (CI) The Zephyr Project operates a Continuous Integration (CI) system that runs on every Pull Request (PR) in order to verify several aspects of the PR:

- Git commit formatting
- Coding Style
- Twister builds for multiple architectures and boards
- Documentation build to verify any doc changes

CI is run on Github Actions and it uses the same tools described in the *CI Tests* section. The CI results must be green indicating "All checks have passed" before the Pull Request can be merged. CI is run when the PR is created, and again every time the PR is modified with a commit.

The current status of the CI run can always be found at the bottom of the GitHub PR page, below the review status. Depending on the success or failure of the run you will see:

- "All checks have passed"
- "All checks have failed"

In case of failure you can click on the "Details" link presented below the failure message in order to navigate to `Github Actions` and inspect the results. Once you click on the link you will be taken to the `Github actions` summary results page where a table with all the different builds will be shown. To see what build or test failed click on the row that contains the failed (i.e. non-green) build.

Running CI Tests Locally

check_compliance.py The scripts/ci/check_compliance.py script serves as a valuable tool for assessing code compliance with Zephyr's established guidelines and best practices. The script acts as wrapper for a suite of tools that performs various checks, including linters and formatters.

Developers are encouraged to run the script locally to validate their changes before opening a new Pull Request:

```
./scripts/ci/check_compliance.py -c <commit range>
```

The checks run in parallel, using one worker process per CPU. Pass `-p N` to limit the number of worker processes to `N`, or `-p 1` to run the checks sequentially.

```
./scripts/ci/check_compliance.py -p 1 -c <commit range>
```

> **Note**
>
> On Windows if the .pl extension has not yet been associated with an application, then the first time a .pl file is run without specifying an interpreter, Windows will ask what application will open Perl files. Set the default app to Strawberry Perl. By default the executable is installed at `C:\Strawberry\perl\bin\perl.exe`.

KeepSorted Check The KeepSorted check ensures that specified blocks of code, configuration, or documentation remain in sorted order.

To use the KeepSorted check, wrap your sorted content between dedicated lines containing start and stop markers, typically using comments:

```
// zephyr-keep-sorted-start
option_a
option_b
option_c
// zephyr-keep-sorted-stop
```

KeepSorted Marker Options The sorting behavior of each block can be customized in several ways. To do this, one or more of the following parameters can be added on the same line as the start marker itself:

re(regex_pattern)

Enables regex mode where only lines matching the specified regular expression are checked for sorting. Other lines are ignored.

Example checking sorted properties in a yaml file:

```yaml
# zephyr-keep-sorted-start re(^\s+\- name:)
projects:
  - name: application
    revision: main
  - name: library1
    revision: feature-branch
  - name: library2
    revision: main
# zephyr-keep-sorted-stop
```

strip(characters)

Strips the specified characters from lines before performing the sort comparison. This is useful when lines have optional prefixes or suffixes that should be ignored during sorting.

Example stripping quotes from yaml dictionary keys:

```
# zephyr-keep-sorted-start strip(":)
ACPI:
  status: odd fixes
"West project: acpica":
  status: odd fixes
# zephyr-keep-sorted-stop
```

nofold
: Disables line folding. By default, indented lines following a main line are concatenated (folded) together for sorting comparison. The `nofold` option disables this behavior and ignores indented lines.

ignorecase
: Enables case-insensitive sorting using Python's str.casefold. This allows mixing uppercase and lowercase items in sorted blocks without causing sort order violations. Defaults to Python's string ordering if omitted.

Multiple options can be combined on the same marker line:

```
.. zephyr-keep-sorted-start re(^\* \w) ignorecase
* Shell
  Some important message about the shell.

* STM32
  Updates for this vendor.
.. zephyr-keep-sorted-stop
```

> **Note**
>
> twister is only fully supported on Linux; on Windows and MacOS the execution of tests is not supported on all target devices.

twister If you think your change may break some test, you can submit your PR as a draft and let the project CI automatically run the *Test Runner (Twister)* for you.

If a test fails, you can check from the CI run logs how to rerun it locally, for example:

```
west twister -p native_sim -s␣
↪tests/drivers/build_all/sensor/drivers.sensor.generic_test
```

Static Code Analysis

Coverity Scan is a free service for static code analysis of Open Source projects. It is based on Coverity's commercial product and is able to analyze C, C++ and Java code.

Coverity's static code analysis doesn't run the code. Instead of that it uses abstract interpretation to gain information about the code's control flow and data flow. It's able to follow all possible code paths that a program may take. For example the analyzer understands that malloc() returns a memory that must be freed with free() later. It follows all branches and function calls to see if all possible combinations free the memory. The analyzer is able to detect all sorts of issues like resource leaks (memory, file descriptors), NULL dereferencing, use after free, unchecked return values, dead code, buffer overflows, integer overflows, uninitialized variables, and many more.

The results are available on the Coverity Scan website. In order to access the results you have to create an account yourself. From the Zephyr project page, you may select "Add me to project" to be added to the project. New members must be approved by an admin.

Static analysis of the Zephyr codebase is conducted on a bi-weekly basis. GitHub issues are automatically created for any issues detected by static analysis tools. These issues will have the same (or equivalent) priority initially defined by the tool.

To ensure accountability and efficient issue resolution, they are assigned to the respective maintainer who is responsible for the affected code.

A dedicated team comprising members with expertise in static analysis, code quality, and software security ensures the effectiveness of the static analysis process and verifies that identified issues are properly triaged and resolved in a timely manner.

Workflow If after analyzing the Coverity report it is concluded that it is a false positive please set the classification to either "False positive" or "Intentional", the action to "Ignore", owner to your own account and add a comment why the issue is considered false positive or intentional.

Update the related Github issue in the zephyr project with the details, and only close it after completing the steps above on scan service website. Any issues closed without a fix or without ignoring the entry in the scan service will be automatically reopened if the issue continues to be present in the code.

Contribution Workflow

One general practice we encourage, is to make small, controlled changes. This practice simplifies review, makes merging and rebasing easier, and keeps the change history clear and clean.

When contributing to the Zephyr Project, it is also important you provide as much information as you can about your change, update appropriate documentation, and test your changes thoroughly before submitting.

The general GitHub workflow used by Zephyr developers uses a combination of command line Git commands and browser interaction with GitHub. As it is with Git, there are multiple ways of getting a task done. We'll describe a typical workflow here:

1. Create a Fork of Zephyr to your personal account on GitHub. (Click on the fork button in the top right corner of the Zephyr project repo page in GitHub.)
2. On your development computer, change into the `zephyr` folder that was created when you *obtained the code*:

   ```
   cd zephyrproject/zephyr
   ```

   Rename the default remote pointing to the upstream repository from `origin` to `upstream`:

   ```
   git remote rename origin upstream
   ```

   Let Git know about the fork you just created, naming it `origin`:

   ```
   git remote add origin https://github.com/<your github id>/zephyr
   ```

   and verify the remote repos:

   ```
   git remote -v
   ```

   The output should look similar to:

   ```
   origin   https://github.com/<your github id>/zephyr (fetch)
   origin   https://github.com/<your github id>/zephyr (push)
   upstream https://github.com/zephyrproject-rtos/zephyr (fetch)
   upstream https://github.com/zephyrproject-rtos/zephyr (push)
   ```

3. Create a topic branch (off of `main`) for your work (if you're addressing an issue, we suggest including the issue number in the branch name):

```
git switch main
git switch -c fix_comment_typo
```

Some Zephyr subsystems do development work on a separate branch from `main` so you may need to indicate this in your checkout:

```
git switch -c fix_out_of_date_patch origin/net
```

4. Make changes, test locally, change, test, test again, ... (Check out the prior chapter on *twister* as well).
5. When things look good, start the pull request process by adding your changed files:

```
git add [file(s) that changed, add -p if you want to be more specific]
```

You can see files that are not yet staged using:

```
git status
```

6. Verify changes to be committed look as you expected:

```
git diff --cached
```

7. Commit your changes to your local repo:

```
git commit -s
```

The `-s` option automatically adds your `Signed-off-by:` to your commit message. Your commit will be rejected without this line that indicates your agreement with the *Developer Certification of Origin (DCO)*. See the *Commit Message Guidelines* section for specific guidelines for writing your commit messages.

8. Push your topic branch with your changes to your fork in your personal GitHub account:

```
git push origin fix_comment_typo
```

9. In your web browser, go to your forked repo and click on the `Compare & pull request` button for the branch you just worked on and you want to open a pull request with.
10. Review the pull request changes, and verify that you are opening a pull request for the `main` branch. The title and message from your commit message should appear as well.
11. A bot will assign one or more suggested reviewers (based on the MAINTAINERS file in the repo). If you are a project member, you can select additional reviewers now too.
12. Click on the submit button and your pull request is sent and awaits review. Email will be sent as review comments are made, or you can check on your pull request at https://github.com/zephyrproject-rtos/zephyr/pulls.
13. While you're waiting for your pull request to be accepted and merged, you can create another branch to work on another issue. (Be sure to make your new branch off of `main` and not the previous branch.):

```
git switch main
git switch -c fix_another_issue
```

and use the same process described above to work on this new topic branch.

14. If reviewers do request changes to your patch, you can interactively rebase commit(s) to fix review issues. In your development repo:

```
git rebase -i <offending-commit-id>^
```

In the interactive rebase editor, replace `pick` with `edit` to select a specific commit (if there's more than one in your pull request), or remove the line to delete a commit entirely. Then edit files to fix the issues in the review.

As before, inspect and test your changes. When ready, continue the patch submission:

```
git add [file(s)]
git rebase --continue
```

Update commit comment if needed, and continue:

```
git push --force origin fix_comment_typo
```

By force pushing your update, your original pull request will be updated with your changes so you won't need to resubmit the pull request.

15. After pushing the requested change, check on the PR page if there is a merge conflict. If so, rebase your local branch:

```
git fetch --all
git rebase --ignore-whitespace upstream/main
```

The `--ignore-whitespace` option stops `git apply` (called by rebase) from changing any whitespace. Resolve the conflicts and push again:

```
git push --force origin fix_comment_typo
```

> **Note**
>
> While amending commits and force pushing is a common review model outside GitHub, and the one recommended by Zephyr, it's not the main model supported by GitHub. Forced pushes can cause unexpected behavior, such as not being able to use "View Changes" buttons except for the last one - GitHub complains it can't find older commits. You're also not always able to compare the latest reviewed version with the latest submitted version. When rewriting history GitHub only guarantees access to the latest version.

16. If the CI run fails, you will need to make changes to your code in order to fix the issues and amend your commits by rebasing as described above. Additional information about the CI system can be found in *Continuous Integration*.

Contribution Tips The following is a list of tips to improve and accelerate the review process of Pull Requests. If you follow them, chances are your pull request will get the attention needed and it will be ready for merge sooner than later:

1. When pushing follow-up changes, use the `--keep-base` option of git-rebase
2. On the PR page, check if the change can still be merged with no merge conflicts
3. Make sure title of PR explains what is being fixed or added
4. Make sure your PR has a body with more details about the content of your submission
5. Make sure you reference the issue you are fixing in the body of the PR
6. Watch early CI results immediately after submissions and fix issues as they are discovered
7. Revisit PR after 1-2 hours to see the status of all CI checks, make sure all is green
8. If you get request for changes and submit a change to address them, make sure you click the "Re-request review" button on the GitHub UI to notify those who asked for the changes

Identifying Contribution Origin When adding a new file to the tree, it is important to detail the source of origin on the file, provide attributions, and detail the intended usage. In cases where the file is an original to Zephyr, the commit message should include the following ("Original" is the assumption if no Origin tag is present):

```
Origin: Original
```

In cases where the file is *imported from an external project*, the commit message shall contain details regarding the original project, the location of the project, the SHA-id of the origin commit for the file and the intended purpose.

For example, a copy of a locally maintained import:

```
Origin: Contiki OS
License: BSD 3-Clause
URL: https://www.contiki-os.org/
commit: 853207acfdc6549b10eb3e44504b1a75ae1ad63a
Purpose: Introduction of networking stack.
```

For example, a copy of an externally maintained import in a module repository:

```
Origin: Tiny Crypt
License: BSD 3-Clause
URL: https://github.com/01org/tinycrypt
commit: 08ded7f21529c39e5133688ffb93a9d0c94e5c6e
Purpose: Introduction of TinyCrypt
```

Contributions to External Modules

Follow the guidelines in the *Modules (External projects)* section for contributing *new modules* and submitting changes to *existing modules*.

Treewide Changes

This section describes contributions that are treewide changes and some additional associated requirements that apply to them. These requirements exist to try to give such changes increased review and user visibility due to their large impact.

Definition and Decision Making A *treewide change* is defined as any change to Zephyr APIs, coding practices, or other development requirements that either implies required changes throughout the zephyr source code repository or can reasonably be expected to do so for a wide class of external Zephyr-based source code.

This definition is informal by necessity. This is because the decision on whether any particular change is treewide can be subjective and may depend on additional context.

Project maintainers should use good judgement and prioritize the Zephyr developer experience when deciding when a proposed change is treewide. Protracted disagreements can be resolved by the Zephyr Project's Technical Steering Committee (TSC), but please avoid premature escalation to the TSC.

Requirements for Treewide Changes

- The zephyr repository must apply the 'treewide' GitHub label to any issues or pull requests that are treewide changes
- The person proposing a treewide change must create an RFC issue describing the change, its rationale and impact, etc. before any pull requests related to the change can be merged

- The project's Architecture Working Group (WG) must include the issue on the agenda and discuss whether the project will accept or reject the change before any pull requests related to the change can be merged (with escalation to the TSC if consensus is not reached at the WG)
- The Architecture WG must specify the procedure for merging any PRs associated with each individual treewide change, including any required approvals for pull requests affecting specific subsystems or extra review time requirements
- The person proposing a treewide change must email devel@lists.zephyrproject.org about the RFC if it is accepted by the Architecture WG before any pull requests related to the change can be merged

Examples Some example past treewide changes are:

- the deprecation of version 1 of the *Logging API* in favor of version 2 (see commit 262cc55609)
- the removal of support for a legacy *Devicetree bindings* syntax (6bf761fc0a)

Note that adding a new version of a widely used API while maintaining support for the old one is not a treewide change. Deprecation and removal of such APIs, however, are treewide changes.

Specialized driver requirements

Drivers for standalone devices should use the Zephyr bus APIs (SPI, I2C...) whenever possible so that the device can be used with any SoC from any vendor implementing a compatible bus.

If it is not technically possible to achieve full performance using the Zephyr APIs due to specialized accelerators in a particular SoC family, one could extend the support for an external device by providing a specialized path for that SoC family. However, the driver must still provide a regular path (via Zephyr APIs) for all other SoCs. Every exception must be approved by the Architecture WG in order to be validated and potentially to be learned/improved from.

7.1.2 Contributor Expectations

The Zephyr project encourages *contributors* to submit changes as smaller pull requests. Smaller pull requests (PRs) have the following benefits:

- Reviewed more quickly and reviewed more thoroughly. It's easier for reviewers to set aside a few minutes to review smaller changes several times than it is to allocate large blocks of time to review a large PR.
- Less wasted work if reviewers or maintainers reject the direction of the change.
- Easier to rebase and merge. Smaller PRs are less likely to conflict with other changes in the tree.
- Easier to revert if the PR breaks functionality.

> **Note**
>
> This page does not apply to draft PRs which can have any size, any number of commits and any combination of smaller PRs for testing and preview purposes. Draft PRs have no review expectation and PRs created as drafts from the start do not notify anyone by default.

Defining Smaller PRs

- Smaller PRs should encompass one self-contained logical change.

- When adding a new large feature or API, the PR should address only one part of the feature. In this case create an *RFC proposal* to describe the additional parts of the feature for reviewers.
- PRs should include tests or samples under the following conditions:
  - Adding new features or functionality.
  - Modifying a feature, especially for API behavior contract changes.
  - Fixing a hardware agnostic bug. The test should fail without the bug fixed and pass with the fix applied.
- PRs must update any documentation affected by a functional code change.
- If introducing a new API, the PR must include an example usage of the API. This provides context to the reviewer and prevents submitting PRs with unused APIs.

Multiple Commits on a Single PR

Contributors are further encouraged to break up PRs into multiple commits. Keep in mind each commit in the PR must still build cleanly and pass all the CI tests.

For example, when introducing an extension to an API, contributors can break up the PR into multiple commits targeting these specific changes:

1. Introduce the new APIs, including shared devicetree bindings
2. Update driver implementation X, with driver specific devicetree bindings
3. Update driver implementation Y
4. Add tests for the new API
5. Add a sample using the API
6. Update the documentation

Large Changes

Large changes to the Zephyr project must submit an *RFC proposal* describing the full scope of change and future work. The RFC proposal provides the required context to reviewers, but allows for smaller, incremental, PRs to get reviewed and merged into the project. The RFC should also define the minimum viable implementation.

Changes which require an RFC proposal include:

- Submitting a new feature.
- Submitting a new API.
- *Treewide Changes*.
- Other large changes that can benefit from the RFC proposal process.

Maintainers have the discretion to request that contributors create an RFC for PRs that are too large or complicated.

PR Requirements

Important

Pull requests that do not meet the quality expectations outlined below are less likely to receive reviewer engagement, and reviewers may request changes on such pull requests without detailed feedback.

> Contributors are expected to self-review their changes and ensure all requirements are met before requesting a review.

- Each commit in the PR must provide a commit message following the *Commit Message Guidelines*.
- No fixup or merge commits are allowed, see *Contribution Workflow* for more information.
- The PR description must include a summary of the changes and their rationales.
- All files in the PR must comply with *Licensing Requirements*.
- The code must follow the Zephyr *Coding Style Guidelines* and *Coding Guidelines*.
- The PR must pass all CI checks, as described in *Pull Request Merge Criteria*. Contributors may mark a PR as draft and explicitly request reviewers to provide early feedback, even with failing CI checks.
- Commits in a pull request should represent clear, logical units of change that are easy to review and maintain bisectability. The following guidelines expand on this principle:
  1. Distinct, Logical Units of Change

     Each commit should correspond to a self-contained, meaningful change. For example, adding a feature, fixing a bug, or refactoring existing code should be separate commits. Avoid mixing different types of changes (e.g., feature implementation and unrelated refactoring) in the same commit.
  2. Retain Bisectability

     Every commit in the pull request must build successfully and pass all relevant tests. This ensures that git bisect can be used effectively to identify the specific commit that introduced a bug or issue.
  3. Squash Intermediary or Non-Final Development History

     During development, commits may include intermediary changes (e.g., partial implementations, temporary files, or debugging code). These should be squashed or rewritten before submitting the pull request. Remove non-final artifacts, such as:
     - Temporary renaming of files that are later renamed again.
     - Code that was rewritten or significantly changed in later commits.
  4. Ensure Clean History Before Submission

     Use interactive rebasing (git rebase -i) to clean up the commit history before submitting the pull request. This helps in:
     - Squashing small, incomplete commits into a single cohesive commit.
     - Ensuring that each commit remains bisectable.
     - Maintaining proper attribution of authorship while improving clarity.
  5. Renaming and Code Rewrites

     If files or code are renamed or rewritten in later commits during development, squash or rewrite earlier commits to reflect the final structure. This ensures that:
     - The history remains clean and easy to follow.
     - Bisectability is preserved by eliminating redundant renaming or partial rewrites.
  6. Attribution of Authorship

     While cleaning up the commit history, ensure that authorship attribution remains accurate.

7. Readable and Reviewable History

   The final commit history should be easy to understand for future maintainers. Logical units of change should be grouped into commits that tell a clear, coherent story of the work done.

- When major new functionality is added, tests for the new functionality shall be added to the automated test suite. All API functions should have test cases and there should be tests for the behavior contracts of the API. Maintainers and reviewers have the discretion to determine if the provided tests are sufficient. The examples below demonstrate best practices on how to test APIs effectively.
  - Kernel timer tests provide around 85% test coverage for the kernel timer, measured by lines of code.
  - Emulators for off-chip peripherals are an effective way to test driver APIs. The fuel gauge tests use the smart battery emulator, providing test coverage for the fuel gauge API and the smart battery driver.
  - Code coverage reports for the Zephyr project are available on Codecov.
- Incompatible changes to APIs must also update the release notes for the next release detailing the change. APIs marked as experimental are excluded from this requirement.
- Changes to APIs must increment the API version number according to the API version rules.
- Documentation must be added and/or updated to reflect the changes in the code introduced by the PR. The documentation changes must use the proper terminology as present in the existing pages, and must be written in American English. If you include images as part of the documentation, those must follow the rules in *Images*. Please refer to *Documentation Guidelines* for additional information.
- PRs must also satisfy all *Pull Request Merge Criteria* before a member of the release engineering team merges the PR into the zephyr tree.

Maintainers may request that contributors break up a PR into smaller PRs and may request that they create an *RFC proposal*.

Workflow Suggestions That Help Reviewers

- Unless they applied the reviewer's recommendation exactly, authors must not resolve and hide comments, they must let the initial reviewer do it. The Zephyr project does not require all comments to be resolved before merge. Leaving some completed discussions open can sometimes be useful to understand the greater picture.
- Respond to comments using the "Start Review" and "Add Review" green buttons in the "Files changed" view. This allows responding to multiple comments and publishing the responses in bulk. This reduces the number of emails sent to reviewers.
- As GitHub does not implement `git range-diff`, try to minimize rebases in the middle of a review. If a rebase is required, push this as a separate update with no other changes since the last push of the PR. When pushing a rebase only, add a comment to the PR indicating which commit is the rebase.

Getting PRs Reviewed The Zephyr community is a diverse group of individuals, with different levels of commitment and priorities. As such, reviewers and maintainers may not get to a PR right away.

- After 1 week of inactivity, ping the Assignee or reviewers on the PR by adding a comment to the PR.
- After 2 weeks of inactivity, post a message on the #pr-help channel on Discord linking to the PR.

PR Technical Escalation In cases where a contributor objects to change requests from reviewers, Zephyr defines the following escalation process for resolving technical disagreements.

Before escalation of technical disagreements, follow the steps below:

- Resolve in the PR among assignee, maintainers and reviewer.
  - Assignee to act as moderator if applicable.
- If no progress is made, the assignee (maintainer) has the right to dismiss stale, unrelated or irrelevant change requests by reviewers giving the reviewers a minimum of 1 business day to respond and revisit their initial change requests or start the escalation process.

  The assignee has the responsibility to document the reasoning for dismissing any reviews in the PR and should notify the reviewer about their review being dismissed.

  To give the reviewers time to respond and escalate, the assignee is expected to block the PR from being merged either by not approving the PR or by setting the *DNM* label.

Escalation can be triggered by any party participating in the review process (assignee, reviewers or the original author of the change) following the steps below:

- Escalate to the Architecture Working Group by adding the *Architecture Review* label on the PR. Beside the weekly meeting where such escalations are processed, the Architecture Working Group shall facilitate an offline review of the escalation if requested, especially if any of the parties can't attend the meeting.
- If all avenues of resolution and escalation have failed, assignees can escalate to the TSC and get a binding resolution in the TSC by adding the *TSC* label on the PR.
- The Assignee is expected to ensure the resolution of the escalation and the outcome is documented in the related pull request or issues on Github.

7.1.3 Reviewer Expectations

- Be respectful when commenting on PRs. Refer to the Zephyr Code of Conduct for more details.
- The Zephyr Project recognizes that reviewers and maintainers have limited bandwidth. As a reviewer, prioritize review requests in the following order:
  1. PRs related to items in the Zephyr Release Plan or those targeting the next release during the stabilization period (after RC1).
  2. PRs where the reviewer has requested blocking changes.
  3. PRs assigned to the reviewer as the area maintainer.
  4. All other PRs.
- Reviewers shall strive to advance the PR to a mergeable state with their feedback and engagement with the PR author.
- Try to provide feedback on the entire PR in one shot. This provides the contributor an opportunity to address all comments in the next PR update.
- Partial reviews are permitted, but the reviewer must add a comment indicating what portion of the PR they reviewed. Examples of useful partial reviews include:
  - Domain specific reviews (e.g. Devicetree).
  - Code style changes that impact the readability of the PR.
  - Reviewing commits separately when the requested changes cascade into the later commits.
- Avoid increasing scope of the PR by requesting new features, especially when there is a corresponding *RFC* associated with the PR. Instead, reviewers should add suggestions as a

comment to the *RFC*. This also encourages more collaboration as it is easier for multiple contributors to work on a feature once the minimum implementation has merged.

- When using the "Request Changes" option, mark trivial, non-functional, requests as "Non-blocking" in the comment. Reviewers should approve PRs once only non-blocking changes remain. The PR author has discretion as to whether they address all non-blocking comments. PR authors should acknowledge every review comment in some way, even if it's just with an emoticon.
- Style changes that the reviewer disagrees with but that are not documented as part of the project can be pointed out as non-blocking, but cannot constitute a reason for a request for changes. The reviewer can optionally correct any potential inconsistencies in the tree, document the new guidelines or rules, and then enforce them as part of the review.
- Whenever requesting style related changes, reviewers should be able to point out the corresponding guideline, rule or rationale in the project's documentation. This does not apply to certain types of requests for changes, notably those specific to the changes being submitted (e.g. the use of a particular data structure or the choice of locking primitives).
- Reviewers shall be *clear* about what changes they are requesting when the "Request Changes" option is used. Requested changes shall be in the scope of the PR in question and following the contribution and style guidelines of the project. Furthermore, reviewers must be able to point back to the exact issues in the PR that triggered a request for changes.
- Reviewers should not request changes for issues which are automatically caught by CI, as this causes the pull request to remain blocked even after CI failures have been addressed and may unnecessarily delay it from being merged.
- Reviewers shall not close a PR due to technical or structural disagreement. If requested changes cannot be resolved within the review process, the *PR Technical Escalation* path shall be used for any potential resolution path, which may include closing the PR.
- When using AI tools to assist in reviewing code or drafting responses:
  - Reviewers are responsible for following up on comments generated by the AI tools they may be using (e.g. GitHub Copilot).
  - Reviewers must never paste raw, unverified LLM output as a PR comment.
  - In general: all AI-generated feedback must be vetted, verified, and contextualized by a human reviewer who assumes full accountability for the accuracy and tone of the comment or review.

7.1.4 Coding Guidelines

Main rules

The coding guideline rules are based on MISRA-C 2012 and are a **subset** of MISRA-C. The subset is listed in the table below with a summary of the rules, its MISRA-C severity and the equivalent rules from other standards for reference.

The severity and other references in the table below are for informational purposes only. The listed rules are all required for Zephyr and all new code should comply with the rules listed below.

> **Note**
>
> For existing Zephyr maintainers and collaborators, if you are unable to obtain a copy through your employer, a limited number of copies will be made available through the project. If you need a copy of MISRA-C 2012, please send email to safety@lists.zephyrproject.org and provide details on reason why you can't obtain one through other options and expected contributions once you have one. The safety committee will review all requests.

Table 1: Main rules

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood	Dir 1.1	FLP30-C MSC09-C EXP11-C
2	All source files shall compile without any compilation errors	Dir 2.1	N/A
3	All code shall be traceable to documented requirements	Dir 3.1	N/A
4	Run-time failures shall be minimized	Dir 4.1	N/A
5	All usage of assembly language should be documented	Dir 4.2	N/A
6	Sections of code should not be "commented out"	Dir 4.4	MSC04-C
7	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	Dir 4.5	DCL02-C
8	typedefs that indicate size and signedness should be used in place of the basic numerical types	Dir 4.6	N/A
9	If a function returns error information, then that error information shall be tested	Dir 4.7	N/A
10	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	Dir 4.8 example 1 Dir 4.8 example 2	DCL12-C
11	A function should be used in preference to a function-like macro where they are interchangeable	Dir 4.9	PRE00-C
12	Precautions shall be taken in order to prevent the contents of a header file being included more than once	Dir 4.10	PRE06-C
13	The validity of values passed to library functions shall be checked	Dir 4.11	N/A
14	Dynamic memory allocation shall not be used	Dir 4.12	API03-C API04-C STR01-C
15	Functions which are designed to provide operations on a resource should be called in an appropriate sequence	Dir 4.13	N/A
16	The validity of values received from external sources shall be checked	Dir 4.14	N/A
17	Language extensions should not be used	Rule 1.2	MSC04-C
18	There shall be no occurrence of undefined or critical unspecified behaviour	Rule 1.3	N/A
19	A project shall not contain unreachable code	Rule 2.1 example 1 Rule 2.1 example 2	MSC12-C

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
20	There shall be no dead code	Rule 2.2	DCL22-C MSC12-C
21	A project should not contain unused type declarations	Rule 2.3	N/A
22	A function should not contain unused label declarations	Rule 2.6	N/A
23	There should be no unused parameters in functions	Rule 2.7	N/A
24	The character sequences /* and // shall not be used within a comment	Rule 3.1	MSC04-C
25	Line-splicing shall not be used in // comments	Rule 3.2	N/A
26	Octal and hexadecimal escape sequences shall be terminated	Rule 4.1	MSC09-C
27	Trigraphs should not be used	Rule 4.2	PRE07-C
28	External identifiers shall be distinct	Rule 5.1 example 1 Rule 5.1 example 2	DCL23-C
29	Identifiers declared in the same scope and name space shall be distinct	Rule 5.2	DCL23-C
30	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	Rule 5.3	DCL01-C DCL23-C
31	Macro identifiers shall be distinct	Rule 5.4	DCL23-C
32	Identifiers shall be distinct from macro names	Rule 5.5	DCL23-C
33	A typedef name shall be a unique identifier	Rule 5.6	N/A
34	A tag name shall be a unique identifier	Rule 5.7	N/A
35	Identifiers that define objects or functions with external linkage shall be unique	Rule 5.8 example 1 Rule 5.8 example 2	N/A
36	Identifiers that define objects or functions with internal linkage should be unique	Rule 5.9 example 1 Rule 5.9 example 2	N/A
37	Bit-fields shall only be declared with an appropriate type	Rule 6.1	INT14-C
38	Single-bit named bit fields shall not be of a signed type	Rule 6.2	INT14-C
39	Octal constants shall not be used	Rule 7.1	DCL18-C
40	A u or U suffix shall be applied to all integer constants that are represented in an unsigned type	Rule 7.2	N/A
41	The lowercase character l shall not be used in a literal suffix	Rule 7.3	DCL16-C
42	A string literal shall not be assigned to an object unless the objects type is pointer to const-qualified char	Rule 7.4	N/A
43	Types shall be explicitly specified	Rule 8.1	DCL31-C

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
44	Function types shall be in prototype form with named parameters	Rule 8.2	DCL07-C DCL20-C DCL36-C EXP37-C
45	All declarations of an object or function shall use the same names and type qualifiers	Rule 8.3	N/A
46	A compatible declaration shall be visible when an object or function with external linkage is defined	Rule 8.4	DCL36-C DCL40-C
47	An external object or function shall be declared once in one and only one file	Rule 8.5 example 1 Rule 8.5 example 2	N/A
48	An identifier with external linkage shall have exactly one external definition	Rule 8.6 example 1 Rule 8.6 example 2	N/A
49	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	Rule 8.8	DCL15-C DCL36-C
50	An object should be defined at block scope if its identifier only appears in a single function	Rule 8.9	DCL19-C
51	An inline function shall be declared with the static storage class	Rule 8.10	N/A
52	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique	Rule 8.12	INT09-C
53	The restrict type qualifier shall not be used	Rule 8.14	EXP43-C
54	The value of an object with automatic storage duration shall not be read before it has been set	Rule 9.1	N/A
55	The initializer for an aggregate or union shall be enclosed in braces	Rule 9.2	N/A
56	Arrays shall not be partially initialized	Rule 9.3	N/A
57	An element of an object shall not be initialized more than once	Rule 9.4	N/A
58	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly	Rule 9.5	ARR02-C
59	Operands shall not be of an inappropriate essential type	Rule 10.1	INT02-C INT07-C INT12-C INT31-C STR04-C STR34-C
60	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations	Rule 10.2	STR04-C STR34-C

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
61	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category	Rule 10.3	INT02-C INT07-C INT31-C STR04-C STR34-C
62	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category	Rule 10.4	INT02-C INT07-C INT31-C STR04-C STR34-C
63	The value of an expression should not be cast to an inappropriate essential type	Rule 10.5	EXP14-C INT02-C
64	The value of a composite expression shall not be assigned to an object with wider essential type	Rule 10.6	INT02-C INT31-C
65	If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type	Rule 10.7	INT02-C INT31-C
66	The value of a composite expression shall not be cast to a different essential type category or a wider essential type	Rule 10.8	INT02-C
67	Conversions shall not be performed between a pointer to an incomplete type and any other type	Rule 11.2	EXP36-C
68	A cast shall not be performed between pointer to void and an arithmetic type	Rule 11.6	N/A
69	A cast shall not be performed between pointer to object and a noninteger arithmetic type	Rule 11.7	EXP36-C
70	A cast shall not remove any const or volatile qualification from the type pointed to by a pointer	Rule 11.8	EXP05-C EXP32-C
71	The macro NULL shall be the only permitted form of integer null pointer constant	Rule 11.9	N/A
72	The precedence of operators within expressions should be made explicit	Rule 12.1	EXP00-C
73	The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand	Rule 12.2	N/A
74	Evaluation of constant expressions should not lead to unsigned integer wrap-around	Rule 12.4	N/A
75	The sizeof operator shall not have an operand which is a function parameter declared as “array of type”	Rule 12.5	N/A
76	Initializer lists shall not contain persistent side effects	Rule 13.1 example 1 Rule 13.1 example 2	N/A

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
77	The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders	Rule 13.2	EXP30-C
78	A full expression containing an increment (++) or decrement (–) operator should have no other potential side effects other than that caused by the increment or decrement operator	Rule 13.3	N/A
79	The result of an assignment operator should not be used	Rule 13.4	N/A
80	The right hand operand of a logical && or || operator shall not contain persistent side effects	Rule 13.5 example 1 Rule 13.5 example 2	EXP10-C
81	The operand of the sizeof operator shall not contain any expression which has potential side effects	Rule 13.6	N/A
82	A loop counter shall not have essentially floating type	Rule 14.1	FLP30-C
83	A for loop shall be well-formed	Rule 14.2	N/A
84	Controlling expressions shall not be invariant	Rule 14.3	N/A
85	The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type	Rule 14.4	N/A
86	The goto statement shall jump to a label declared later in the same function	Rule 15.2	N/A
87	Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement	Rule 15.3	N/A
88	The body of an iteration-statement or a selection-statement shall be a compound-statement	Rule 15.6	EXP19-C
89	All if else if constructs shall be terminated with an else statement	Rule 15.7	MSC01-C
90	All switch statements shall be well-formed	Rule 16.1	DCL41-C
91	A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement	Rule 16.2	MSC20-C
92	An unconditional break statement shall terminate every switch-clause	Rule 16.3	N/A
93	Every switch statement shall have a default label	Rule 16.4	N/A
94	A default label shall appear as either the first or the last switch label of a switch statement	Rule 16.5	N/A
95	Every switch statement shall have at least two switch-clauses	Rule 16.6	N/A
96	A switch-expression shall not have essentially Boolean type	Rule 16.7	N/A
97	The features of <stdarg.h> shall not be used	Rule 17.1	DCL10-C DCL11-C ERR00-C
98	Functions shall not call themselves, either directly or indirectly	Rule 17.2	MEM05-C

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
99	A function shall not be declared implicitly	Rule 17.3	DCL36-C EXP37-C
100	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	Rule 17.4	N/A
101	The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements	Rule 17.5	N/A
102	The declaration of an array parameter shall not contain the static keyword between the [ ]	Rule 17.6	N/A
103	The value returned by a function having non-void return type shall be used	Rule 17.7	N/A
104	A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand	Rule 18.1	ARR30-C ARR39-C EXP08-C
105	Subtraction between pointers shall only be applied to pointers that address elements of the same array	Rule 18.2	ARR39-C EXP08-C
106	The relational operators >, >=, < and <= shall not be applied to objects of pointer type except where they point into the same object	Rule 18.3	ARR39-C EXP08-C
107	Declarations should contain no more than two levels of pointer nesting	Rule 18.5	N/A
108	The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist	Rule 18.6 example 1 Rule 18.6 example 2	DCL30-C MEM30-C
109	Variable-length array types shall not be used	Rule 18.8	N/A
110	An object shall not be assigned or copied to an overlapping object	Rule 19.1	N/A
111	The ‘, or characters and the /* or // character sequences shall not occur in a header file name”	Rule 20.2	N/A
112	The #include directive shall be followed by either a <filename> or “filename” sequence	Rule 20.3	N/A
113	A macro shall not be defined with the same name as a keyword	Rule 20.4	N/A
114	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	Rule 20.7	PRE01-C
115	The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1	Rule 20.8	N/A
116	All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #defined before evaluation	Rule 20.9	N/A
117	A macro parameter immediately following a # operator shall not immediately be followed by a ## operator	Rule 20.11	N/A

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
118	A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators	Rule 20.12	N/A
119	A line whose first token is # shall be a valid preprocessing directive	Rule 20.13	N/A
120	All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related	Rule 20.14	N/A
121	#define and #undef shall not be used on a reserved identifier or reserved macro name	Rule 21.1	DCL37-C
122	A reserved identifier or macro name shall not be declared	Rule 21.2	DCL37-C
123	The memory allocation and deallocation functions of <stdlib.h> shall not be used	Rule 21.3	API03-C API04-C MSC24-C
124	The standard header file <setjmp.h> shall not be used	Rule 21.4	N/A
125	The Standard Library input/output functions shall not be used	Rule 21.6	N/A
126	The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used	Rule 21.7	N/A
127	The library functions bsearch and qsort of <stdlib.h> shall not be used	Rule 21.9	N/A
128	The standard header file <tgmath.h> shall not be used	Rule 21.11	N/A
129	The exception handling features of <fenv.h> should not be used	Rule 21.12	N/A
130	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF	Rule 21.13	N/A
131	The Standard Library function memcmp shall not be used to compare null terminated strings	Rule 21.14	N/A
132	The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types	Rule 21.15	N/A
133	The pointer arguments to the Standard Library function memcmp shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type	Rule 21.16	N/A
134	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	Rule 21.17	N/A
135	The size_t argument passed to any function in <string.h> shall have an appropriate value	Rule 21.18	N/A

continues on next page

Table 1 – continued from previous page

Zephyr rule	Description	MISRA-C 2012 reference	CERT C reference
136	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type	Rule 21.19	N/A
137	The pointer returned by the Standard Library functions asctime, ctime, gmtime, localtime, localeconv, getenv, setlocale or strerror shall not be used following a subsequent call to the same function	Rule 21.20	N/A
138	All resources obtained dynamically by means of Standard Library functions shall be explicitly released	Rule 22.1	N/A
139	A block of memory shall only be freed if it was allocated by means of a Standard Library function	Rule 22.2	N/A
140	The same file shall not be open for read and write access at the same time on different streams	Rule 22.3	N/A
141	There shall be no attempt to write to a stream which has been opened as read-only	Rule 22.4	N/A
142	A pointer to a FILE object shall not be dereferenced	Rule 22.5	N/A
143	The value of a pointer to a FILE shall not be used after the associated stream has been closed	Rule 22.6	N/A
144	The macro EOF shall only be compared with the unmodified return value from any Standard Library function capable of returning EOF	Rule 22.7	N/A
145	The value of errno shall be set to zero prior to a call to an errno-setting-function	Rule 22.8	N/A
146	The value of errno shall be tested against zero after calling an errno-setting-function	Rule 22.9	N/A
147	The value of errno shall only be tested when the last function to be called was an errno-setting-function	Rule 22.10	N/A

Additional rules

Rule A.1: Conditional Compilation

Severity
Required

Description
Do not conditionally compile function declarations in header files. Do not conditionally compile structure declarations in header files. You may conditionally exclude fields within structure definitions to avoid wasting memory when the feature they support is not enabled.

Rationale
Excluding declarations from the header based on compile-time options may prevent their documentation from being generated. Their absence also prevents use of `if (IS_ENABLED(CONFIG_FOO)) {}` as an alternative to preprocessor conditionals when the code path should change based on the selected options.

Rule A.2: Inclusive Language

Severity

Required

Description

Do not introduce new usage of offensive terms listed below. This rule applies but is not limited to source code, comments, documentation, and branch names. Replacement terms may vary by area or subsystem, but should aim to follow updated industry standards when possible.

Exceptions are allowed for maintaining existing implementations or adding new implementations of industry standard specifications governed externally to the Zephyr Project.

Existing usage is recommended to change as soon as updated industry standard specifications become available or new terms are publicly announced by the governing body, or immediately if no specifications apply.

Offensive Terms	Recommended Replacements
`{master,leader} / slave`	• `{primary,main} / {secondary,replica}` • `{initiator,requester} / {target,responder}` • `{controller,host} / {device,worker,proxy, target}` • `director / performer` • `central / peripheral`
`blacklist / whitelist`	• `denylist / allowlist` • `blocklist / allowlist` • `rejectlist / acceptlist`
`grandfather policy`	• `legacy`
`sanity`	• `coherence` • `confidence`

Rationale

Offensive terms do not create an inclusive community environment and therefore violate the Zephyr Project Code of Conduct. This coding rule was inspired by a similar rule in Linux.

Status

Related GitHub Issues and Pull Requests are tagged with the Inclusive Language Label.

Area	Selected Replacements	Status
Bluetooth	See Bluetooth Appropriate Language Mapping Tables	
CAN	This CAN in Automation Inclusive Language news post has a list of general recommendations. See CAN in Automation Inclusive Language for terms to be used in specification document updates.	
eSPI	• `master / slave` => `controller / target`	Refer to eSPI Specification for new terminology
gPTP	• `master / slave` => TBD	
Inter-Integrc Circuit (I2C) Bus	• `master / slave` => `controller / target`	Refer to I2C Specification for new terminology.
Inter-IC Sound (I2S) Bus	• `master / slave` => `controller / target`	Refer to I2S Specification for new terminology. The Zephyr I2S API was migrated to these terms in Zephyr 4.4; the former macro names are retained as deprecated aliases and will be removed in Zephyr 5.0.
SMP/A	• `master / slave` => TBD	
Serial Peripheral Interface (SPI) Bus	• `master / slave` => `controller / peripheral` • `MOSI / MISO / SS` => `SDO / SDI / CS`	The Open Source Hardware Association has selected these replacement terms. See OSHWA Resolution to Redefine SPI Signal Names. The Zephyr SPI API and tree were migrated to these terms in v4.5; the former names are retained as compatibility aliases, deprecated since v4.5 and scheduled for removal in v5.0.
Test Runner (Twiste	• `platform_whitelist` => `platform_allow` • `sanitycheck` => `twister`	

Rule A.3: Macro name collisions

Severity

Required

Description

Macros with commonly used names such as `MIN`, `MAX`, `ARRAY_SIZE`, must not be modified or protected to avoid name collisions with other implementations. In particular, they must not be prefixed to place them in a Zephyr-specific namespace, re-defined using `#undef`, or conditionally excluded from compilation using `#ifndef`. Instead, if a conflict arises with an existing definition originating from a *module*, the module's code itself needs to be modified (ideally upstream, alternatively via a change in Zephyr's own fork).

This rule applies to Zephyr as a project in general, regardless of the time of introduction of the macro or its current name in the tree. If a macro name is commonly used in several other well-known open source projects then the implementation in Zephyr should use that name. While there is a subjective and non-measurable component to what "commonly

used" means, the ultimate goal is to offer users familiar macros.

Finally, this rule applies to inter-module name collisions as well: in that case both modules, prior to their inclusion, should be modified to use module-specific versions of the macro name that collides.

Rationale

Zephyr is an RTOS that comes with additional functionality and dependencies in the form of modules. Those modules are typically independent projects that may use macro names that can conflict with other modules or with Zephyr itself. Since, in the context of this documentation, Zephyr is considered the central or main project, it should implement the non-namespaced versions of the macros. Given that Zephyr uses a fork of the corresponding upstream for each module, it is always possible to patch the macro implementation in each module to avoid collisions.

Rule A.4: C Standard Library Usage Restrictions in Zephyr Kernel

Severity

Required

Description

The use of the C standard library functions and macros in the Zephyr kernel shall be limited to the following functions and macros from the ISO/IEC 9899:2011 standard, also known as C11, and their extensions:

Table 2: List of allowed libc functions and macros in the Zephyr kernel

Function	Source
abort()	ISO/IEC 9899:2011
abs()	ISO/IEC 9899:2011
aligned_alloc()	ISO/IEC 9899:2011
assert()	ISO/IEC 9899:2011
atoi()	ISO/IEC 9899:2011
bsearch()	ISO/IEC 9899:2011
calloc()	ISO/IEC 9899:2011
exit()	ISO/IEC 9899:2011
fprintf()	ISO/IEC 9899:2011
fputc()	ISO/IEC 9899:2011
fputs()	ISO/IEC 9899:2011
free()	ISO/IEC 9899:2011
fwrite()	ISO/IEC 9899:2011
gmtime()	ISO/IEC 9899:2011
isalnum()	ISO/IEC 9899:2011
isalpha()	ISO/IEC 9899:2011
iscntrl()	ISO/IEC 9899:2011
isdigit()	ISO/IEC 9899:2011
isgraph()	ISO/IEC 9899:2011
isprint()	ISO/IEC 9899:2011
isspace()	ISO/IEC 9899:2011
isupper()	ISO/IEC 9899:2011
isxdigit()	ISO/IEC 9899:2011
labs()	ISO/IEC 9899:2011
llabs()	ISO/IEC 9899:2011
malloc()	ISO/IEC 9899:2011
memchr()	ISO/IEC 9899:2011
memcmp()	ISO/IEC 9899:2011
memcpy()	ISO/IEC 9899:2011
memmove()	ISO/IEC 9899:2011

continues on next page

Table 2 – continued from previous page

Function	Source
memset()	ISO/IEC 9899:2011
perror()	ISO/IEC 9899:2011
printf()	ISO/IEC 9899:2011
putc()	ISO/IEC 9899:2011
putchar()	ISO/IEC 9899:2011
puts()	ISO/IEC 9899:2011
qsort()	ISO/IEC 9899:2011
rand()	ISO/IEC 9899:2011
realloc()	ISO/IEC 9899:2011
snprintf()	ISO/IEC 9899:2011
sprintf()	ISO/IEC 9899:2011
sqrt()	ISO/IEC 9899:2011
sqrtf()	ISO/IEC 9899:2011
srand()	ISO/IEC 9899:2011
strcat()	ISO/IEC 9899:2011
strchr()	ISO/IEC 9899:2011
strcmp()	ISO/IEC 9899:2011
strcpy()	ISO/IEC 9899:2011
strcspn()	ISO/IEC 9899:2011
strerror()	ISO/IEC 9899:2011
strlen()	ISO/IEC 9899:2011
strncat()	ISO/IEC 9899:2011
strncmp()	ISO/IEC 9899:2011
strncpy()	ISO/IEC 9899:2011
strnlen()	POSIX.1-2008
strrchr()	ISO/IEC 9899:2011
strspn()	ISO/IEC 9899:2011
strstr()	ISO/IEC 9899:2011
strtol()	ISO/IEC 9899:2011
strtoll()	ISO/IEC 9899:2011
strtoul()	ISO/IEC 9899:2011
strtoull()	ISO/IEC 9899:2011
time()	ISO/IEC 9899:2011
tolower()	ISO/IEC 9899:2011
toupper()	ISO/IEC 9899:2011
vfprintf()	ISO/IEC 9899:2011
vprintf()	ISO/IEC 9899:2011
vsnprintf()	ISO/IEC 9899:2011
vsprintf()	ISO/IEC 9899:2011

All of the functions listed above must be implemented by the *minimal libc* to ensure that the Zephyr kernel can build with the minimal libc.

In addition, any functions from the above list that are not part of the ISO/IEC 9899:2011 standard must be implemented by the *common libc* to ensure their availability across multiple C standard libraries.

Introducing new C standard library functions to the Zephyr kernel is allowed with justification given that the above requirements are satisfied.

Note that the use of the functions listed above are subject to secure and safe coding practices and it should not be assumed that their use in the Zephyr kernel is unconditionally permitted by being listed in this rule.

The "Zephyr kernel" in this context consists of the following components:

- Kernel (`kernel`)

- OS Library (`lib/os`)
- Architecture Port (`arch`)
- Logging Subsystem (`subsys/logging`)

Rationale

Zephyr kernel must be able to build with the *minimal libc*, a limited C standard library implementation that is part of the Zephyr RTOS and maintained by the Zephyr Project, to allow self-contained testing and verification of the kernel and core OS services.

In order to ensure that the Zephyr kernel can build with the minimal libc, it is necessary to restrict the use of the C standard library functions and macros in the Zephyr kernel to the functions and macros that are available as part of the minimal libc.

Rule A.5: C Standard Library Usage Restrictions in Zephyr Codebase

Severity

Required

Description

The use of the C standard library functions and macros in the Zephyr codebase shall be limited to the functions, excluding the Annex K "Bounds-checking interfaces", from the ISO/IEC 9899:2011 standard, also known as C11, unless exempted by this rule.

The "Zephyr codebase" in this context refers to all embedded source code files committed to the main Zephyr repository, except the Zephyr kernel as defined by the *Rule A.3: Macro name collisions*. With embedded source code we refer to code which is meant to be executed in embedded targets, and therefore excludes host tooling, and code specific for the native test targets.

The following non-ISO 9899:2011, hereinafter referred to as non-standard, functions and macros are exempt from this rule and allowed to be used in the Zephyr codebase:

Table 3: List of allowed non-standard libc functions

Function	Source
gmtime_r()	POSIX.1-2001
strnlen()	POSIX.1-2008
strtok_r()	POSIX.1-2001

All non-standard functions and macros listed above must be implemented by the *common libc* in order to make sure that these functions can be made available when using a C standard library that does not implement these functions.

Adding a new non-standard function from common C standard libraries to the above list is allowed with justification, given that the above requirement is satisfied. However, when there exists a standard function that is functionally equivalent, the standard function shall be used.

Rationale

Some C standard libraries, such as Newlib and Picolibc, include additional functions and macros that are defined by the standards and de-facto standards that extend the ISO C standard (e.g. POSIX, Linux).

The ISO/IEC 9899:2011 standard does not require C compiler toolchains to include the support for these non-standard functions, and therefore using these functions can lead to compatibility issues with the third-party toolchains that come with their own C standard libraries.

7.1.5 Coding Style Guidelines

Naming conventions

This section describes the naming conventions adopted by the Zephyr Project, for each individual programming language or tool used in it.

C Code naming conventions The naming conventions in this section apply to C source and header files, as stated in each individual sub-section.

Public symbol prefixes All *public APIs* introduced to Zephyr must be prefixed according to the area or subsystem they belong to. Examples of area or subsystem prefixes are provided below for reference.

- `k_` for the kernel
- `sys_` for system-wide code and features
- `net_` for the networking subsystem
- `bt_` for the Bluetooth subsystem
- `i2c_` for the I2C controller subsystem

C Code and General Style Guidelines

Coding style is enforced on any new or modified code, but contributors are not expected to correct the style on existing code that they are not modifying.

For style aspects where the guidelines don't offer explicit guidance or permit multiple valid ways to express something, contributors should follow the style of existing code in the tree, with higher importance given to "nearby" code (first look at the function, then the same file, then subsystem, etc).

In general, follow the Linux kernel coding style, with the following exceptions and clarifications:

- Tabs are 8 characters.
- Use snake case for code and variables.
- The line length is 100 columns or fewer. In the documentation, longer lines for URL references are an allowed exception.
- Add braces to every `if`, `else`, `do`, `while`, `for` and `switch` body, even for single-line code blocks.
- Use spaces instead of tabs to align comments after declarations, as needed.
- Use C89-style single line comments, `/* */`. The C99-style single line comment, `//`, is not allowed.
- Use `/** */` for doxygen comments that need to appear in the documentation.
- Avoid using binary literals (constants starting with `0b`).
- Avoid using non-ASCII symbols in code, unless it significantly improves clarity, avoid emojis in any case.
- Use proper capitalization of nouns in code comments (e.g. `UART` and not `uart`, `CMake` and not `cmake`).

Doxygen Style Guidelines

The Zephyr Project uses Doxygen to generate API documentation from source code comments. This guide defines the conventions for documenting Zephyr public APIs consistently.

All the Doxygen commands are available for use even if they don't explicitly appear in this document.

To build and preview Zephyr's Doxygen documentation locally, refer to *Documentation Generation*.

General Rules All *public header files and their public API symbols* (functions, structs, enums, unions, typedefs, macros, and global variables):

- Must be fully documented (see *Hiding internal details* for exceptions)
- Must belong to at least one Doxygen group (see *Groups*)

You must use the following syntax when documenting public APIs:

- Use `/**` to start a block comment and `*/` to end it.
- Use `/**<` for trailing comments on the same line as a symbol.
- Use `@` (not `\`) for Doxygen commands (e.g., `@param` not `\param`).
- The `@brief` command is optional. If not used, the first sentence (ending with a period) is treated as the brief description.

Constructs meant only for internal use must be hidden from the public documentation, as described in *Hiding internal details*.

What to document For any API element that accepts, returns, or stores a value (function parameters, return values, struct/union members, enum values, typedefs, and macros), their documentation must describe the following items when applicable:

Semantics
What this element represents and how callers/users should interpret it. Don't restate the identifier or the C type.

Example:

- Avoid: `@param timeout Timeout value in ms.`
- Prefer: `@param timeout Maximum time to wait before returning, in milliseconds.`

Valid values
What values are accepted, and how they are interpreted. Specify one or more of the following:

- Range: minimum/maximum values (for example, the type is `uint8_t` but only 0–100 are valid).
- Discrete set: permitted values, when only a subset is allowed.
- Enum: whether the value must be a valid member of a given enum (and whether all enumerators are allowed or only a subset).
- Flags/bitmasks: which bits are valid and whether combinations are allowed.
- Nullability: whether `NULL` is allowed for pointer values, and what it means.

Units
When the value represents a quantity, specify the unit and reference (for example, milliseconds vs ticks, Hertz, bytes). Use SI unit symbols when applicable, and write a space between the number and the unit symbol (for example, `10 ms`).

Representation
Any non-obvious encoding or scaling (for example, fixed-point scale, Q format, split integer and fractional fields, endianness requirements).

Ownership and lifetime
For pointers/buffers, state who allocates/frees and how long the memory must remain valid. Note required size/alignment when relevant.

Writing style Brief descriptions should use the imperative mood (a verb phrase) rather than third-person prose. This keeps API summaries consistent and scannable.

- Avoid: "Transmits data through a pipe.", "This function gets the device state.".
- Prefer: "Transmit data through a pipe.", "Get the device state.", "Initialize the subsystem.".

Parameter and member descriptions may be sentence fragments, but they should remain descriptive and avoid repeating the parameter/member name.

Groups Groups organize related symbols into a hierarchy that helps users navigate the API.

- Define each group once with `@defgroup`.
  - The provided title should be a short, descriptive name for the group. As groups inherently collapse various interfaces/API together, do *not* use "API" or "Interface" (or any other variant of these) terms in the title, as this would be redundant.
  - The brief description of the group should not paraphrase its title.
- Group names use snake_case.
- Use `@ingroup` to specify the parent group(s) a group belongs to.

Example:

```
/**
 * @defgroup mqtt_socket MQTT Client library
 * @ingroup networking
 * @since 1.14
 * @version 0.8.0
 * @{
 */

 /* documented contents of the MQTT header file */

 /** @} */
```

Note

A group may belong to multiple parent groups, so you may have multiple `@ingroup` commands. For example, device driver emulators typically appear both in "Emulator Interfaces" and "Device drivers" groups. See I2C Emulation Interface for an example.

Important

A group with no parent becomes a top-level entry of the *API Overview*, whose categories are deliberately few. Always give a new group a parent with `@ingroup`.

The top-level entries are allowlisted in doc/_doxygen/toplevel_groups.txt and enforced by scripts/ci/doxygen_toplevel_groups.py. Adding a new one requires documentation maintainer approval.

API Versioning Use `@since` and `@version` on group definitions (`@defgroup`) to document API history and maturity:

- `@since` indicates the Zephyr release when the API was introduced (e.g., `@since 3.7`)
- `@version` indicates the current API version, following semantic versioning

The version number reflects the API's maturity as defined in *API Overview*.

Example:

Listing 1: Example for an API introduced in Zephyr 1.14, with current version 0.8.0 (unstable).

```
/**
 * @defgroup mqtt_socket MQTT Client library
 * @ingroup networking
 * @since 1.14
 * @version 0.8.0
 * @{
 */

 /* documented contents of the MQTT header file */

 /** @} */
```

Files Each public header file must have an `@file` block at the top, appearing after the SPDX license and copyright notice.

The `@file` block must also belong to a Doxygen group, typically the one defined in the same header file. This allows users browsing by file to easily navigate to the associated group.

```
/**
 * @file
 * @brief Public API for the GPIO driver.
 * @ingroup gpio_interface
 */
```

Type Definitions Document `struct`, `enum`, `union`, and `typedef` definitions with a brief description.

Document each member (struct field, enum value, etc.) as well. It is recommended to use `/**<` trailing comments style when only a brief description is provided and it can fit on a single line.

Listing 2: Examples of fully documented enum, struct, and typedef.

```
/** @brief Parity modes */
enum uart_config_parity {
     UART_CFG_PARITY_NONE,    /**< No parity */
     UART_CFG_PARITY_ODD,     /**< Odd parity */
     UART_CFG_PARITY_EVEN,    /**< Even parity */
     UART_CFG_PARITY_MARK,    /**< Mark parity */
     UART_CFG_PARITY_SPACE,   /**< Space parity */
};

/**
 * GPIO pin configuration.
 *
 * Specifies direction, pull resistor, and interrupt settings.
```

(continues on next page)

(continued from previous page)

```
 */
struct gpio_config {
    uint32_t flags;    /**< Pin configuration flags. */
    gpio_pin_t pin;    /**< Pin number within the port. */
};

/**
 * @brief Identifies a set of pins associated with a port.
 *
 * The pin with index n is present in the set if and only if the bit
 * identified by (1U << n) is set.
 */
typedef uint32_t gpio_port_pins_t;
```

Functions

Parameters

- Document parameters with `@param` in declaration order
- Use `@param[out]` for pointers the function writes to.
- Use `@param[in,out]` for pointers the function both reads and writes.
- You may omit direction specifiers for const pointers and scalars (implicitly input-only).

Return Values

- Use `@return <description>` for general descriptions (e.g., boolean or computed values).
- Use `@retval <value> <description>` for specific, discrete return values (typically error codes), starting with the success case (when applicable). The `<value>` must be provided as the first word, followed by the description (e.g., `@retval -EINVAL Invalid arguments`, not `@retval -EINVAL if invalid arguments`).

> **Note**
>
> Since there might be multiple reasons for the same discrete value to be returned, it is allowed to use the same value in multiple `@retval` statements.

Example:

Listing 3: Examples of fully documented functions.

```
/**
 * @brief Write data to the TX queue from a provided buffer
 *
 * @param dev Pointer to the device structure for the driver instance.
 * @param buf Pointer to a buffer containing the data to transmit.
 * @param size Number of bytes to write. This value has to be equal or smaller
 *        than the size of the channel's TX memory block configuration.
 *
 * @retval 0 on success.
 * @retval -EIO The interface is not in READY or RUNNING state.
 * @retval -EBUSY Returned without waiting.
 * @retval -EAGAIN Waiting period timed out.
 * @retval -ENOMEM No memory in TX slab queue.
```

(continues on next page)

(continued from previous page)

```
 * @retval -EINVAL Size parameter larger than TX queue memory block.
 */
int i2s_buf_write(const struct device *dev, void *buf, size_t size);

/**
 * @brief Add an application callback.
 *
 * @param port Pointer to the device structure for the driver instance.
 * @param callback A valid application's callback structure pointer.
 *
 * @return 0 on success, negative errno value on failure.
 * @retval -ENOSYS Driver does not implement the operation.
 */
int gpio_add_callback(const struct device *port, struct gpio_callback *callback);

/**
 * @brief Helper function for converting struct sensor_value to float.
 *
 * @param val A pointer to a sensor_value struct.
 * @return The converted value.
 */
static inline float sensor_value_to_float(const struct sensor_value *val)
{
    return (float)val->val1 + (float)val->val2 / 1000000;
}
```

Macros For function-like macros, document parameters like you would for functions.

Listing 4: Example of a fully documented function-like macro.

```
/**
 * @brief Get a node's (only) register block size
 *
 * Equivalent to DT_REG_SIZE_BY_IDX(node_id, 0).
 *
 * @param node_id node identifier
 * @return node's only register block's size
 */
#define DT_REG_SIZE(node_id) DT_REG_SIZE_BY_IDX(node_id, 0)
```

Referencing the main documentation API documentation can reference content from the main, Sphinx-based documentation using the commands described below. In the generated API documentation pages, these references are rendered as hyperlinks pointing back to the corresponding page of the main documentation.

`@kconfig{<option>}`
: Reference a Kconfig option by its full name (including the `CONFIG_` prefix). This is the Doxygen counterpart of the *kconfig:option* role.

  Example: `@kconfig{CONFIG_GPIO}`

`@kconfig_regex{<regex>}`
: Reference all the Kconfig options matching a regular expression, as a link to the Kconfig search page with the pattern pre-filled. This is the Doxygen counterpart of the *kconfig:option-regex* role. As commas have a special meaning in Doxygen commands, they must be escaped with a backslash.

  Example: `@kconfig_regex{CONFIG_SECURE_STORAGE_ITS_.*_CUSTOM}`

@dtcompatible{<compatible>}
Reference a Devicetree binding by its compatible string. This is the Doxygen counterpart of the *dtcompatible* role. As commas have a special meaning in Doxygen commands, they must be escaped with a backslash.

Example: `@dtcompatible{zephyr\,input-longpress}`

@rstref{<target>} or @rstref{<text> <target>}
Reference any documentation page or section by its reference label (or document name), similar to the Sphinx `ref` role. When no custom text is provided, the title of the referenced page or section is used as the link text.

Example: `@rstref{zephyr_licensing}` or `@rstref{the licensing page <zephyr_licensing>}`

References are checked when the documentation is built: a reference to a Kconfig option, binding, or label that does not exist causes a documentation build warning.

> **Note**
> These commands expand to plain text when the Doxygen documentation is built standalone, i.e. without the rest of the documentation. See *Documentation Generation* for more details.

Referencing IETF RFCs

@rfc{<number>} or @rfc{<number>,<anchor>}
Reference an IETF RFC by number, optionally pointing to an anchor within it. This is the Doxygen counterpart of the `rfc` role and renders as a hyperlink to the RFC on the IETF Datatracker.

The anchor is passed through verbatim, so it can target anything the Datatracker defines for that RFC, such as `section-3.1`, `appendix-B.1.2`, `figure-2`, or `table-1`. Which anchors exist depends on the RFC: figures and tables are only anchored on RFCs rendered from their XML source.

Do not put whitespace after the comma: Doxygen keeps it as part of the argument, where it ends up inside the URL fragment. The rendered link text still looks correct, so a broken anchor is only noticeable when following the link.

Example: `@rfc{7519}`, `@rfc{8613,section-3.1}` or `@rfc{8613,appendix-B.1.2}`

Hiding internal details Use `@cond INTERNAL_HIDDEN` / `@endcond` to hide internal details from generated documentation.

You may still optionally document these internal symbols to provide a useful reference to internal users.

```
/** Timer structure.
 *
 * Opaque type for a timer object. All the fields in this structure are internal␣
↪and should not
 * be accessed outside of kernel code.
 */
struct k_timer {
    /** @cond INTERNAL_HIDDEN */

    /* ... internal members ... */

    /** @endcond */
};
```

Driver Backend API Driver subsystems expose a public API for applications and a "backend" API for driver implementers.

While the backend API is not intended to be used directly by applications, it still constitutes a public contract between the subsystem and driver implementers and must therefore be properly documented. It typically includes the driver operations structure, typedefs defining the signature of each operation, and, in some cases, additional helper types or macros that are useful for driver implementation.

Backend API Doxygen group Use `@def_driverbackendgroup` to create a subgroup that is a child of the main API group and that will contain all the symbols associated with the backend API. This command takes two arguments: a human-readable name for the backend API (typically the same as the parent group) and the parent group identifier.

```
/**
 * @def_driverbackendgroup{Haptics,haptics_interface}
 * @{
 */

/* callback typedefs, driver ops struct, helpers ... */

/** @} */
```

Driver operations typedefs Define a `typedef` for each driver operation. The detailed description may reference the corresponding public API function as it often has a similar signature.

```
/**
 * @brief Set the haptic device to stop output.
 * See haptics_stop_output() for argument description.
 */
typedef int (*haptics_stop_output_t)(const struct device *dev);
```

Driver operations structure Annotate the struct with `@driver_ops{Name}` (where *Name* matches the name passed to `@def_driverbackendgroup`). For each member, use `@driver_ops_mandatory` or `@driver_ops_optional` to indicate whether the driver must implement it, and `@copybrief` to inherit the brief from the public API function:

```
/**
 * @driver_ops{Haptics}
 */
__subsystem struct haptics_driver_api {
    /**
     * @driver_ops_mandatory @copybrief haptics_start_output
     */
    haptics_start_output_t start_output;
    /**
     * @driver_ops_mandatory @copybrief haptics_stop_output
     */
    haptics_stop_output_t stop_output;
    /**
     * @driver_ops_optional @copybrief haptics_register_error_callback
     */
    haptics_register_error_callback_t register_error_callback;
};
```

Conditional code To ensure conditionally-compiled code appears in documentation, use either of the following approaches:

1. Add the Kconfig macro (`CONFIG_...`) that is gating the code to the `PREDEFINED` list in doc/zephyr.doxyfile.in. This makes the code visible to Doxygen during documentation generation.
2. Alternatively, you can rely on the `__DOXYGEN__` macro being set to make the conditional code visible to Doxygen, as this macro is automatically defined by Doxygen.

```
struct coap_packet {
    uint8_t *data;  /**< User allocated buffer. */
#if defined(CONFIG_COAP_KEEP_USER_DATA) || defined(__DOXYGEN__)
    /**
     * Application-specific user data.
     * @kconfig_dep{CONFIG_COAP_KEEP_USER_DATA}
     */
    void *user_data;
#endif
};
```

Kconfig Dependencies The `@kconfig_dep` command can be used to document the Kconfig option(s) that are required in order for an API symbol to be available to the user. The command can be used with one, two or three Kconfig options.

For example, adding `@kconfig_dep{CONFIG_PM,CONFIG_SMP}` to the Doxygen documentation of an API symbol will cause the generated documentation to include a note reading: "Available only when the following Kconfig options are enabled: `CONFIG_PM`, `CONFIG_SMP`.".

You can see an example of `@kconfig_dep` usage in the documentation of `coap_packet`.

CMake Style Guidelines

A subset of these guidelines is enforced in CI by the `CMakeStyle` compliance check. You can run the same checks locally with `scripts/cmake/cmake_style.py` (a directory is searched recursively for `CMakeLists.txt` and `*.cmake` files):

```
pip install tree-sitter tree-sitter-cmake
./scripts/cmake/cmake_style.py path/to/CMakeLists.txt
./scripts/cmake/cmake_style.py drivers/
```

General Formatting

- **Indentation**: Use **2 spaces** for indentation. Avoid tabs to ensure consistency across different environments.
- **Line Length**: Limit line length to **100 columns** where possible.
- **Empty Lines**: Use empty lines to separate logically distinct sections within a CMake file.
- **No Space Before Opening Brackets**: Do not add a space between a command and the opening parenthesis. Use `if(...)` instead of `if (...)`.

  ```
  # Good:
  if(ENABLE_TESTS)
    add_subdirectory(tests)
  endif()

  # Bad:
  if (ENABLE_TESTS)
    add_subdirectory(tests)
  endif()
  ```

Commands and Syntax

- **Lowercase Commands**: Always use **lowercase** CMake commands (e.g., `add_executable`, `find_package`). This improves readability and consistency.

```
# Good:
add_library(my_lib STATIC src/my_lib.cpp)

# Bad:
ADD_LIBRARY(my_lib STATIC src/my_lib.cpp)
```

  Commands defined by CMake modules, and Zephyr or sysbuild extension commands that extend an existing mixed-case command, are an exception: they follow a mixed-case `Module_Action` naming convention and must be spelled with their canonical case (e.g. CMake's `ExternalProject_Add` extended by Zephyr's `ExternalZephyrProject_Add`). The `CMakeStyle` check exempts these from the lowercase rule. Extensions of a lowercase command stay lowercase (e.g. `add_dependencies` extended by sysbuild's `sysbuild_add_dependencies`).

- **One File Argument per Line**: Break the file arguments across multiple lines to make it easier to scan and identify each source file or item.

```
# Good:
target_sources(my_target PRIVATE
  src/file1.cpp
  src/file2.cpp
)

 # Bad:
target_sources(my_target PRIVATE src/file1.cpp src/file2.cpp)
```

Variable Naming

- **Use Uppercase for Cache Variables or variables shared across CMake files**: When defining cache variables using `option` or `set(... CACHE ...)`, use **UPPERCASE names**.

```
option(ENABLE_TESTS "Enable test suite" ON)
set(CMAKE_CXX_STANDARD 17 CACHE STRING "C++ standard version")
```

- **Use Lowercase for Local Variables**: For local variables within CMake files, use **lowercase** or **snake_case**.

```
set(output_dir "${CMAKE_BINARY_DIR}/output")
```

- **Consistent Prefixing**: Use consistent prefixes for variables to avoid name conflicts, especially in large projects.

```
set(MYPROJECT_SRC_DIR "${CMAKE_SOURCE_DIR}/src")
```

Quoting

- **Quote Strings and Variables**: Always quote string literals and variables to prevent unintended behavior, especially when dealing with paths or arguments that may contain spaces.

```
# Good:
set(my_path "${CMAKE_SOURCE_DIR}/include")

# Bad:
set(my_path ${CMAKE_SOURCE_DIR}/include)
```

- **Do Not Quote Boolean Values**: For boolean values (`ON`, `OFF`, `TRUE`, `FALSE`), avoid quoting them.

```
option(BUILD_SHARED_LIBS "Build shared libraries" OFF)
```

Avoid Hardcoding Paths

- Use CMake variables (`CMAKE_SOURCE_DIR`, `CMAKE_BINARY_DIR`, `CMAKE_CURRENT_SOURCE_DIR`) instead of hardcoding paths.

```
set(output_dir "${CMAKE_BINARY_DIR}/bin")
```

Conditional Logic

- Use `if`, `elseif`, and `else` with proper indentation, and close with `endif()`.

```
if(ENABLE_TESTS)
  add_subdirectory(tests)
endif()
```

Documentation

- Use comments to document complex logic in the CMake files.

```
# Find LlvmLld components required for building with llvm
find_package(LlvmLld 14.0.0 REQUIRED)
```

Devicetree Style Guidelines

- Indent with tabs.
- Tab size is 8 characters.
- Follow the Devicetree specification conventions and rules.
- If the Linux kernel rules in Devicetree Sources (DTS) Coding Style make a recommendation, it's the preferred style in Zephyr too.
- You can split related groups of properties into "paragraphs" by separating them with one empty line (two newline characters) if it aids readability.
- Use dashes (`-`) as word separators for node and property names.
- Use underscores (`_`) as word separators in node labels.
- Leave a single space on each side of the equal sign (`=`) in property definitions.
- Don't insert empty lines before a dedenting `};`.
- Insert a single empty line to separate nodes at the same hierarchy level.
- When splitting long property values across multiple lines, specify the first value on the same line as the opening bracket (`<` or `[`). Place the closing bracket and semicolon (`>;` or `];`) on the same line after the final value of the property.

Examples:

```
/dts-v1/;

/ {
        /* Node labels use the "_" separator, but node names use "-". */
        psa_rng: psa-rng {
```

(continues on next page)

(continued from previous page)

```
                compatible = "zephyr,psa-crypto-rng";
                status = "okay";
        };

        ramdisk0 {
                compatible = "zephyr,ram-disk";
                /* Property names use the "-" separator. */
                disk-name = "RAM";
                sector-size = <512>;
                sector-count = <128>;
        };

        periph0 {
                /* Wrap array values to conform with line lengths. Indent
                 * continuation so that values align.
                 */
                pinctrl-0 = <&periph0_siga_px0_default &periph0_sigb_py7_default
                             &periph0_sigc_pz1_default>;
                pinctrl-names = "default";
        };

        mipi-dbi {
                compatible = "zephyr,mipi-dbi-spi";

                lcd0: lcd@0 {
                        compatible = "sitronix,st7735r";
                        /* Split array values across multiple lines to help␣
↪readability. */
                        gamctrp1 = [10 0e 02 03
                                    0e 07 02 07
                                    0a 12 27 37
                                    00 0d 0e 10];
                        gamctrn1 = [10 0e 03 03
                                    0f 06 02 08
                                    0a 13 26 36
                                    00 0d 0e 10];
                };
        };
};
```

Kconfig Style Guidelines

This document provides style guidelines for writing Kconfig files in the Zephyr project. Following these guidelines ensures consistency and readability across the codebase, making it easier for developers to understand and maintain configuration options.

The following sections provide guidelines with examples to illustrate proper Kconfig formatting and naming conventions.

Basic Formatting Rules When writing Kconfig files, follow these basic formatting rules:

- **Line length**: Keep lines to 100 columns or fewer.
- **Indentation**: Use tabs for indentation, except for `help` entry text which should be placed at one tab plus two extra spaces.
- **Spacing**: Leave a single empty line between option declarations.

- **Comments**: Format comments as `# Comment` rather than `#Comment`.
- **Conditional blocks**: Insert an empty line before/after each top-level `if` and `endif` statement.
- **End of file**: End the file with exactly one newline.

For guidance on using statements like `select`, see *Kconfig - Tips and Best Practices* for more information.

These formatting rules are enforced in CI by the `KconfigFormat` compliance check. You can check your Kconfig files locally with the `scripts/kconfig/kconfig_style.py` script, which reports any style issues (a directory is searched recursively for Kconfig files):

```
./scripts/kconfig/kconfig_style.py path/to/Kconfig
./scripts/kconfig/kconfig_style.py drivers/sensor/
```

Symbol Naming and Structure The following examples demonstrate proper Kconfig symbol naming and structure:

```
# Symbol to enable/disable the feature ("enabling symbol")
config ADC_AD405X
        bool "AD405X ADC driver"
        depends on DT_HAS_ADI_AD4052_ADC_ENABLED || DT_HAS_ADI_AD4050_ADC_ENABLED
        select SPI
        default y
        help
          Enable ADC driver for AD405X.
```

```
# Symbol to enable/disable the feature ("enabling symbol")
menuconfig ADC_AD4114
        bool "AD4114 ADC driver"
        depends on DT_HAS_ADI_AD4114_ADC_ENABLED
        select SPI
        default y
        help
          Enable the AD4114 ADC driver.

if ADC_AD4114

# Symbol to configure the behavior of the feature ("config symbol")
config ADC_AD4114_ACQUISITION_THREAD_STACK_SIZE
        int "Stack size for the data acquisition thread"
        default 512
        help
          Size of the stack used for the internal data acquisition
          thread.

# Symbol to configure the behavior of the feature ("config symbol")
config ADC_AD4114_ACQUISITION_THREAD_PRIO
        int "Priority for the data acquisition thread"
        default 0
        help
          Priority level for the internal ADC data acquisition thread.

endif # ADC_AD4114
```

Naming Conventions

- As a general rule, symbols concerning the same component should be distinct from other symbols. This can usually be accomplished by using a common prefix. This prefix can be a simple keyword or, as in the case for drivers, several keywords for more precision.
- A common prefix usually indicates the subsystem or component the symbol belongs to.
- An enabling symbol name should consist of keywords to provide the context of the symbol in a scope from most general to most specific (e.g. *Driver Type -> Driver Name*).
- The prompt for an enabling symbol should use the same logic as the symbol name itself but use the keywords in reversed order.
  - Adhering to this style makes searching for symbols easier in the UI because one can filter by a scope keyword.
- When the enabling symbol is dependent on a devicetree node, consider depending on the *automatically created* `DT_HAS_<node>_ENABLED` symbol.
- When building complex expressions based on devicetree node compatibles, use the *automatically defined* `DT_COMPAT_VND_DEVICE` instead of defining a variable equal to `vnd,device` manually.

The specific formats by subtree:

- **Drivers (/drivers)**: Use the format `{Driver Type}_{Driver Name}` for symbols and `{Driver Name} {Driver Type} driver` for prompts.
- **Sensors (/drivers/sensors)**: Use the format `SENSOR_{Sensor Name}` for symbols and `{Sensor Name} {Sensor Type} sensor driver` for prompts.
- **Architecture (/arch)**: Many symbols are shared across architectures. Before creating new symbols, check if similar ones already exist in other architectures.
- **Samples (/samples)**: Use `SAMPLE_` for symbols, this is to prevent conflicts with external modules.
- **Tests (/tests)**: Use `TEST_` for symbols, this is to prevent conflicts with external modules.
- **Boards (/boards)**: Use `BOARD_` for symbols.
- **SoCs (/soc)**: Use the most appropriate base for symbols: `SOC_VENDOR_{SoC vendor}_` if it relates to several families from a SoC vendor, `SOC_FAMILY_{SoC family}_` if it relates to a whole SoC family, `SOC_SERIES_{SoC series}_` if it relates to a whole SoC series, or `SOC_{SoC}_` if it relates to a specific SoC - see *SoC Porting Guide* for details on these terms and where they must originate from. This is to prevent conflicts with other vendors and external modules.

> **Note**
>
> The following examples only show the symbol and prompt lines for brevity.

Examples **Driver Examples:**

```
config ADC_AD405X
        bool "AD405X ADC driver"

config CAN_MAX32
        bool "MAX32 CAN driver"

config I2C_EMUL
        bool "I2C emulator driver"
```

Sensor Examples:

```
config SENSOR_BMP581
        bool "BMP581 barometric pressure sensor driver"

config SENSOR_TMP007
        bool "TMP007 infrared thermopile sensor driver"

config SENSOR_LM75
        bool "LM75 temperature sensor driver"
```

Sample examples:

```
config SAMPLE_LOCK
        bool "Locking feature"

config SAMPLE_TEMPERATURE_READINGS
        bool "Fetch readings from temperature sensor"
```

Test examples:

```
config TEST_ITERATIONS
        int "Number of times test should run"
        default 3
```

SoC examples:

```
# This assumes the following soc.yml file for the vendor "hotel":
# family:
#   - name: alpha
#     series:
#       - name: foxtrot
#         socs:
#           - name: charlie

config SOC_VENDOR_HOTEL_FLASH_PREFETCH
        bool "Flash prefetch"

config SOC_FAMILY_ALPHA_JTAG_LOCK
        bool "Lock JTAG access"

config SOC_SERIES_FOXTROT_EMULATE_POWER_OFF
        bool "Emulate power off mode"

config SOC_CHARLIE_IMAGE_HEADER
        bool "Include image header"
```

Configuration Symbol Organization When a feature uses config symbols to configure its behavior:

- Use `menuconfig` instead of `config` to define the enabling feature (even when the config symbol has no prompt).
- Encapsulate the config symbols in an `if` statement to declare their dependency on the enabling symbol (this automatically groups these symbols under the enabling symbol in the UI).
- Prefix the config symbol with the enabling symbol's name for scope and context.
- In the prompt of the config symbol, describe what the symbol configures without repeating the scope keywords, because the grouping in the UI provides this context.

File Organization When organizing Kconfig files:

- Keep the Kconfig file close to the source files it configures.
- When dealing with large Kconfig files (e.g. with many config symbols), consider grouping (some of) them into a separate file and import it using the `source` directive to improve readability.

Python Style Guidelines

Python should be formatted in a PEP 8 compliant manner. Zephyr uses the ruff formatter to achieve this. This opinionated formatter aims for consistency, generality, readability and reducing git diffs.

To apply the formatter, run:

```
ruff check --select I --fix <file> # Sort imports
ruff format <file>
```

Ruff configuration A small set of options is applied on top of the defaults:

- Line length of 100 columns or fewer.
- Both single `'` and double `"` quote styles are allowed.
- Line endings will be converted to `\n`. The default line ending on Unix.

Excluded files The formatter is enforced in CI, but only for newly added Python files, because the project already had a large Python codebase when this was introduced. The .ruff-excludes.toml file has a `[format]` section where all the files that are currently excluded are listed. It is encouraged for contributors, when changing an excluded file, to remove it from the list and format it in a separate commit.

Style Tools

Checkpatch The Linux kernel GPL-licensed tool `checkpatch` is used to check coding style conformity.

> **Note**
>
> checkpatch does not currently run on Windows.

Checkpatch is available in the scripts directory. To invoke it when committing code, make the file *$ZEPHYR_BASE/.git/hooks/pre-commit* executable and edit it to contain:

```
#!/bin/sh
set -e exec
exec git diff --cached | ${ZEPHYR_BASE}/scripts/checkpatch.pl -
```

Instead of running checkpatch at each commit, you may prefer to run it only before pushing on zephyr repo. To do this, make the file *$ZEPHYR_BASE/.git/hooks/pre-push* executable and edit it to contain:

```
#!/bin/sh
remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000
```

(continues on next page)

(continued from previous page)

```
echo "Run push hook"

while read local_ref local_sha remote_ref remote_sha
do
    args="$remote $url $local_ref $local_sha $remote_ref $remote_sha"
    exec ${ZEPHYR_BASE}/scripts/series-push-hook.sh $args
done

exit 0
```

If you want to override checkpatch verdict and push you branch despite reported issues, you can add option –no-verify to the git push command.

A different way for running `checkpatch` is by using *check_compliance.py* script, which does additional style and compliance related checks.

clang-format The clang-format tool can be helpful to quickly reformat large amounts of new source code to our *Coding Style Guidelines* standards together with the `.clang-format` configuration file provided in the repository. `clang-format` is well integrated into most editors, but you can also run it manually like this:

```
clang-format -i my_source_file.c
```

`clang-format` is part of LLVM, which can be downloaded from the project releases page. Note that if you are a Linux user, `clang-format` will likely be available as a package in your distribution repositories.

When there are differences between the *Coding Style Guidelines* guidelines and the formatting generated by code formatting tools, the *Coding Style Guidelines* guidelines take precedence. If there is ambiguity between formatting tools and the guidelines, maintainers may decide which style should be adopted.

dts-linter The dts-linter can be helpful to quickly reformat large amounts of devicetree files to our *Coding Style Guidelines* standards. You can also run it manually like this:

For individual files

```
npx --ignore-scripts --prefix ./scripts/ci dts-linter --format --file board.dts --
↪file board_pinctrl.dtsi --patchFile diff.patch
git apply diff.patch
```

You can omit `--file` and this will format all files under the directory where the command has been called. Alternatively `--cwd` can also be passed set the base dir where the tool should look for files. This option is also used to make the paths relative in the patch file.

You can also fix in place with

```
npx --ignore-scripts --prefix ./scripts/ci dts-linter --formatFixAll
```

Editor Integration

- For VS Code: Install the extension from the VS Code Marketplace or Open VSIX
- For other editors with LSP Client support: Use the devicetree-language-server devicetree-language-server

Make sure you follow Devicetree Style Guidelines requirements to configure the editor correctly.

7.1.6 Proposals and RFCs

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Many changes however are "substantial" and need to go through a design process and produce a consensus among the project stakeholders.

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the project.

Contributors and project stakeholders should consider using this process if they intend to make "substantial" changes to Zephyr or its documentation. Some examples that would benefit from an RFC are:

- A new feature that creates new API surface area, and would require a feature flag if introduced.
- The modification of an existing stable API.
- The removal of features that already shipped as part of Zephyr.
- The introduction of new idiomatic usage or conventions, even if they do not include code changes to Zephyr itself.

The RFC process is a great opportunity to get more eyeballs on proposals coming from contributors before it becomes a part of Zephyr. Quite often, even proposals that seem "obvious" can be significantly improved once a wider group of interested people have a chance to weigh in.

The RFC process can also be helpful to encourage discussions about a proposed feature as it is being designed, and incorporate important constraints into the design while it's easier to change, before the design has been fully implemented.

For a Major Feature, first open an issue and outline your proposal so that it can be discussed. This will also allow us to better coordinate our efforts, prevent duplication of work, and help you to craft the change so that it is successfully accepted into the project. Providing the following information will increase the chances of your issue being dealt with quickly:

- Overview of the Proposal
- Motivation for or Use Case
- Design Details
- Alternatives
- Test Strategy

Some changes or contributions do not require an RFC, the rationale and details of the changes should however be part of the pull-request:

- Small enhancements and modifications to existing and established subsystems.
- Rephrasing, reorganizing or refactoring
- Addition or removal of warnings
- Addition of new boards, SoCs or drivers to existing subsystems
- ...

The process in itself consists in creating a GitHub issue with the *RFC label* that documents the proposal thoroughly. You are encouraged to use the RFC form to make sure the proposal follows a template which project participants are already familiar with.

As with Pull Requests, RFCs might require discussion in the context of one of the Zephyr meetings in order to move it forward in cases where there is either disagreement or not enough voiced opinions in order to proceed. Make sure to either label it appropriately or include it in the corresponding GitHub project in order for it to be examined during the next meeting.

7.1.7 Modifying Contributions made by other developers

Scenarios

Zephyr contributors and collaborators are encouraged to assist as reviewers in pull requests, so that patches may be approved and merged to Zephyr's main branch as part of the original pull requests. The authors of the pull requests are responsible for amending their original commits following the review process.

There are occasions, however, when a contributor might need to modify patches included in pull requests that are submitted by other Zephyr contributors. For instance, this is the case when:

- a developer cherry-picks commits submitted by other contributors into their own pull requests in order to:
  - integrate useful content which is part of a stale pull request, or
  - get content merged to the project's main branch as part of a larger patch
- a developer pushes to a branch or pull request opened by another contributor in order to:
  - assist in updating pull requests in order to get the patches merged to the project's main branch
  - drive stale pull requests to completion so they can be merged

Accepted policies

A developer who intends to cherry-pick and potentially modify patches sent by another contributor shall:

- clarify in their pull request the reason for cherry-picking the patches, instead of assisting in getting the patches merged in their original pull request, and
- invite the original author of the patches to their pull request review.

A developer who intends to force-push to a branch or pull request of another Zephyr contributor shall clarify in the pull request the reason for pushing and for modifying the existing patches (e.g. stating that it is done to drive the pull request review to completion, when the pull request author is not able to do so).

> **Note**
>
> Developers should try to limit the above practice to pull requests identified as *stale*. Read about how to identify pull requests as stale in *development processes and tools*

If the original patches are substantially modified, the developer can either:

- (preferably) reach out to the original author and request them to acknowledge that the modified patches may be merged while having the original sign-off line and author identity, or
- submit the modified patches as their *own* work (i.e. with their *own* sign-off line and author identity). In this case, the developer shall identify in the commit message(s) the original source the submitted work is based on (mentioning, for example, the original PR number).

> **Note**
>
> Contributors should uncheck the box *"Allow Edits By Maintainers"* to indicate that they do not wish their patches to be amended, inside their original branch or pull request, by other Zephyr developers.

7.1.8 Pull Request Lifecycle Policy

This policy keeps the open pull request list focused on contributions that are actively progressing and have a realistic path to merge.

Goals

- Keep open pull requests manageable for contributors and maintainers.
- Prioritize review bandwidth for changes that are moving forward.
- Provide clear expectations for draft pull requests and inactive pull requests.

Scope

This policy applies to all pull requests in the repository, including draft pull requests.

Definitions

Active pull request
: A pull request with meaningful progress such as commits, review responses, or updates addressing requested changes.

Draft pull request
: A pull request opened for work in progress and early feedback.

Stalled pull request
: A pull request without meaningful activity inside the inactivity window.

Closed inactive pull request
: A pull request closed due to inactivity, supersession, or no clear merge path.

Draft Pull Request Expectations

- Draft pull requests should include a clear problem statement, current status, and known gaps.
- Draft pull requests should move to ready-for-review within 30 days.
- If a draft pull request is inactive for 30 days, it may be marked stale and a reminder is posted.
- If there is no meaningful progress for 7 additional days after stale notice, the pull request may be closed.

Ready-for-Review Pull Request Expectations

- Authors should respond to review feedback within 21 days.
- If a ready pull request has no meaningful activity for 30 days, it may be marked stale.
- If no meaningful update is made within 7 days after stale notice, the pull request may be closed.
- Pull requests with unresolved blocking feedback for more than 45 days may be closed if no concrete plan is provided.

What Counts as Meaningful Activity

Examples of meaningful activity include:

- New commits that address review feedback.
- Concrete technical responses and follow-up updates.

- Significant revision updates that materially change the pull request.

Examples that typically do not count as meaningful activity include:

- Ping-only comments with no technical update.
- Trivial rebases with no substantive change.

Exceptions

The following pull requests may be exempt from automatic inactive closure:

- Security-critical fixes.
- Release blockers.
- Pull requests explicitly waiting on an external dependency with a linked tracking item.

Exempt pull requests still need a status update at least every 30 days.

Superseded Pull Requests

When work is replaced by another pull request:

- Close the old pull request as superseded.
- Link the replacement pull request.
- Add a short summary comment preserving context.

Reopening Closed Inactive Pull Requests

Closing a pull request due to inactivity is an administrative step to keep the open list manageable. It is not a rejection of the contribution.

Closed inactive pull requests may be reopened when:

- The author posts an updated plan.
- Prior blocking feedback is addressed.
- Maintainers confirm that review can continue.

Contributors are encouraged to reopen at any time when they can provide meaningful updates.

Responsibilities

Contributors are expected to:

- Keep pull request descriptions up to date.
- Respond to review comments in a timely manner.
- Move draft pull requests to ready-for-review once they are reviewable.

Consult with the *contributor expectations* document for additional guidance.

Maintainers are expected to:

- Apply stale and closure decisions consistently.
- Provide clear blocking feedback.
- Close with explicit rationale and a reopen path.

Consult with the *maintainer responsibilities* document for additional guidance.

Contribution Guidelines
Learn about the overall process and guidelines for contributing to the Zephyr project.

This page is a mandatory read for first-time contributors as it contains important information on how to ensure your contribution can be considered for inclusion in the project and potentially merged.

Contributor Expectations
This document is another mandatory read that describes the expected behavior of *all* contributors to the project.

Reviewer Expectations
This document is another mandatory read that describes the expected behavior when reviewing contributions to the project.

Coding Guidelines
Code contributions are expected to follow a set of coding guidelines to ensure consistency and readability across the code base.

Coding Style Guidelines
Code contributions are expected to follow a set of style guidelines to ensure consistency and readability across the code base.

Proposals and RFCs
Learn when and how to submit RFCs (Request for Comments) for new features and changes to the project.

Modifying Contributions made by other developers
Guidelines for modifying contributions made by other developers and how to deal with stale pull requests.

Pull Request Lifecycle Policy
Policy for keeping open pull requests focused on work that is actively progressing and likely to merge.

7.2 Documentation

The Zephyr project thrives on good documentation. Whether it is as part of a code contribution or as a standalone effort, contributing documentation is particularly valuable to the project.

7.2.1 Documentation Guidelines

Note

For instructions on building the documentation, see *Documentation Generation*.

Zephyr Project content is written using the reStructuredText[2] markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted standalone website. Developers can view this content either in its raw form as .rst markup files, or (with Sphinx installed) they can *build the documentation* locally to generate the documentation in HTML or PDF format. The HTML content can then be viewed using a web browser. This same .rst content is served by the Zephyr documentation[4] website.

You can read details about reStructuredText[2] and about Sphinx extensions[1] from their respective websites.

[2] https://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html
[4] https://docs.zephyrproject.org
[1] https://www.sphinx-doc.org/en/stable/contents.html

This document provides a quick reference for commonly used reST and Sphinx-defined directives and roles used to create the documentation you're reading.

For instructions regarding writing good C API documentation, see *Doxygen Style Guidelines*.

Content Structure

Tabs, spaces, and indenting Indenting is significant in reST file content, and using spaces is preferred. Extra indenting can (unintentionally) change the way content is rendered too. For lists and directives, indent the content text to the first non-white space in the preceding line. For example:

```
* List item that spans multiple lines of text
  showing where to indent the continuation line.

1. And for numbered list items, the continuation
   line should align with the text of the line above.

.. code-block::

   The text within a directive block should align with the
   first character of the directive name.
```

Refer to the Zephyr *Coding Style Guidelines* for additional requirements.

Headings While reST allows use of both an overline and matching underline to indicate a heading, we only use an underline indicator for headings.

- Document title (h1) use # for the underline character
- First section heading level (h2) use *
- Second section heading level (h3) use =
- Third section heading level (h4) use -

The heading underline must be the same length as the heading text.

For example:

```
This is a title heading
#######################

some content goes here

First section heading
*********************
```

Lists For bullet lists, place an asterisk (*) or hyphen (-) at the start of a paragraph and indent continuation lines with two spaces.

The first item in a list (or sublist) must have a blank line before it and should be indented at the same level as the preceding paragraph (and not indented itself).

For numbered lists start with a 1. or a. for example, and continue with autonumbering by using a # sign. Indent continuation lines with three spaces:

```
* This is a bulleted list.
* It has two items, the second
  item and has more than one line of reST text.  Additional lines
  are indented to the first character of the
```

(continues on next page)

(continued from previous page)

```
  text of the bullet list.

1. This is a new numbered list. If the wasn't a blank line before it,
   it would be a continuation of the previous list (or paragraph).
#. It has two items too.

a. This is a numbered list using alphabetic list headings
#. It has three items (and uses autonumbering for the rest of the list)
#. Here's the third item

#. This is an autonumbered list (default is to use numbers starting
   with 1).

   #. This is a second-level list under the first item (also
      autonumbered).  Notice the indenting.
   #. And a second item in the nested list.
#. And a second item back in the containing list.  No blank line
   needed, but it wouldn't hurt for readability.
```

Definition lists (with a term and its definition) are a convenient way to document a word or phrase with an explanation. For example this reST content:

```
The Makefile has targets that include:

html
   Build the HTML output for the project

clean
   Remove all generated output, restoring the folders to a
   clean state.
```

Would be rendered as:

The Makefile has targets that include:

html
: Build the HTML output for the project

clean
: Remove all generated output, restoring the folders to a clean state.

Multi-column lists If you have a long bullet list of items, where each item is short, you can indicate the list items should be rendered in multiple columns with a special `.. rst-class:: rst-columns` directive. The directive will apply to the next non-comment element (e.g., paragraph), or to content indented under the directive. For example, this unordered list:

```
.. rst-class:: rst-columns

* A list of
* short items
* that should be
* displayed
* horizontally
* so it doesn't
* use up so much
* space on
* the page
```

would be rendered as:

- A list of
- short items
- that should be
- displayed
- horizontally
- so it doesn't
- use up so much
- space on
- the page

A maximum of three columns will be displayed, and change based on the available width of the display window, reducing to one column on narrow (phone) screens if necessary. We've deprecated use of the `hlist` directive because it misbehaves on smaller screens.

Tables There are a few ways to create tables, each with their limitations or quirks. Grid tables offer the most capability for defining merged rows and columns, but are hard to maintain:

```
+------------------------+------------+----------+----------+
| Header row, column 1   | Header 2   | Header 3 | Header 4 |
| (header rows optional) |            |          |          |
+========================+============+==========+==========+
| body row 1, column 1   | column 2   | column 3 | column 4 |
+------------------------+------------+----------+----------+
| body row 2             | ...        | ...      | you can  |
+------------------------+------------+----------+ easily   +
| body row 3 with a two column span   | ...      | span     |
+------------------------+------------+----------+ rows     +
| body row 4             | ...        | ...      | too      |
+------------------------+------------+----------+----------+
```

This example would render as:

Header row, column 1 (header rows optional)	Header 2	Header 3	Header 4
body row 1, column 1	column 2	column 3	column 4
body row 2	...	...	you can easily span rows too
body row 3 with a two column span		...	
body row 4	...	...	

List tables are much easier to maintain, but don't support row or column spans:

```
.. list-table:: Table title
   :widths: 15 20 40
   :header-rows: 1

   * - Heading 1
     - Heading 2
     - Heading 3
   * - body row 1, column 1
     - body row 1, column 2
     - body row 1, column 3
```

(continues on next page)

(continued from previous page)

```
  * - body row 2, column 1
    - body row 2, column 2
    - body row 2, column 3
```

This example would render as:

Table 4: Table title

Heading 1	Heading 2	Heading 3
body row 1, column 1	body row 1, column 2	body row 1, column 3
body row 2, column 1	body row 2, column 2	body row 2, column 3

The `:widths:` parameter lets you define relative column widths. The default is equal column widths. If you have a three-column table and you want the first column to be half as wide as the other two equal-width columns, you can specify `:widths: 1 2 2`. If you'd like the browser to set the column widths automatically based on the column contents, you can use `:widths: auto`.

Tabbed Content As introduced in the *Getting Started Guide*, you can provide alternative content to the reader via a tabbed interface. When the reader clicks on a tab, the content for that tab is displayed, for example:

```
.. tabs::

   .. tab:: Apples

      Apples are green, or sometimes red.

   .. tab:: Pears

      Pears are green.

   .. tab:: Oranges

      Oranges are orange.
```

will display as:

Apples

Apples are green, or sometimes red.

Pears

Pears are green.

Oranges

Oranges are orange.

Tabs can also be grouped, so that changing the current tab in one area changes all tabs with the same name throughout the page. For example:

Linux

Linux Line 1

macOS

macOS Line 1

Windows

Windows Line 1

Linux

Linux Line 2

macOS

macOS Line 2

Windows

Windows Line 2

In this latter case, we're using `.. group-tab::` instead of simply `.. tab::`. Under the hood, we're using the sphinx-tabs extension that's included in the Zephyr setup. Within a tab, you can have most any content *other than a heading* (code-blocks, ordered and unordered lists, pictures, paragraphs, and such). You can read more about sphinx-tabs from the link above.

Text Formatting

ReSTructuredText supports a variety of text formatting options. This section provides a quick reference for some of the most commonly used text formatting options in Zephyr documentation. For an exhaustive list, refer to the reStructuredText Quick Reference[5], reStructuredText Interpreted Text Roles[6] as well as the additional roles provided by Sphinx[7].

Content Highlighting Some common reST inline markup samples:

- one asterisk: `*text*` for emphasis (*italics*),
- two asterisks: `**text**` for strong emphasis (**boldface**), and
- two backquotes: `` ``text`` `` for `inline code` samples.

If asterisks or backquotes appear in running text and could be confused with inline markup delimiters, you can eliminate the confusion by adding a backslash (\) before it.

File Names and Commands Sphinx extends reST by supporting additional inline markup elements (called "roles") used to tag text with special meanings and allow style output formatting. (You can refer to the Sphinx Inline Markup[3] documentation for the full list).

While double quotes can be used for rendering text as "code", you are encouraged to use the following roles for marking up file names, command names, and other "special" text.

- `file` for file names, e.g., `:file:`CMakeLists.txt`` will render as `CMakeLists.txt`

> **Note**
>
> In case you want to indicate a "variable" file path, you may use curly braces to enclose the variable part of the path, e.g., `:file:`{boardname}_defconfig`` will render as *boardname*`_defconfig`.

- `command` for command names, e.g., `:command:`make`` will render as **`make`**
- `envvar` for environment variables, e.g., `:envvar:`ZEPHYR_BASE`` will render as *`ZEPHYR_BASE`*

For creating references to files that are hosted in the Zephyr organization on GitHub, refer to *Cross-referencing files in the Zephyr tree* section below.

[5] https://docutils.sourceforge.io/docs/user/rst/quickref.html
[6] https://docutils.sourceforge.io/docs/ref/rst/roles.html
[7] https://www.sphinx-doc.org/en/master/usage/restructuredtext/roles.html
[3] https://sphinx-doc.org/markup/inline.html#inline-markup

User Interaction When documenting user interactions, such as key combinations or GUI interactions, use the following roles to highlight the commands in a meaningful way:

- `kbd` for keyboard input, e.g., `:kbd:`Ctrl-C`` will render as `Ctrl-C`
- `menuselection` for menu selections, e.g., `:menuselection:`File --> Open`` will render as *File ▸ Open*
- `guilabel` for GUI labels, e.g., `:guilabel:`Cancel`` will render as *Cancel*

Mathematical Formulas You can include mathematical formulas using either the `math` role or `math` directive. The directive provides more flexibility in case you have a more complex formula.

The input language for mathematics is LaTeX markup. Example:

```
The answer to life, the universe, and everything is :math:`30 + 2^2 + \sqrt{64} =
↪42`.
```

This would render as:

> The answer to life, the universe, and everything is $30 + 2^2 + \sqrt{64} = 42$.

Non-ASCII Characters Prefer plain ASCII unless a specific symbol is required for correctness or conventional typography (for example units like µ, or well-known marks like ™).

Avoid adding non-ASCII characters purely for aesthetic purposes.

The file doc/substitutions.txt contains some basic HTML substitution definitions for special formatting needs (e.g. to force line breaks), but Unicode characters can and should be used directly in the documentation source files.

Code Blocks and Command Examples Use the reST `code-block` directive to create a highlighted block of fixed-width text, typically used for showing formatted code or console commands and output. Smart syntax highlighting is also supported (using the Pygments package). You can also directly specify the highlighting language. For example:

```
.. code-block:: c

   struct k_object {
      char *name;
      uint8_t perms[CONFIG_MAX_THREAD_BYTES];
      uint8_t type;
      uint8_t flags;
      uint32_t data;
   } __packed;
```

Note the blank line between the `code-block` directive and the first line of the code-block body, and the body content is indented three spaces (to the first non-white space of the directive name).

This would be rendered as:

```c
struct k_object {
   char *name;
   uint8_t perms[CONFIG_MAX_THREAD_BYTES];
   uint8_t type;
   uint8_t flags;
   uint32_t data;
} __packed;
```

Other languages are of course supported (see languages supported by Pygments[8]), and in particular, you are encouraged to make use of the following when appropriate:

[8] https://pygments.org/languages/

- `c` for C code
- `cpp` for C++ code
- `python` for Python code
- `console` for console output, i.e. interactive shell sessions where commands are prefixed by a prompt (ex. `$` for Linux, or `uart:~$` for Zephyr's shell), and where the output is also shown. The commands will be highlighted, and the output will not. What's more, copying code block using the "copy" button will automatically copy just the commands, excluding the prompt and the outputs of the commands.
- `shell` or `bash` for shell commands. Both languages get highlighted the same but you may use `bash` for conveying that the commands are bash-specific, and `shell` for generic shell commands.

Note

Do not use `bash` or `shell` if your code block includes a prompt, use `console` instead.

Reciprocally, do not use `console` if your code block does not include a prompt and is not showcasing an interactive session with command(s) and their output.

Table 5: When to use `bash`/`shell` vs. `console`

Use case	code-block snippet	Expected output

One or several commands, no output	`.. code-block:: shell`
	
	`echo "Hello World!"`
		`echo "Hello World!"`

An interactive shell session with command(s) and their output	`.. code-block:: console`
	
	`$ echo "Hello World!"`
	`Hello World!`
		`$ echo "Hello World!"`
		`Hello World!`

An interactive Zephyr shell session, with commands and their outputs	`.. code-block:: console`
	
	`uart:~$ version`
	`Zephyr version 3.5.99`
	`uart:~$ kernel uptime`
	`Uptime: 20970 ms`
		`uart:~$ version`
		`Zephyr version 3.5.99`
		`uart:~$ kernel uptime`
		`Uptime: 20970 ms`

- `bat` for Windows batch files
- `cfg` for config files with "KEY=value" entries (ex. Kconfig `.conf` files)
- `cmake` for CMake
- `devicetree` for Devicetree
- `kconfig` for Kconfig
- `yaml` for YAML
- `rst` for reStructuredText

When no language is specified, the language is set to `none` and the code block is not highlighted. You may also use `none` explicitly to achieve the same result; for example:

```
.. code-block:: none

   This would be a block of text styled with a background
   and box, but with no syntax highlighting.
```

Would display as:

```
This would be a block of text styled with a background
and box, but with no syntax highlighting.
```

There's a shorthand for writing code blocks too: end the introductory paragraph with a double colon (`::`) and indent the code block content that follows it by three spaces. On output, only one colon will be shown. The code block will have no highlighting (i.e. `none`). You may however use the `highlight` directive to customize the default language used in your document (see for example how this is done at the beginning of this very document).

Links and Cross-References

Cross-referencing internal content Traditional ReST links are only supported within the current file using the notation:

```
Refer to the `internal-linking`_ page
```

which renders as,

> Refer to the *internal-linking* page

Note the use of a trailing underscore to indicate an outbound link. In this example, the label was added immediately before a heading, so the text that's displayed is the heading text itself. You can change the text that's displayed as the link writing this as:

```
Refer to the `show this text instead <internal-linking_>`_ page
```

which renders as,

> Refer to the *show this text instead* page

Cross-referencing external content With Sphinx's help, we can create link-references to any tagged text within the Zephyr Project documentation.

Target locations in a document are defined with a label directive:

```
.. _my label name:

Heading
=======
```

Note the leading underscore indicating an inbound link. The content immediately following this label must be a heading, and is the target for a `:ref:`my label name`` reference from anywhere within the Zephyr documentation. The heading text is shown when referencing this label. You can also change the text that's displayed for this link, such as:

```
:ref:`some other text <my label name>`
```

To enable easy cross-page linking within the site, each file should have a reference label before its title so it can be referenced from another file. These reference labels must be unique across the whole site, so generic names such as "samples" should be avoided. For example the top of this document's .rst file is:

```
.. _doc_guidelines:

Documentation Guidelines for the Zephyr Project
###############################################
```

Other .rst documents can link to this document using the `:ref:`doc_guidelines` tag and it will show up as *Documentation Guidelines*. This type of internal cross reference works across multiple files, and the link text is obtained from the document source so if the title changes, the link text will update as well.

You can also define links to any URL and then reference it in your document. For example, with this label definition in the document:

```
.. _Zephyr Wikipedia Page:
   https://en.wikipedia.org/wiki/Zephyr_(operating_system)
```

you can reference it with:

```
Read the `Zephyr Wikipedia Page`_ for more information about the
project.
```

Tip

When a document contains many external links, it can be useful to list them in a single "References" section at the end of the document. This can be done using the `target-notes` directive. Example:

```
References
==========

.. target-notes::

.. _external_link1: https://example.com
.. _external_link2: https://example.org
```

Cross-referencing C documentation

`:c:member:`

`:c:data:`

`:c:var:`

`:c:func:`

`:c:macro:`

`:c:struct:`

`:c:union:`

`:c:enum:`

`:c:enumerator:`

`:c:type:`

You may use these roles to cross-reference the Doxygen documentation of C functions, macros, types, etc.

They are rendered in the HTML output as links to the corresponding Doxygen documentation for the item. For example:

```
Check out :c:func:`gpio_pin_configure` for more information.
```

Will render as:

Check out `gpio_pin_configure()` for more information.

You may provide a custom link text, similar to the built-in `ref` role.

Cross-referencing CMake documentation You may use the following roles to cross-reference the documentation of Zephyr's CMake modules, commands, and variables.

`:cmake:module:`

This role is used to reference a CMake module. For example:

```
See :cmake:module:`extensions` for more information.
```

Will render as:

See *extensions* for more information.

`:cmake:command:`

This role is used to reference a CMake command. For example:

```
See :cmake:command:`yaml_load` for more information.
```

Will render as:

See *yaml_load()* for more information.

Commands documented by CMake itself are referenced through their fully qualified name, given as an explicit link target:

```
See :cmake:command:`target_sources <command:target_sources>` for more␣
↪information.
```

Will render as:

See `target_sources` for more information.

`:cmake:variable:`

This role is used to reference a CMake variable. For example:

```
See :cmake:variable:`CMAKE_C_COMPILER` for more information.
```

Will render as:

See `CMAKE_C_COMPILER` for more information.

Visual Elements

Images Images are included in the documentation by using an `image` directive:

```
.. image:: ../../../../images/doc-gen-flow.png
   :align: center
   :alt: alt text for the image
```

or if you'd like to add an image caption, use:

```
.. figure:: ../../../../images/doc-gen-flow.png
   :alt: image description

   Caption for the figure
```

The file name specified is relative to the document source file, and we recommend putting images into an `images` folder where the document source is found.

The usual image formats handled by a web browser are supported: WebP, PNG, GIF, JPEG, and SVG.

Keep the image size only as large as needed, generally at least 500 px wide but no more than 1000 px, and no more than 100 KB unless a particularly large image is needed for clarity.

Recommended image formats based on content

- **Screenshots**: WebP or PNG.
- **Diagrams**: Consider using Graphviz for simple diagrams (see dedicated section below. If using an external tool, SVG is preferred.
- **Photos** (ex. boards): WebP, no larger than 600 px on the largest dimension. Whenever the subject can be isolated from its background (typically the case for board photos), save the image with a transparent background so it blends in with both light and dark documentation themes.

  You can convert an existing image to a properly sized WebP using cwebp[9] or ImageMagick[10]. For example:

  ```
  # Using cwebp (resize width to 600 px, height auto, ~80% quality).
  # For a portrait image, use "-resize 0 600" to cap the height instead.
  cwebp -resize 600 0 board_name.png -o board_name.webp

  # Using ImageMagick
  magick board_name.png -resize 600x600 -quality 80 board_name.webp
  ```

  When the source already has a transparent background (e.g. a PNG with an alpha channel), both tools preserve transparency in the resulting WebP. The `-resize 600 0` / `600x600` arguments only scale the image down, preserving its aspect ratio.

Graphviz Graphviz[11] is a tool for creating diagrams specified in a simple text language. As it's important to allow for diagrams used in the documentation to be easily maintained, we encourage the use of Graphviz for creating diagrams. Graphviz is particularly well suited for creating state diagrams, flow charts, and other types of diagrams that can be expressed as a graph.

To include a Graphviz diagram in a document, use the `graphviz` directive. For example:

```
.. graphviz::
   :caption: An example graph using Graphviz

   digraph G {
      rankdir=LR;
      A -> B;
      B -> C;
      C -> D;
   }
```

Would render as:

Please refer to the Graphviz documentation[12] for more information on how to create diagrams using Graphviz's DOT language.

Mermaid Mermaid[13] is a tool for creating diagrams and visualizations using a simple text-based syntax. It is particularly well-suited for creating flowcharts, sequence diagrams, class diagrams, and state transition diagrams.

[9] https://developers.google.com/speed/webp/download
[10] https://imagemagick.org/
[11] https://graphviz.org
[12] https://graphviz.org/documentation
[13] https://mermaid.js.org/

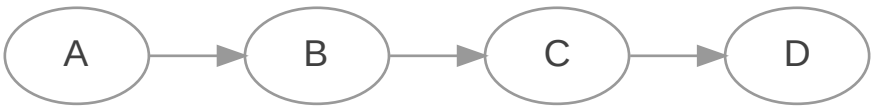


Fig. 1: An example graph using Graphviz

To include a mermaid diagram in a document, use the `mermaid` directive. For example:

```
.. mermaid::
   :caption: State transition diagram for a GPIO debounce
   :alt: GPIO debounce state diagram showing transitions between inactive and␣
↪active states
         through maybe_active and maybe_inactive intermediate states before each␣
↪state becomes
         stable.

   ---
   config:
     state:
       useMaxWidth: false
   ---
   stateDiagram-v2

       State inactive {
           [*] --> stable_inactive
           stable_inactive --> maybe_active : edge to active
           maybe_active --> stable_inactive : edge to inactive
       }

       State active {
           [*] --> stable_active
           stable_active --> maybe_inactive : edge to inactive
           maybe_inactive --> stable_active : edge to active
       }

       [*] --> inactive

       maybe_active --> active : After(x ms)
       maybe_inactive --> inactive : After(x ms)
```

Would render as:

Fig. 2: State transition diagram for a GPIO debounce

A diagram is drawn across the width of the page and its height follows from its aspect ratio, so a diagram that is taller than it is wide ends up much larger than it needs to be. Turning off `useMaxWidth`, as in the example above, keeps the diagram at the size Mermaid computed for it; it still shrinks to fit a narrow screen. The setting belongs to the diagram type, `state` here and `flowchart`, `sequence` or another type elsewhere.

For references about supported diagrams, syntax, and samples; please refer to the Mermaid doc-

umentation[14]. For fast iteration when creating or updating diagrams, you can use the Mermaid live editor[15].

Custom Sphinx Roles and Directives

The Zephyr documentation uses custom Sphinx roles and directives to provide additional functionality and to make it easier to write and maintain consistent documentation.

Application build commands

`.. zephyr-app-commands::`

Generate consistent documentation of the shell commands needed to manage (build, flash, etc.) an application

For example, to generate commands to build `samples/hello_world` for `qemu_x86` use:

```
.. zephyr-app-commands::
   :zephyr-app: samples/hello_world
   :board: qemu_x86
   :goals: build
```

This will render as:

```
# From the root of the zephyr repository
west build -b qemu_x86 samples/hello_world
```

Options

`:tool: (string)`

Which tool to use. Valid options are currently `cmake`, `west` and `all`. The default is `west`.

`:app: (string)`

Path to the application to build.

`:zephyr-app: (string)`

Path to the application to build, this is an app present in the upstream zephyr repository. Mutually exclusive with `:app:`.

`:cd-into: (no value)`

If set, build instructions are given from within the `:app:` folder, instead of outside of it.

`:generator: (string)`

Which build system to generate.

Valid options are currently `ninja` and `make`. The default is `ninja`. This option is not case sensitive.

`:host-os:`

Which host OS the instructions are for.

Valid options are `unix`, `win` and `all`. The default is `all`.

`:board: (string)`

If set, build commands will target the given board.

[14] https://mermaid.js.org/intro/
[15] https://mermaid.live/

`:shield: (string)`
If set, build commands will target the given shield.

Multiple shields can be provided in a comma separated list.

`:conf:`
If set, build commands will use the given configuration file(s).

If multiple configuration files are provided, enclose the space-separated list of files with double quotes, e.g., *"a.conf b.conf"*.

`:gen-args: (string)`
If set, indicates additional arguments to the CMake invocation.

`:build-args: (string)`
If set, indicates additional arguments to the build invocation.

`:west-args: (string)`
If set, additional arguments to the west invocation (ignored for `:tool: cmake`).

`:flash-args: (string)`
If set, additional arguments to the flash invocation.

`:debug-args: (string)`
If set, additional arguments to the debug invocation.

`:debugserver-args: (string)`
If set, additional arguments to the debugserver invocation.

`:attach-args: (string)`
If set, additional arguments to the attach invocation.

`:snippets: (string)`
If set, indicates the application should be compiled with the listed snippets.

Multiple snippets can be provided in a comma separated list.

`:build-dir: (string)`
If set, the application build directory will *APPEND* this relative, Unix-separated, path to the standard build directory. This is mostly useful for distinguishing builds for one application within a single page.

`:build-dir-fmt: (string)`
If set, assume that *west config build.dir-fmt*' has been set to this path.

Exclusive with `:build-dir:` and depends on `:tool: west`.

`:goals: (string)`
A whitespace-separated list of what to do with the app (any of `build`, `flash`, `debug`, `debugserver`, `run`).

Commands to accomplish these tasks will be generated in the right order.

`:maybe-skip-config: (no value)`
If set, this indicates the reader may have already created a build directory and changed there, and will tweak the text to note that doing so again is not necessary.

`:compact: (no value)`
If set, the generated output is a single code block with no additional comment lines.

Cross-referencing files in the Zephyr tree Special roles are available to reference files in the Zephyr tree. For example, referencing this very file can be done using the *zephyr_file* role.

`:zephyr_file:`

This role is used to reference a file in the Zephyr tree. For example:

```
Check out :zephyr_file:`doc/contribute/documentation/guidelines.rst` for more␣
↪information.
```

Will render as:

> Check out doc/contribute/documentation/guidelines.rst for more information.

You can reference specific lines or line ranges in a file by appending `#L`*line_number* or `#L`*start_line*`-L`*end_line* to the file path:

```
See :zephyr_file:`doc/contribute/documentation/guidelines.rst#L3` for the␣
↪main heading of
this document.
```

Will render as:

> See doc/contribute/documentation/guidelines.rst#L3 for the main heading of this document.

The role automatically verifies that the referenced file exists in the Zephyr tree and will generate a warning during documentation build if the file is not found.

> **Note**
>
> Use the line references sparingly as keeping them accurate over time can be challenging as the content of the linked file is subject to change.

You may use the *zephyr_raw* role instead if you want to reference the "raw" content.

`:zephyr_raw:`

This role is used to reference the raw content of a file in the Zephyr tree. For example:

```
Check out :zephyr_raw:`doc/contribute/documentation/guidelines.rst` for more␣
↪information.
```

Will render as:

> Check out doc/contribute/documentation/guidelines.rst for more information.

`:module_file:`

This role is used to reference a module in the Zephyr tree. For example:

```
Check out :module_file:`hal_stm32:CMakeLists.txt` for more information.
```

Will render as:

> Check out hal_stm32:CMakeLists.txt for more information.

Similar to *zephyr_file*, you can reference specific lines or line ranges in a file.

Cross-referencing GitHub issues and pull requests

`:github:`

This role is used to reference a GitHub issue or pull request.

For example, to reference issue #1234:

```
Check out :github:`1234` for more background about this known issue.
```

This will render as:

Check out GitHub #1234 for more background about this known issue.

Doxygen API documentation

`.. doxygengroup:: name`

This directive is used to output a short description of a Doxygen group and a link to the corresponding Doxygen-generated documentation.

All the code samples (declared using the *`zephyr:code-sample`* directive) indicating the group as relevant will automatically be list and referenced in the rendered output.

For example:

```
.. doxygengroup:: can_interface
```

Will render as:

CAN

Options

`:project: (project name (optional))`

Associated Doxygen project. This can be useful when multiple Doxygen projects are configured.

`:c:group:`

This role is used to reference a Doxygen group in the Zephyr tree. In the HTML documentation, they are rendered as links to the corresponding Doxygen-generated documentation for the group. For example:

```
Check out :c:group:`gpio_interface` for more information.
```

Will render as:

Check out GPIO for more information.

You may provide a custom link text, similar to the built-in `ref` role.

Kconfig options If you want to reference a Kconfig option from a document, you can use the *`kconfig:option`* role and provide the name of the option you want to reference. The role will automatically generate a link to the documentation of the Kconfig option when building HTML output.

Make sure to use the full name of the Kconfig option, including the `CONFIG_` prefix.

`:kconfig:option:`

This role is used to reference a Kconfig option in the Zephyr tree. For example:

```
Check out :kconfig:option:`CONFIG_GPIO` for more information.
```

Will render as:

Check out `CONFIG_GPIO` for more information.

`:kconfig:option-regex:`

This role is used to create links to regex searches for Kconfig options. It generates a link to the Kconfig search page with the provided regex pattern automatically filled in as the search query. It is useful for referencing multiple Kconfig options that share a common prefix, or belong to a common category. For example:

```
Check out␣
↪:kconfig:option-regex:`CONFIG_SECURE_STORAGE_ITS_(STORE|TRANSFORM)_.*_CUSTOM`␣
↪for
the various customization possibilities.
```

Will render as:

> Check out CONFIG_SECURE_STORAGE_ITS_(STORE|TRANSFORM)_.*_CUSTOM for the various customization possibilities.

It is encouraged to provide a custom link text to make the reference more readable. For example:

```
Check out the :kconfig:option-regex:`ITS Kconfig options
↪<CONFIG_SECURE_STORAGE_ITS_.*>`
for more information.
```

Will render as:

> Check out the ITS Kconfig options for more information.

Devicetree bindings If you want to reference a Devicetree binding from a document, you can use the *dtcompatible* role and provide the compatible string of the binding you want to reference. The role will automatically generate a link to the documentation of the binding when building HTML output.

:dtcompatible:

This role can be used inline to make a reference to the generated documentation for the Devicetree compatible given as argument.

There may be more than one page for a single compatible. For example, that happens if a binding behaves differently depending on the bus the node is on. If that occurs, the reference points at a "disambiguation" page which links out to all the possibilities, similarly to how Wikipedia disambiguation pages work. Example:

```
Check out :dtcompatible:`zephyr,input-longpress` for more information.
```

Will render as:

> Check out zephyr,input-longpress for more information.

Code samples

.. zephyr:code-sample:: id

This directive is used to describe a code sample, including which noteworthy APIs it may be exercising.

For example:

```
.. zephyr:code-sample:: blinky
   :name: Blinky
   :relevant-api: gpio_interface

   Blink an LED forever using the GPIO API.
```

The content of the directive is used as the description of the code sample.

Options

:name: (text)

Indicates the human-readable short name of the sample.

`:relevant-api: (text)`
Optional space-separated list of Doxygen group names that correspond to the APIs exercised by the code sample.

`:zephyr:code-sample:`
This role is used to reference a code sample described using *zephyr:code-sample*.

For example:

```
Check out :zephyr:code-sample:`blinky` for more information.
```

Will render as:

Check out blinky for more information.

This can be used exactly like the built-in `ref` role, i.e. you may provide a custom link text. For example:

```
Check out :zephyr:code-sample:`blinky code sample <blinky>` for more␣
↪information.
```

Will render as:

Check out blinky code sample for more information.

`.. zephyr:code-sample-category:: id`
This directive is used to define a category for grouping code samples.

For example:

```
.. zephyr:code-sample-category:: gpio
   :name: GPIO
   :show-listing:

   Samples related to the GPIO subsystem.
```

The contents of the directive is used as the description of the category. It can contain any valid reStructuredText content.

Options

`:name: (text)`
Indicates the human-readable name of the category.

`:show-listing: (flag)`
If set, a listing of code samples in the category will be shown. The listing is automatically generated based on all code samples found in the subdirectories of the current document.

`:glob: (text)`
A glob pattern to match the files to include in the listing. The default is */* but it can be overridden e.g. when samples may be found in directories not sitting directly under the category directory.

`:zephyr:code-sample-category:`
This role is used to reference a code sample category described using *zephyr:code-sample-category*.

For example:

```
Check out :zephyr:code-sample-category:`cloud` samples for more information.
```

Will render as:

Check out cloud samples for more information.

`.. zephyr:code-sample-listing::`

This directive is used to show a listing of all code samples found in one or more categories.

For example:

```
.. zephyr:code-sample-listing::
   :categories: cloud
```

Will render as:

- AWS IoT Core MQTTConnect to AWS IoT Core and publish messages using MQTT.
- Microsoft Azure IoT Hub MQTTConnect to Azure IoT Hub and publish messages using MQTT.
- TagoIO HTTP PostSend random temperature values to TagoIO IoT Cloud Platform using HTTP.

Options

`:categories: (text)`

A space-separated list of category IDs for which to show the listing.

`:live-search: (flag)`

A flag to include a search box right above the listing. The search box allows users to filter the listing by code sample name/description, which can be useful for categories with a large number of samples. This option is only available in the HTML builder.

Boards

`.. zephyr:board:: name`

This directive is used at the beginning of a document to indicate it is the main documentation page for a board whose name is given as the directive argument.

For example:

```
.. zephyr:board:: wio_terminal
```

The metadata for the board is read from various config files and used to automatically populate some sections of the board documentation. A board documentation page that uses this directive can be linked to using the *zephyr:board* role.

`:zephyr:board:`

This role is used to reference a board documented using *zephyr:board*.

For example:

```
Check out :zephyr:board:`wio_terminal` for more information.
```

Will render as:

Check out wio_terminal for more information.

`.. zephyr:board-catalog::`

This directive is used to generate a catalog of Zephyr-supported boards that can be used to quickly browse the list of all supported boards and filter them according to various criteria.

`:zephyr:board-catalog:`

This role is used to reference the board catalog page, optionally with filter parameters. For example:

```
Check out :zephyr:board-catalog:`` for more information.
```

Will render as:

> Check out :zephyr:board-catalog:“ for more information.

This role can be used exactly like the built-in `ref` role, i.e. you may provide a custom link text. For example:

```
Check out the :zephyr:board-catalog:`boards using this compatible <
↪#compatibles=ti,hdc2080>`
for more information.
```

Will render as:

> Check out the boards using this compatible for more information.

`.. zephyr:board-supported-hw::`

This directive is used to show supported hardware features for all the targets of the board documented in the current page. The tables are automatically generated based on the board's Devicetree.

The directive must be used in a document that also contains a *`zephyr:board`* directive, as it relies on the board information to generate the table.

> **Note**
>
> This directive requires that the documentation is built with hardware features generation enabled (`zephyr_generate_hw_features` config option set to `True`). If disabled, a warning message will be shown instead of the hardware features tables.
>
> It is possible to limit the hardware features generation to boards from a specific list of vendors to speed up documentation builds without completely disabling the hardware features table. Set the config option `zephyr_hw_features_vendor_filter` to the list of vendors to generate features for. If the option is empty, hardware features are generated for all boards from all vendors.
>
> The config option `zephyr_hw_features_twister_extra_flags` can be used to provide additional flags to the twister command.

`.. zephyr:board-supported-runners::`

This directive is used to show the supported runners for the board documented in the current page, including which runner is the default for flashing and debugging.

The directive must be used in a document that also contains a *`zephyr:board`* directive, as it relies on the board information to generate the table.

> **Note**
>
> Similar to *`zephyr:board-supported-hw`*, this directive requires hardware features generation to be enabled (`zephyr_generate_hw_features` config option set to `True`) to produce a complete table. If disabled, a warning message will be shown instead of the runners tables.

Accessibility Guidelines

Accessibility is an important aspect of documentation, ensuring that all users, including those with disabilities, can access and understand the content.

When writing and maintaining Zephyr Project documentation, please follow these guidelines to improve accessibility for everyone.

Images and Figures All images and figures must include appropriate alternative text (alt text) to convey the meaning of the visual content to users who rely on screen readers or cannot view images.

- Use the `:alt:` attribute when including images using the `image` directive. Example:

```
.. image:: ../../../../contribute/documentation/image/doc-gen-flow.png
   :alt: Documentation generation process overview
```

- If the image contains text, ensure that the alt text includes this text verbatim.
- When using the `figure` directive, which allows for a caption, the `:alt:` text is still important. The alt text should describe the image itself, while the caption provides additional context or interpretation. Example:

```
.. figure:: ../../../../images/arch-diagram.png
   :alt: High-level overview of Zephyr OS architecture showing layers and␣
↪components.

   High-level overview of Zephyr OS architecture.
```

- Avoid using images as the sole method of conveying information that can be explained clearly with text.

Best Practices for writing alt text

- **Be Accurate and Equivalent**: Present the same essential information as the image.
- **Be Succinct**: Convey the core message of the image concisely.
- **Avoid Redundancy**: Do not use phrases like "Image of..." or "Picture of..." as screen readers typically announce the element as an image.
- **Describe, Don't Interpret**: Stick to describing what is visually present on the image.
- **Complex Images**: For charts, diagrams, or other complex visuals, provide a summary in the alt text. If a full understanding requires more detail, consider providing a more detailed description in the surrounding text or as part of the figure caption. Using text-based diagram tools like *Graphviz* can also improve accessibility.

Headings and Structure Use *headings* to structure your document logically. This allows users of assistive technologies to understand the document's organization and navigate it efficiently.

Tables Tables should be used only for tabular data and must be accessible to screen readers.

- Always define headers for rows and columns.
- Use the `list-table` directive when possible for better responsiveness and accessibility.
- Include a caption for tables where context is not immediately obvious. Example:

```
.. list-table:: GPIO Pin Configuration Options
   :widths: 15 30
   :header-rows: 1

   * - Field
     - Description
```

(continues on next page)

(continued from previous page)

```
  * - GPIO_INPUT
    - Configures pin as input
  * - GPIO_OUTPUT
    - Configures pin as output
```

Additional Resources For more general guidance on web accessibility you may refer to W3C's Web Content Accessibility Guidelines (WCAG)[16]

References

7.2.2 Documentation Generation

These instructions will walk you through generating the Zephyr Project's documentation on your local system using the same documentation sources as we use to create the online documentation found at https://docs.zephyrproject.org

Documentation overview

Zephyr Project content is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted stand-alone website. Developers can view this content either in its raw form as .rst markup files, or you can generate the HTML content and view it with a web browser directly on your workstation. This same .rst content is also fed into the Zephyr Project's public website documentation area (with a different theme applied).

You can read details about reStructuredText, and Sphinx from their respective websites.

The project's documentation contains the following items:

- ReStructuredText source files used to generate documentation found at the https://docs.zephyrproject.org website. Most of the reStructuredText sources are found in the `/doc` directory, but others are stored within the code source tree near their specific component (such as `/samples` and `/boards`)
- Doxygen-generated material used to create all API-specific documents also found at https://docs.zephyrproject.org
- Script-generated material for kernel configuration options based on Kconfig files found in the source code tree

The reStructuredText files are processed by the Sphinx documentation system, and make use of the doxygen-generated API material. Additional tools are required to generate the documentation locally, as described in the following sections.

Installing the documentation processors

Our documentation processing has been tested to run with:

- Doxygen version 1.17.0
- Graphviz 2.43
- Latexmk version 4.83
- All Python dependencies listed in the repository file `doc/requirements.txt`

In order to install the documentation tools, first install Zephyr as described in *Getting Started Guide*. Then install additional tools that are only required to generate the documentation, as described below:

[16] https://www.w3.org/WAI/standards-guidelines/wcag/

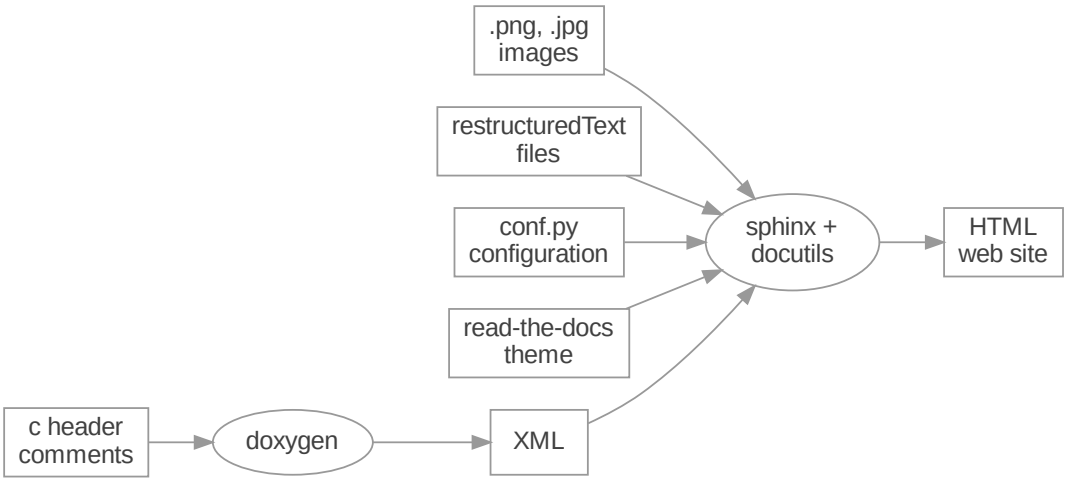


Fig. 3: Schematic of the documentation build process

Linux

Common to all Linux installations, install the Python dependencies required to build the documentation:

```
pip install -U -r ~/zephyrproject/zephyr/doc/requirements.txt
```

On Ubuntu Linux:

```
sudo apt-get install --no-install-recommends doxygen graphviz librsvg2-bin \
texlive-latex-base texlive-latex-extra latexmk texlive-fonts-recommended␣
↪imagemagick
```

On Fedora Linux:

```
sudo dnf install doxygen graphviz texlive-latex latexmk \
texlive-collection-fontsrecommended librsvg2-tools ImageMagick
```

On Clear Linux:

```
sudo swupd bundle-add texlive graphviz ImageMagick
```

On Arch Linux:

```
sudo pacman -S graphviz doxygen librsvg texlive-core texlive-bin \
texlive-latexextra texlive-fontsextra imagemagick
```

macOS

Install the Python dependencies required to build the documentation:

```
pip install -U -r ~/zephyrproject/zephyr/doc/requirements.txt
```

Use `brew` and `tlmgr` to install the tools:

```
brew install doxygen graphviz mactex librsvg imagemagick
tlmgr install latexmk
tlmgr install collection-fontsrecommended
```

Windows

Install the Python dependencies required to build the documentation:

```
pip install -U -r %HOMEPATH$\zephyrproject\zephyr\doc\requirements.txt
```

Open a `cmd.exe` window as **Administrator** and run the following command:

```
choco install doxygen.install graphviz strawberryperl miktex rsvg-convert␣
↪imagemagick
```

> **Note**
>
> On Windows, the Sphinx executable `sphinx-build.exe` is placed in the `Scripts` folder of your Python installation path. Depending on how you have installed Python, you might need to add this folder to your `PATH` environment variable. Follow the instructions in Windows Python Path to add those if needed.

Documentation presentation theme

Sphinx supports easy customization of the generated documentation appearance through the use of themes. Replace the theme files and do another `make html` and the output layout and style is changed. The `read-the-docs` theme is installed as part of the *Get Zephyr and install Python dependencies* step you took in the getting started guide.

Running the documentation processors

The `/doc` directory in your cloned copy of the Zephyr project git repo has all the .rst source files, extra tools, and Makefile for generating a local copy of the Zephyr project's technical documentation. Assuming the local Zephyr project copy is in a folder `zephyr` in your home folder, here are the commands to generate the html content locally:

```
# On Linux/macOS
cd ~/zephyrproject/zephyr/doc
# On Windows
cd %userprofile%\zephyrproject\zephyr\doc

# Use cmake to configure a Ninja-based build system:
cmake -GNinja -B_build .

# Enter the build directory
cd _build

# To generate HTML output, run ninja on the generated build system:
ninja html
# If you modify or add .rst files, run ninja again:
ninja html

# To generate PDF output, run ninja on the generated build system:
ninja pdf
```

> **Warning**
>
> The documentation build system creates copies in the build directory of every .rst file used to generate the documentation, along with dependencies referenced by those .rst files.

This means that Sphinx warnings and errors refer to the **copies**, and **not the version-controlled original files in Zephyr**. Be careful to make sure you don't accidentally edit the copy of the file in an error message, as these changes will not be saved.

Depending on your development system, it will take up to 15 minutes to collect and generate the HTML content. When done, you can view the HTML output with your browser started at `doc/_build/html/index.html` and if generated, the PDF file is available at `doc/_build/latex/zephyr.pdf`.

If you want to build the documentation from scratch just delete the contents of the build folder and run `cmake` and then `ninja` again.

Note

If you add or remove a file from the documentation, you need to re-run CMake.

On Unix platforms a convenience doc/Makefile can be used to build the documentation directly from there:

```
cd ~/zephyrproject/zephyr/doc

# To generate HTML output
make html

# To generate PDF output
make pdf
```

Developer-mode Document Building

When making and testing major changes to the documentation, we provide an option to temporarily stub-out the auto-generated Devicetree bindings documentation so the doc build process runs faster.

To enable this mode, set the following option when invoking cmake:

```
-DDT_TURBO_MODE=1
```

Another step that typically takes a long time is the generation of the list of supported features for each board. This can be disabled by setting the following option when invoking cmake:

```
-DHW_FEATURES_TURBO_MODE=1
```

Invoking `make` with the following target will build the documentation without either of the aforementioned features:

```
cd ~/zephyrproject/zephyr/doc

# To generate HTML output without detailed Devicetree bindings documentation
# and supported features index
make html-fast
```

For even faster iteration when working on narrative pages, additional `SKIP_*` options allow skipping entire categories of auto-generated content:

`SKIP_DOXYGEN`
: Skip running Doxygen and building the C API reference.

`SKIP_KCONFIG`
: Skip generating the Kconfig option reference and its search page.

`SKIP_EXTERNAL_CONTENT`
: Skip copying in the board, sample, and snippet pages that are maintained outside of the `doc/` folder (i.e. the contents of the boards, samples, and snippets folders).

Each option can be enabled individually:

```
make html SKIP_DOXYGEN=1
```

The **`make html-minimal`** target combines all of them on top of `html-fast`, making it the fastest way to preview a change to a narrative page:

```
make html-minimal
```

Warning

Content that a skipped generator would have produced is replaced with a placeholder, cross-references to it render as plain text, and the related warnings are suppressed. A build using any `SKIP_*` option is therefore only suitable for a local preview: it cannot be used to validate cross-references, and its output must not be published.

When working with documentation for boards from a specific vendor, it is also possible to limit generation of the list of supported features to subset of board vendors. This can be done by setting the following option when invoking cmake:

```
-DHW_FEATURES_VENDOR_FILTER=vendor1,vendor2
```

This option can also be used with the **`make`** wrapper:

```
cd ~/zephyrproject/zephyr/doc

# To generate HTML output with supported features limited to a subset of vendors
make html HW_FEATURES_VENDOR_FILTER=vendor1,vendor2
```

Viewing generated documentation locally

The generated HTML documentation can be hosted locally with python for viewing with a web browser:

```
$ python3 -m http.server -d _build/html
```

Note

WSL2 users may need to explicitly bind the address to `127.0.0.1` in order to be accessible from the host machine:

```
$ python3 -m http.server -d _build/html --bind 127.0.0.1
```

Alternatively, the documentation can be built with the `make html-live` (or `make html-live-fast` or `make html-live-minimal`) command, which will build the documentation, host it locally, and watch the documentation directory for changes. When changes are observed, it will automatically rebuild the documentation and refresh the hosted files.

Building the Doxygen documentation standalone

The `doxygen` build target can be used to only build the Doxygen (API) documentation, which is much faster than building the full documentation set:

```
cd ~/zephyrproject/zephyr/doc
make doxygen
```

The output can be found in `_build/doxygen/html`.

The `doxygen-xml` target builds the same documentation with only the XML output enabled, in `_build/doxygen-xml/xml`.

In the regular documentation build, references made from Doxygen comments to the main documentation (e.g. `@kconfig{}`, `@dtcompatible{}` or `@rstref{}`, see *Referencing the main documentation*) are automatically resolved into hyperlinks. In a standalone Doxygen build, they are left as plain text since the rest of the documentation is not available.

Linking external Doxygen projects against Zephyr

External projects that build upon Zephyr functionality and wish to refer to Zephyr documentation in Doxygen (through the use of @ref), can utilize the tag file exported at zephyr.tag

Once downloaded, the tag file can be used in a custom `doxyfile.in` as follows:

```
TAGFILES = "/
↪path/to/zephyr.tag=https://docs.zephyrproject.org/latest/doxygen/html/"
```

For additional information refer to Doxygen External Documentation.

Documentation Guidelines
This page provides some simple guidelines for writing documentation using the reSTructuredText (reST) markup language and Sphinx documentation generator.

Documentation Generation
As you write documentation, it can be helpful to see how it will look when rendered.

This page describes how to build the Zephyr documentation locally.

7.3 Dealing with external components

7.3.1 Contributing External Components

In some cases it is desirable to leverage existing, external source code in order to avoid re-implementing basic functionality or features that are readily available in other open source projects.

This section describes the circumstances under which external source code can be imported into Zephyr, and the process that governs the inclusion.

There are three main factors that will be considered during the inclusion process in order to determine whether it will be accepted. These will be described in the following sections.

Note that most of this page deals with external components that end up being compiled and linked into the final image, and programmed into the target hardware. For external tooling that is only used during compilation, code analysis, testing or simulation please refer to the *Contributing External Tooling* section at the end of the page.

Software License

> **Note**
>
> External source code licensed under the Apache-2.0 license is not subject to this section.

Integrating code into the Zephyr Project from other projects that use a license other than the Apache 2.0 license needs to be fully understood in context and approved by the Zephyr governing board, as described in the Zephyr project charter. The board will automatically reject licenses that have not been approved by the Open Source Initiative (OSI). See the *Submission and review process* section for more details.

By carefully reviewing potential contributions and also enforcing a *Developer Certification of Origin (DCO)* for contributed code, we ensure that the Zephyr community can develop products with the Zephyr Project without concerns over patent or copyright issues.

Merit

Just like with any other regular contribution, one that contains external code needs to be evaluated for merit. However, in the particular case of code that comes from an existing project, there are additional questions that must be answered in order to accept the contribution. More specifically, the following will be considered by the Technical Steering Committee and evaluated carefully before the external source code is accepted into the project:

- Is this the most optimal way to introduce the functionality to the project? Both the cost of implementing this internally and the one incurred in maintaining an externally developed codebase need to be evaluated.
- Is the external project being actively maintained? This is particularly important for source code that deals with security or cryptography.
- Have alternatives to the particular implementation proposed been considered? Are there other open source project that implement the same functionality?

Mode of integration

There are two ways of integrating external source code into the Zephyr Project, and careful consideration must be taken to choose the appropriate one for each particular case.

Integration in the main tree The first way to integrate external source code into the project is to simply import the source code files into the main `zephyr` repository. This automatically implies that the imported source code becomes part of the "mainline" codebase, which in turn requires that:

- The code is formatted according to the Zephyr *Coding Style Guidelines*
- The code adheres to the project's *Coding Guidelines*
- The code is subject to the same checks and verification requirements as the rest of the code in the main tree, including static analysis
- All files contain an SPDX tag if not already present
- If the source is not Apache 2.0 licensed, an `[[annotations]]` entry describing the component is added to the REUSE.toml file at the root of the repository. This is what the *licensing page* is generated from.

This mode of integration can be applicable to both small and large external codebases, but it is typically used more commonly with the former.

Integration as a module The second way of integrating external source code into the project is to import the whole or parts of the third-party open source project into a separate repository, and then include it under the form of a *module*. With this approach the code is considered as being developed externally, and thus it is not automatically subject to the requirements of the previous section.

Integration in main manifest file (west.yaml) Integrating external code into the main `west.yml` manifest file is limited to code that is used by a Zephyr subsystem (libraries), by a platform, drivers (HAL) or tooling needed to test or build Zephyr components.

The integration of modules in this group is validated by the Zephyr project CI, and verified to be working with each Zephyr release.

Integrated modules will not be removed from the tree without a detailed migration plan.

Integration as optional modules Standalone or loose integration of modules/projects without any incoming dependencies shall be made optional and shall be kept standalone. Optional projects that provide value to users directly and through a Zephyr subsystem or platform shall be added to an optional manifest file that is filtered by default. (`submanifests/optional.yml`).

Such optional projects might include samples and tests in their own repositories.

There shall not be any direct dependency added in the Zephyr code tree (Git repository) and all sample or test code shall be maintained as part of the module.

> **Note**
>
> This is valid for all new optional modules. Existing optional modules with samples and test code in the Zephyr Git repository will be transitioned out over time.

Integration as external modules Similar to optional modules, but added to the Zephyr project as an entry in the documentation using a pre-defined template. This type of modules exists outside the Zephyr project manifest with documentation instructing users and developers how to integrate the functionality.

Ongoing maintenance

Regardless of the mode of integration, external source code that is integrated in Zephyr requires regular ongoing maintenance. The submitter of the proposal to integrate external source code must therefore commit to maintain the integration of such code for the foreseeable future. This may require adding an entry in the `MAINTAINERS.yml` as part of the process.

Submission and review process

Before external source code can be included in the project, it must be reviewed and accepted by the Technical Steering Committee (TSC) and, in some cases, by the Zephyr governing board.

A request for external source code integration must be made by creating a new issue in the Zephyr project issue tracking system on GitHub with details about the source code and how it integrates into the project.

Follow the steps below to begin the submission process:

1. Make sure to read through the *Contributing External Components* section in detail, so that you are informed of the criteria used by the TSC and board in order to approve or reject a request
2. Use the New External Source Code Issue to open an issue

3. Fill out all required sections, making sure you provide enough detail for the TSC to assess the merit of the request. Optionally you can also create a Pull Request that demonstrates the integration of the external source code and link to it from the issue
4. Wait for feedback from the TSC, respond to any additional questions added as GitHub issue comments

If, after consideration by the TSC, the conclusion is that integrating external source code is the best solution, and the external source code is licensed under the Apache-2.0 license, the submission process is complete and the external source code can be integrated.

If, however, the external source code uses a license other than Apache-2.0, then these additional steps must be followed:

1. The TSC chair will forward the link to the GitHub issue created during the early submission process to the Zephyr governing board for further review
2. The Zephyr governing board has two weeks to review and ask questions:
   - If there are no objections, the matter is closed. Approval can be accelerated by unanimous approval of the board before the two weeks are up
   - If a governing board member raises an objection that cannot be resolved via email, the board will meet to discuss whether to override the TSC approval or identify other approaches that can resolve the objections
3. On approval of the Zephyr TSC and governing board the submission process is complete

The flowchart below shows an overview of the process:

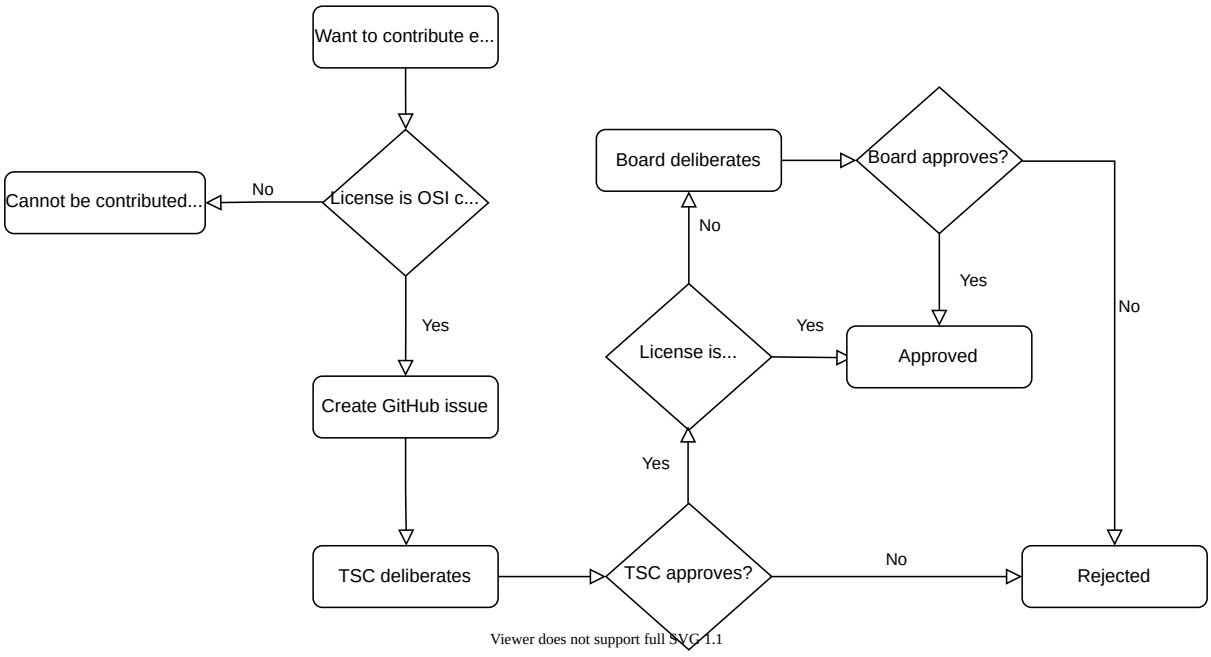


Fig. 4: Submission process

Contributing External Tooling

This section deals exclusively with the inclusion of external tooling in the Zephyr project, where tooling is defined as software that assists the compilation, testing or simulation processes but in no case ends up being part of the code compiled and linked into the final image. "Inclusion" in this context means becoming part of the Zephyr default distribution either in the main tree directly under the `scripts/` folder or indirectly as a west project in the main `west.yml` manifest.

Therefore, this section does not apply to 3rd-party tooling such as toolchains, simulators or others, which may still be referenced by the Zephyr build system or docs without being included in Zephyr.

Tooling components must be released under a license approved by the Open Source Initiative (OSI).

Just like with regular external components, tooling that is imported from another project can be integrated either in the main tree or as a *west project*. Note that in this case the corresponding west project will not be a *module*, because tooling does not make use of the Zephyr build system and does not need to be processed by it. Please see *Modules vs west projects* for additional information on the differences.

If the tool is integrated in the main tree it should be placed under the `scripts/` folder. If the tool is integrated as a west project, then the project repository can be hosted outside the zephyrproject-rtos GitHub organization, provided that the project is made optional via the `group-filter:` field in the main `west.yml` manifest. More info on optional projects can be found in *this section*.

The TSC must approve every Pull Request that introduces a new external tooling component. This will be done on a case-by-case, individual analysis of the proposed addition by the TSC representatives.

Additional considerations about the main manifest

In general, any additions or removals whatsoever to the `projects:` section of the main manifest file requires TSC approval. This includes, but is not limited to:

- Adding and removing groups and group filters
- Adding and removing projects
- Adding and removing `import` statements

7.3.2 Binary Blobs

In the context of an operating system that supports multiple architectures and many different IC families, some functionality may be unavailable without the help of executable code distributed in binary form. Binary blobs (or blobs for short) are files containing proprietary machine code or data in a binary format, e.g. without corresponding source code released under an OSI approved license.

Zephyr supports downloading and using third-party binary blobs via its built-in mechanisms, with some important caveats, described in the following sections. It is important to note that all the information in this section applies only to upstream (vanilla) Zephyr.

There are no limitations whatsoever (except perhaps license compatibility) in the support for binary blobs in forks or third-party distributions of Zephyr. In fact, Zephyr's build system supports arbitrary use cases related to blobs. This includes linking against libraries, flashing images to targets, etc. Users are therefore free to create Zephyr-based downstream software which uses binary blobs if they cannot meet the requirements described in this page.

Software license

Most binary blobs are distributed under proprietary licenses which vary significantly in nature and conditions. It is up to the vendor to specify the license as part of the blob submission process. Blob vendors may impose a click-through or other EULA-like workflow when users fetch and install blobs.

Hosting

Blobs must be hosted on the Internet and managed by third-party infrastructure. Two potential examples are Git repositories and web servers managed by individual hardware vendors.

The Zephyr Project does not host binary blobs in its Git repositories or anywhere else.

Fetching blobs

Blobs are fetched from official third-party sources by the *west blobs* command.

The blobs themselves must be specified in the *module.yml* files included in separate Zephyr *module repositories* maintained by their respective vendors. This means that in order to include a reference to a binary blob to the upstream Zephyr distribution, a module repository must exist first or be created as part of the submission process.

Each blob which may be fetched must be individually identified in the corresponding `module.yml` file. A specification for a blob must contain:

- An abstract description of the blob itself
- Version information
- A reference to vendor-provided documentation
- The blob's *type*, which must be one of the allowed types
- A checksum for the blob, which `west blobs` checks after downloading. This is required for reproducibility and to allow bisecting issues as blobs change using Git and west
- License text applicable to the blob or a reference to such text, in SPDX format

See the *corresponding section* for a more formal definition of the fields.

The *west blobs* command can be used to list metadata of available blobs and to fetch blobs from user-selected modules.

The `west blobs` command only fetches and stores the binary blobs themselves. Any accompanying code, including interface header files for the blobs, must be present in the corresponding module repository.

Tainting

Inclusion of binary blobs will taint the Zephyr build. The definition of tainting originates in the Linux kernel and, in the context of Zephyr, a tainted image will be one that includes binary blobs in it.

Tainting will be communicated to the user in the following manners:

- One or more Kconfig options `TAINT_BLOBS_*` will be set to `y`
- The Zephyr build system, during its configuration phase, will issue a warning. It will be possible to disable the warning using Kconfig
- The `west spdx` command will include the tainted status in its output
- The kernel's default fatal error handler will also explicitly print out the kernel's tainted status
- The boot banner prints the kernel's tainted status

Allowed types

The following binary blob types are acceptable in Zephyr:

- Precompiled libraries: Hardware enablement libraries, distributed in precompiled binary form, typically for SoC peripherals. An example could be an enablement library for a wireless peripheral
- Firmware images: An image containing the executable code for a secondary processor or CPU. This can be full or partial (typically delta or patch data) and is generally copied into

RAM or flash memory by the main CPU. An example could be the firmware for the core running a Bluetooth LE Controller

- Miscellaneous binary data files. An example could be pre-trained neural network model data

Hardware agnostic features provided via a proprietary library are not acceptable. For example, a proprietary and hardware agnostic TCP/IP stack distributed as a static archive would be rejected.

Note that just because a blob has an acceptable type does not imply that it will be unconditionally accepted by the project; any blob may be rejected for other reasons on a case by case basis (see library-specific requirements below). In case of disagreement, the TSC is the arbiter of whether a particular blob fits in one of the above types.

Precompiled library-specific requirements

This section contains additional requirements specific to precompiled library blobs.

Any person who wishes to submit a precompiled library must represent that it meets these requirements. The project may remove a blob from the upstream distribution if it is discovered that the blob fails to meet these requirements later on.

Interface header files The precompiled library must be accompanied by one or more header files, distributed under a non-copyleft OSI approved license, that define the interface to the library.

Allowed dependencies This section defines requirements related to external symbols that a library blob requires the build system to provide.

- The blob must not depend on Zephyr APIs directly. In other words, it must have been possible to build the binary without any Zephyr source code present at all. This is required for loose coupling and maintainability, since Zephyr APIs may change and such blobs cannot be modified by all project maintainers
- Instead, if the code in the precompiled library requires functionality provided by Zephyr (or an RTOS in general), an implementation of an OS abstraction layer (aka porting layer) can be provided alongside the library. The implementation of this OS abstraction layer must be in source code form, released under an OSI approved license and documented using Doxygen.

Toolchain requirements Precompiled library blobs must be in a data format which is compatible with and can be linked by a toolchain supported by the Zephyr Project. This is required for maintainability and usability. Use of such libraries may require special compiler and/or linker flags, however. For example, a porting layer may require special flags, or a static archive may require use of specific linker flags.

Limited scope Allowing arbitrary library blobs carries a risk of degrading the degree to which the upstream Zephyr software distribution is open source. As an extreme example, a target with a zephyr kernel clock driver that is just a porting layer around a library blob would not be bootable with open source software.

To mitigate this risk, the scope of upstream library blobs is limited. The project maintainers define an open source test suite that an upstream target must be able to pass using only open source software included in the mainline distribution and its modules. The open source test suite currently consists of:

- `samples/philosophers`
- `tests/kernel`

The scope of this test suite may grow over time. The goal is to specify tests for a minimal feature set which must be supported via open source software for any target with upstream Zephyr support.

At the discretion of the release team, the project may remove support for a hardware target if it cannot pass this test suite.

Support and maintenance

The Zephyr Project is not expected to be responsible for the maintenance and support of contributed binary blobs. As a consequence, at the discretion of the Zephyr Project release team, and on a case-by-case basis:

- GitHub issues reported on the zephyr repository tracker that require use of blobs to reproduce may not be treated as bugs
- Such issues may be closed as out of scope of the Zephyr project

This does not imply that issues which require blobs to reproduce will be closed without investigation. For example, the issue may be exposing a bug in a Zephyr code path that is difficult or impossible to trigger without a blob. Project maintainers may accept and attempt to resolve such issues.

However, some flexibility is required because project maintainers may not be able to determine if a given issue is due to a bug in Zephyr or the blob itself, may be unable to reproduce the bug due to lack of hardware, etc.

Blobs must have designated maintainers that must be responsive to issue reports from users and provide updates to the blobs to address issues. At the discretion of the Zephyr Project release team, module revisions referencing blobs may be removed from `zephyr/west.yml` at any time due to lack of responsiveness or support from their maintainers. This is required to maintain project control over bit-rot, security issues, etc.

The submitter of the proposal to integrate a binary blob must commit to maintain the integration of such blob for the foreseeable future.

Regarding Continuous Integration, binary blobs will **not** be fetched in the project's CI infrastructure that builds and optionally executes tests and samples to prevent regressions and issues from entering the codebase. This includes both CI ran when a new GitHub Pull Request is opened as well as any other regularly scheduled execution of the CI infrastructure.

Submission and review process

For references to binary blobs to be included in the project, they shall be submitted through the standard pull request (PR) workflow and do not require Technical Steering Committee (TSC) approval. Maintainers and reviewers are responsible for ensuring that all blob-related submissions conform to the documented criteria as defined in *Binary Blobs*.

The PR for initial integration with binary blobs shall include details about the blobs and the functionality they provide. The PR shall have the following information to support the review:

- Binary blob origin.
- Type of blob (precompiled library, firmware image).
- Zephyr module that the blob(s) will be referenced from.
- Brief description of what the blob(s) do.
- What other components do the blob(s) depend on, if any?
- License the blob(s) are distributed under.

If the initial integration PR receives the appropriate approvals the binary blob(s) can be integrated. The standard *PR escalation procedures* shall apply for any technical disagreements.

Subsequent updates to binary blobs follow the *module update procedure*.

Contributing External Components
Basic functionality or features that would make useful addition to Zephyr might be readily available in other open source projects, and it is recommended and encouraged to reuse such code. This page describes in more details when and how to import external source code into Zephyr.

Contributing External Tooling
Similarly, external tooling used during compilation, code analysis, testing or simulation, can be beneficial and is covered in this section.

Binary Blobs
As some functionality might only be made available with the help of executable code distributed in binary form, this page describes the process and guidelines for *contributing binary blobs* to the project.

7.4 Need help along the way?

If you have questions related to the contribution process, the Zephyr community is here to help. You may join our Discord channel or use the Developer Mailing List.

Chapter 8

Project and Governance

8.1 Technical Steering Committee (TSC)

8.1.1 TSC Member Role

The TSC role and its responsibilities is defined in the Zephyr project charter.

Membership

A TSC member plays a pivotal role in shaping the technical direction of the Zephyr Project. TSC members work collaboratively with other TSC members, contributors, and stakeholders to ensure the project's success and sustainability.

By fulfilling the rights and responsibilities below, TSC members contribute to the overall success and growth of the Zephyr Project, ensuring that it remains a vibrant and thriving open-source community for years to come.

Rights

Decision Making
Participate in key decisions related to the project's technical direction, including architectural changes, feature additions, and release planning.

Voting
Exercise voting rights on important matters discussed within the TSC, including feature proposals, code contributions, and community initiatives.

Access
Gain access to relevant project repositories, documentation, and communication channels to stay informed and contribute effectively.

Leadership
Take on leadership roles within working groups or subcommittees dedicated to specific technical areas or initiatives.

Representation
Act as a representative of the broader Zephyr community, advocating for the interests of contributors, users, and stakeholders.

Responsibilities TSC members are expected to fulfill the following responsibilities, though it is not mandatory to fulfill all:

Technical Oversight
Provide guidance and oversight on technical matters, ensuring alignment with project

goals, standards, and best practices through active participation as core members in working groups and committees.

Code Review
Participate in code reviews to maintain code quality, consistency, and compatibility with project standards.

Community Engagement
Engage with the community through forums, mailing lists, conferences, and other channels to foster collaboration, address concerns, and gather feedback.

Documentation
Contribute to the development and maintenance of project documentation, including technical guides, API references, and best practices.

Release Management
Collaborate with the release manager and other stakeholders to plan and coordinate project releases, ensuring timely delivery and quality assurance.

Contributor Support
Support and mentor new contributors, helping them navigate the project's codebase, processes, and community norms.

Issue Triage
Assist in triaging and prioritizing issues reported by users and contributors, facilitating timely resolution and communication.

Compliance and Licensing
Ensure compliance with project licensing requirements and open-source best practices, addressing any licensing-related issues that may arise.

Conflict Resolution
Facilitate constructive discussions and resolution of technical disagreements or conflicts within the community, promoting a healthy and inclusive environment.

Continuous Improvement
Continuously seek opportunities to improve project governance, processes, and infrastructure, driving innovation and sustainability.

Appointed TSC Members

See Zephyr project charter for more details.

- Appointed TSC members have no term limits besides the term of their employment at the organization they represent or their organization's membership in the Zephyr Project.
- Appointed TSC members can select an Alternate from the same organization.

Elected TSC Members

Per the Zephyr project charter, TSC members can nominate representatives from the technical community at the rate of no more than one per quarter.

- Majority vote is required to confirm a candidate.
- Once elected, a TSC member serves for 2 years.
- Elected TSC members do not have the right to appoint an Alternate.
- To ensure continuity of the TSC, at the end of the 2 year term, the TSC is required to reconfirm the membership of elected members. If the elected member declines a new term or if the TSC fails to reconfirm the term, the seat will be open for new nominations.
- If an elected TSC member resigns before the end of the 2 year term, their spot will be open for new members outside of the quarterly nomination limit. The elected member will serve a 2 year term.

- The TSC has the right to terminate elected members who become inactive and are not fulfilling the responsibilities of TSC members as described in this document.
- The number of elected members shall not exceed 20% of the total of appointed members.
- Existing TSC members who were elected before May 2024 shall be re-confirmed after completing the 2 year term since they were first elected.

Suspensions

As noted under Section 8b of the Project Charter, voting rights for a representative who misses three consecutive meetings are subject to suspension and suspended representatives do not count towards the quorum requirement.

A representative's suspension will end and voting rights be restored at the start of the next attended meeting. The TSC enforces the suspension policy for voting members who miss three consecutive TSC weekly meetings.

Multi-day meetings (F2F events) are counted as "one" meeting. The TSC voted on February 16, 2022 to discontinue default enforcement of the suspension policy. The TSC voted on January 18, 2023 to re-enact default enforcement of the suspension policy.

Notice of suspension will be sent to representatives who miss three consecutive meetings, noting that rights will be restored upon next attendance of a TSC meeting.

Note

As per Section 4b of the Project Charter, Platinum and Silver Members may choose to opt out of a voting seat on the TSC.

Members who opt out and then wish to reclaim their seat later will have their voting rights restored at the start of the second consecutive meeting attended following notification to the TSC Chair.

Voting

Voting in the Zephyr Project is defined under Section 8 of the Project Charter.

Additional points of clarity / TSC interpretation have been added below. The Governing Board may opt to update the Charter to include the below refinements. Until then, additional clarifications (if/where needed) will be discussed in the Process Working Group, and approved in the TSC.

- TSC In-Meeting Voting For items requesting an in-meeting vote of the Zephyr Technical Steering Committee (TSC), assuming quorum requirements have been met, the default voting mechanism will be a verbal motion to determine if there is general consensus.
- If there are no objections to a motion being brought forward, general consensus is assumed and the motion passes.
- Should there be any objections raised, the vote will move to email, and be executed using the Voting Guidelines outlined in Section 8 of the Project Charter.
- Should a motion be deemed urgent by the TSC Chair, and assuming quorum requirements have been met, the Chair may call for a roll call vote in-meeting.

Voting Options

- Voting Options are:
  - "Yes",
  - "No" or

– "Abstain"

Abstention Abstentions do not count in tallying the vote negatively or positively; when members abstain, they are in effect attending only to contribute to a quorum.

Abstentions do not impact the number of votes needed to decide a vote.

Quorum Quorum for TSC meetings shall require 60% of the voting representatives… (ref 8b of the Charter)

Decisions Decisions by vote shall be based on a majority vote, provided that at least sixty percent (60%) of the **TSC** representatives must be either present or participating electronically or by written action in order to conduct a valid vote. (ref 8c of the Charter)

Example A:

> 40 eligible TSC voters. 3 abstain from a vote on a motion. 12 vote Yes. 11 vote No.
>
> Quorum reached: 26 votes cast (quorum = 60% of 40 = 24) Majority vote: 12 Yes vs. 11 No. Yes wins. Motion adopted.

Example B:

> 40 eligible TSC voters. 5 abstain from a vote on a motion. 12 vote Yes. 6 vote No. Quorum reached? 23 votes cast (quorum = 60% of 40 = 24)
>
> Vote is not valid. Quorum not reached.

Example C:

> 40 eligible TSC voters. 21 abstain from a vote on a motion. 2 vote Yes. 1 votes No. Quorum reached? 24 votes cast (quorum = 60% of 40 = 24)
>
> Majority vote: 2 Yes vs. 1 No. Yes wins.

Immutable Votes

Votes are considered immutable once cast. A voter may not change their vote, once cast, between the time a Motion is brought forth and the time at which results are announced.

8.2 TSC Project Roles

8.2.1 Project Roles

You can participate in the Zephyr Project as a *Contributor*, *Collaborator*, or *Maintainer*.

Contributor: Any community member who contributes code, documentation, or other assets to the project.

Collaborator: An active Contributor who is involved in one or more areas of the project.

Maintainer: A lead Collaborator responsible for a specific area or subsystem within the project. Maintainers also serve as representatives for their area on the Technical Steering Committee (TSC) as needed.

Areas in this context refer to specific subsystems, components, or modules within the Zephyr Project codebase. Examples of areas include, but are not limited to, networking, file systems, device drivers, architecture support, and board support packages (boards and SoC definitions).

Areas in the project should at least have one Maintainer and may have multiple Collaborators. Depending on the size and complexity of the area, there may be multiple Maintainers as well, however, the number of maintainers should be kept to a practical minimum to ensure effective management and decision-making.

Contributor

A *Contributor* is a developer who wishes to contribute to the project, at any level.

Contributors are granted the following rights and responsibilities:

- Right to contribute code, documentation, translations, artwork, etc.
- Right to report defects (bugs) and suggestions for enhancement.
- Right to participate in the process of reviewing contributions by others.
- Right to participate in the feature development process.
- Responsibility to abide by decisions, once made. They are welcome to provide new, relevant information to reopen decisions.
- Responsibility for issues and bugs introduced by one's own contributions.
- Responsibility to respect the rules of the community.
- Responsibility to provide constructive advice whenever participating in discussions and in the review of contributions.
- Responsibility to follow the project's code of conduct (https://github.com/zephyrproject-rtos/zephyr/blob/main/CODE_OF_CONDUCT.md)

Zephyr Contributor Badge When your first contribution to the Zephyr project gets merged, you'll become eligible to claim your Zephyr Contributor Badge. This digital badge can be displayed on your website, blog, social media profile, etc. It will allow you to showcase your involvement in the Zephyr project and help raise its awareness.

You may apply for your Contributor Badge by filling out the Zephyr Contributor Badge form.

Collaborator

A *Collaborator* is a Contributor who is also participating in the maintenance of Zephyr source code. Their input weighs more when decisions are made, in a fully meritocratic fashion.

You become a collaborator by showing consistent involvement in the project, including making significant contributions to the project over a period of time and by participating in the development and review of at least one area.

To request to become a Collaborator, submit a pull request adding yourself to the `collaborators` section of an area in the *MAINTAINERS File* and notify the maintainers of that area. The addition needs to be approved by the maintainers of that area.

Collaborators have the following rights and responsibilities, in addition to those listed for Contributors:

- Right to participate in setting goals for the short and medium terms for the area being maintained, alongside the maintainers of that area.
- Responsibility to participate in the feature development process.
- Responsibility to review relevant code changes within reasonable time.
- Responsibility to ensure the quality of the code to expected levels.
- Responsibility to participate in community discussions.
- Responsibility to mentor new contributors when appropriate
- Responsibility to participate in the quality verification and release process, when those happen.

Collaborator change requests on pull requests should be addressed by the original submitter. In cases where the changes requested do not follow the *expectations* and the guidelines of the

project or in cases of disagreement, it is the responsibility of the assignee to advance the review process and resolve any disagreements.

Collaborator approval of pull requests are counted toward the minimum required approvals needed to merge a PR. Other criteria for merging may apply.

Maintainer

A *Maintainer* is a Collaborator who is also responsible for knowing, directing and anticipating the needs of a given zephyr source code area.

An individual may become a maintainer of a given area by demonstrating sustained and meaningful contributions over time. This includes:

- Making significant technical contributions to the area.
- Actively participating in the development process, including code reviews and design discussions.
- Establishing familiarity with the architecture, constraints, and evolution of the area.

Maintainer status may also be granted when introducing a new area or subsystem to the project as its original author while demonstrating recognized expertise and long-term commitment to the area.

In all cases, maintainership reflects both technical ownership and ongoing engagement with the community supporting that area.

To request to become a Maintainer, submit a pull request adding yourself to the `maintainers` section of an area in the *MAINTAINERS File* and notify the existing maintainers of that area. The addition needs to be approved by the maintainers of that area.

Usually, there's only one maintainer but depending on the area, there may be multiple maintainers sharing the responsibility.

Maintainers have the following rights and responsibilities, in addition to those listed for Contributors and Collaborators:

- Right to set the overall architecture of the relevant subsystems or areas of involvement.
- Right to make decisions in the relevant subsystems or areas of involvement, in conjunction with the collaborators and submitters. See *PR Technical Escalation*.
- Responsibility to convey the direction of the relevant subsystem or areas to the TSC
- Responsibility to ensure all contributions of the project have been reviewed within reasonable time.
- Responsibility to enforce the Code of Conduct. As leaders in the community, maintainers are expected to be role models in upholding the project's code of conduct and creating a welcoming and inclusive environment for everyone.
- Responsibility to triage static analysis issues in their code area. See *Static Code Analysis*.

Maintainer approval of pull requests are counted toward the minimum required approvals needed to merge a PR. Other criteria for merging may apply.

8.2.2 Teams and Supporting Activities

Assignee

An *Assignee* is one of the maintainers of an area being changed in a pull request.

Assignees are set automatically based on the code being changed. Other Maintainers or the Release Engineering team can set assignees when the latter is not possible, or when another assignee is more appropriate, in this last case documenting in a comment the motivation for the change.

- Responsibility to drive the pull request to a mergeable state
- Right to dismiss stale and unrelated reviews or reviews not following *expectations* from reviewers and seek reviews from additional maintainers, developers and contributors
- Right to block pull requests from being merged until issues or changes requested are addressed
- Responsibility to re-assign a pull request if they are the original submitter of the code
- Solicit approvals from maintainers of the subsystems affected
- Responsibility to drive the *PR Technical Escalation* process

Static Analysis Audit Team

The Static Analysis Audit team works closely with the release engineering team to ensure that static analysis defects opened during a release cycle are properly addressed. The team has the following rights and responsibilities:

- Right to revert any triage in a static analysis tool (e.g: Coverity) that does not follow the project expectations.
- Responsibility to inform code owners about improper classifications.
- Responsibility to alert TSC if any issues are not adequately addressed by the responsible code owners.

Joining the Static Analysis Audit team

- Contributors highly involved in the project with some expertise in static analysis.

Release Engineering Team

A team of active Maintainers involved in multiple areas.

- The members of the Release Engineering team are expected to fill the Release Manager role based on a defined cadence and selection process.
- The cadence and selection process are defined by the Release Engineering team and are approved by the TSC.
- The team reports directly into the TSC.

Release Engineering team has the following rights and responsibilities:

- Right to merge code changes to the zephyr tree following the project rules.
- Right to revert any changes that have broken the code base
- Right to close any stale changes after <N> months of no activity
- Responsibility to take directions from the TSC and follow them.
- Responsibility to coordinate code merges with maintainers.
- Responsibility to merge all contributions regardless of their origin and area if they have been approved by the respective maintainers and follow the merge criteria of a change.
- Responsibility to keep the Zephyr code base in a working and passing state (as per CI)

Joining the Release Engineering team

- Maintainers highly involved in the project may be nominated by a TSC voting member to join the Release Engineering team. Nominees may become members of the team by approval of the existing TSC voting members.

- To ensure a functional Release Engineering team the TSC shall periodically review the team's followed processes, the appropriate size, and the membership composition (ensure, for example, that team members are geographically distributed across multiple locations and time-zones).

Release Manager

A *Maintainer* responsible for driving a specific release to completion following the milestones and the roadmap of the project for this specific release.

- TSC has to approve a release manager.

A Release Manager is a member of the Release Engineering team and has the rights and responsibilities of that team in addition to the following:

- Right to manage and coordinate all code merges after the code freeze milestone (M3, see program management overview.)
- Responsibility to drive and coordinate the triaging process for the release
- Responsibility to create the release notes of the release
- Responsibility to notify all stakeholders of the project, including the community at large about the status of the release in a timely manner.
- Responsibility to coordinate with QA and validation and verify changes either directly or through QA before major changes and major milestones.

Roles / Permissions

Table 1: Project Roles vs GitHub Permissions

		Admin	**Merge Rights**	Member	Owner	Collaborator
Main Roles	Contributor					x
	Collaborator			x		
	Maintainer			x		
Supportive Roles	QA/Validation			x		x
	DevOps	**x**				
	System Admin	**x**			x	
	Release Engineering		**x**	x		

8.2.3 Role Retirement

Individuals approved by the TSC or representatives of the TSC to fill a project role who are no longer actively fulfilling the rights and responsibilities associated with their role may be requested by the TSC to retire from that role.

Retirements of inactive maintainers or collaborators are reflected by removing the individual's GitHub user name from the relevant sections of the *MAINTAINERS File* in the Zephyr repository and may be initiated by the TSC, representatives of the TSC or by the individuals themselves. Maintainers may also initiate the removal of inactive collaborators in their area.

A maintainer may object to being retired, and request a decision by the TSC.

8.2.4 MAINTAINERS File

The following guidelines apply to the structure, scope, and maintenance of the MAINTAINERS file.

- The MAINTAINERS file shall have designated individuals responsible for the accuracy, structure, and upkeep of the file, in accordance with the Zephyr Project Charter. These individuals shall be appointed by the TSC.
- The granularity of maintainership should remain practical and manageable.
- The TSC, in collaboration with existing maintainers and contributors, should actively identify and encourage contributors to step up as maintainers for orphaned areas of the codebase and should facilitate the assignment of maintainers to those components.
- Unmaintained areas shall be clearly marked as such in the MAINTAINERS file.
- Updates to the MAINTAINERS file should:
  - Generally be included as standalone commits when introducing new files or directories.
  - Major changes, including the addition of new areas and new maintainers, should be submitted as separate pull requests, requiring approval by the MAINTAINERS file's maintainers. Such activities might be the result of splitting existing large areas into smaller ones or merging smaller areas.

Guidelines for assigning maintainers to different areas of the codebase:

Architectures, core components, subsystems, samples, and tests:
Each area shall have an explicitly assigned maintainer.

Boards (including related samples and tests) and SoCs (including DTS definitions)
Each board and SoC should have an explicitly assigned maintainer through a platform area covering the boards, SoCs, and their related components and drivers.

Drivers / Backends
The area of the API level shall have a maintainer and specific driver implementations or backends shall also be covered through a platform area covering the driver implementation. The driver area or the subsystem maintainers are assigned in case of changes to driver instances or backends.

8.3 TSC Working Groups

8.3.1 Overview

The TSC, at its discretion, may establish working groups or subcommittees to serve as focused teams dedicated to specific technical areas, initiatives, or tasks.

8.3.2 Membership

Working Group Membership Eligibility

- Each Working group (WG) shall determine its own membership eligibility, in consultation with the TSC.
- Each working group shall have a team of members who are actively involved in its activities and decision-making processes.
- It is expected that WG membership shall be **open to all** Zephyr project *Collaborators*; however, working groups may impose restrictions such as the number of participants from a single company.
- All TSC members are eligible to join a working group as members, part of the responsibilities being a TSC member.
- The minimal number of members may vary depending on the complexity of the tasks and the breadth of expertise required to address them effectively.

- A working group should aim to have at least five to seven members to ensure diversity of perspectives, collaboration, and continuity.
- The structure of each working group within the Zephyr Project should be designed to ensure effectiveness, productivity, and inclusivity. While the optimal size of a working group can vary depending on the specific context and scope of its activities.
- Participation in WG meetings and discussions is open to all project *contributors*.

Working Group Chair / Co-chair

Each working group may elect a Chair and optionally a Co-Chair who is responsible for leading meetings and representing the working group to the TSC.

Working Group Chair / Co-Chair Elections

- The Chair and Co-Chair shall be elected by the members of the working group
- Any member of the working group has the right to nominate themselves for the chair/co-chair positions.
- The term for the chair/co-chair is one year
- If a chair/co-chair resigns from the position before the end of the term, a vote is to be held to elect a new chair/co-chair.

Working Group Voter Eligibility

- Voting for a Chair or Co-Chair is open to the members of the working group.
- Only 1 working group member from each company may vote in the election.
- The Chair and Co-Chair shall be members of the working group.

Working Group Election Confirmation

- The elected Chair (and/or Co-Chair) is submitted to the TSC for confirmation.
- The TSC decides to accept the outcome or requests a new voting.

8.3.3 Advisory role

- Working Groups are advisory in nature. They provide advice to the projects and to the TSC.
- Working groups operate on a rough consensus basis. If the working group is unable to reach consensus on what advice to offer, the working group Chair shall raise the issue with the TSC or the relevant committee (Safety and Security), where a formal vote can be taken, or advise the project that the working group cannot reach consensus.
- Working groups shall keep track of discussions and record any votes, decisions, or recommendations made and share results with the community and the TSC.
- Working group meetings and offline discussions shall be captured in a standalone document with all supporting details such as attendance, quorum, actions to be taken, and next steps.
- Decisions made within a working group are non-binding and are only considered ratified after communicating decisions and outcomes to the TSC.
- Lacking any objections from the TSC within 1 week after the communication or report of any results, decisions of a working group are considered confirmed and ratified.

8.3.4 TSC Working Group Lifecycle

Creation of a TSC working group

In order to create a TSC working group, a TSC member shall make a proposal to the TSC (via TSC email list) that shall cover at least the following:

- TSC working group name.
- TSC working group purpose
- TSC working group expected deliverables
- TSC working group starting participants with at least one TSC member acting as a sponsor.
- Optionally TSC working group definition of done

Update of a TSC working group

The TSC can modify a TSC working group via a TSC decision. To request such a modification, a request is made to the TSC email list.

Conclusion of a TSC working group

The TSC decides the termination of the TSC working group in accordance with the TSC decision procedure. The submission of a request to terminate the TSC working group should cover:

- TSC working group name
- TSC working group deliveries
- Motivation for termination of the TSC working group

8.4 Release Process

The Zephyr project releases on a time-based cycle, rather than a feature-driven one. Zephyr releases represent an aggregation of the work of many contributors, companies, and individuals from the community.

A time-based release process enables the Zephyr project to provide users with a balance of the latest technologies and features and excellent overall quality. A roughly 6-month release cycle allows the project to coordinate development of the features that have actually been implemented, allowing the project to maintain the quality of the overall release without delays because of one or two features that are not ready yet.

8.4.1 Release Phases Overview

Each release cycle consists of a *development phase* followed by a *stabilization phase*. Release candidates will be created during the stabilization phase.

During the development phase, all sorts of changes can be merged. This is called the *merge window* for the main branch for this reason. During the stabilization phase, the project and its maintainers focus on fixing bugs and documenting the release. During this time, new features are generally not accepted. The main branch is in *feature freeze* during the stabilization phase.

The milestones for the current major version can be found on the Official GitHub Wiki. Information on previous releases can be found here.

> **Note**
>
> After each stable release is made, a new “release branch” for that release is created.

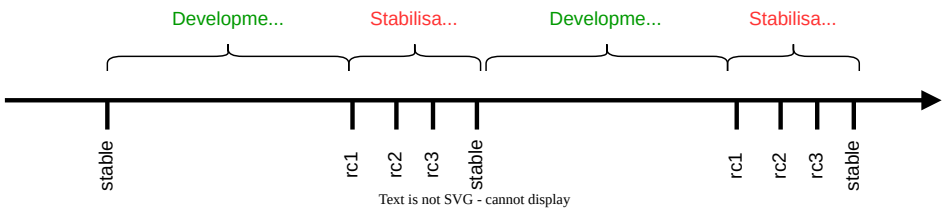


Fig. 1: Release Cycle

For example, after Zephyr stable release v4.4.0 was announced, a new branch `v4.4-branch` was created. Commits intended for v4.4.x point releases (v4.4.1, v4.4.2, etc.) should be proposed with that branch as a target.

This page currently does not document the point release process in detail.

Merge Window (Development Phase)

After tagging a release, the release managers open the main branch for development for the next release cycle. We call this time the *merge window* or the *development phase* for each release. During the merge window, any code which meets the *Pull Request Merge Criteria* can be merged into the main branch. Most changes for each release are made during its merge window.

Each release's merge window lasts approximately five months.

After the merge window, the release team tags the first *release candidate* for the upcoming release. At that point, the main branch is closed for most feature development, and the project is in *feature freeze* for the next release. Feature freeze is described in the next section.

Here is an example timeline for a merge window:

- the merge window opened when v4.3.0 was tagged on 13 November 2025. At this point, the main branch was open for merges for the next release, which was v4.4.0.
- the merge window for v4.4.0 closed when v4.4.0-rc1 was tagged on 24 March 2026. At this time, the project entered feature freeze for the v4.4.0 release.

Feature Freeze (Stabilization Phase)

When the first release candidate is tagged, the release's merge window is closed. This is called the *feature freeze* or *stabilization phase* for that release. During feature freeze, no new features will be merged for that release without a *Technical Steering Committee (TSC)* approved exception.

Please be patient during the stabilization phase while waiting for pull request review for new features.

As mentioned, the exception process for merging features during the feature freeze requires TSC approval and a justification. As a general rule, if your pull request is not yet merged and feature freeze is declared, the best thing to do is to wait for the next *merge window* to open.

If you believe your feature merits an exception and should be merged during feature freeze, contact the TSC. See the TSC wiki page for general contact information.

Here is an example timeline for a feature freeze:

- The feature freeze for v4.4.0 began when v4.4.0-rc1 was tagged on 24 March 2026. At that time, new features required a TSC exception in order to be merged to the main branch.
- The feature freeze for v4.4.0 ended when v4.4.0 was tagged on 14 April 2026. At that time, the main branch was open again for the next release's *merge window*.

The following sections describe the feature freeze in more detail.

Allowed Changes After feature freeze, **only** stabilization-related changes **shall** be accepted:

- Bug fixes addressing functional defects
- Stabilization fixes improving reliability, build stability, or test behavior
- Documentation updates (clarifications, corrections, improvements)
- Tests for existing functionality (no new feature enablement)

For more details on when certain changes are accepted, see *table*.

Exceptions to these allowed changes are rare and require both explicit justification and TSC approval. As a general rule, if a feature misses the development phase window, it should wait for the next development cycle.

Disallowed Changes The following changes **shall not** be accepted after feature freeze:

- New features or feature enhancements
- API changes, unless required to resolve a critical defect
- Refactoring not directly tied to a bug fix or stabilization effort
- New subsystems, drivers, boards, or architectural changes

8.4.2 Issue Tracking during Feature Freeze

Referencing a GitHub issue in a pull request is **recommended** but **not required** during feature freeze.

Pull requests **must** include a clear description of:

- The issue being addressed
- The impact of the issue
- How the change resolves the issue

Maintainers and release engineers **may** request additional clarification when the intent or scope of a pull request is not clear.

Mandatory Issue Usage

A GitHub issue **shall** be required in the following cases:

- Release blockers
- Fixes targeting maintenance of already released branches

Review and Approval

All pull requests **shall** follow the standard review and approval process.

Maintainers **shall** ensure that:

- Changes comply with this policy
- Scope is minimal and focused on stabilization
- Risk is appropriate for the release phase

Release engineers **may**:

- Reject or defer changes based on risk assessment
- Request changes or additional justification

Risk Expectations

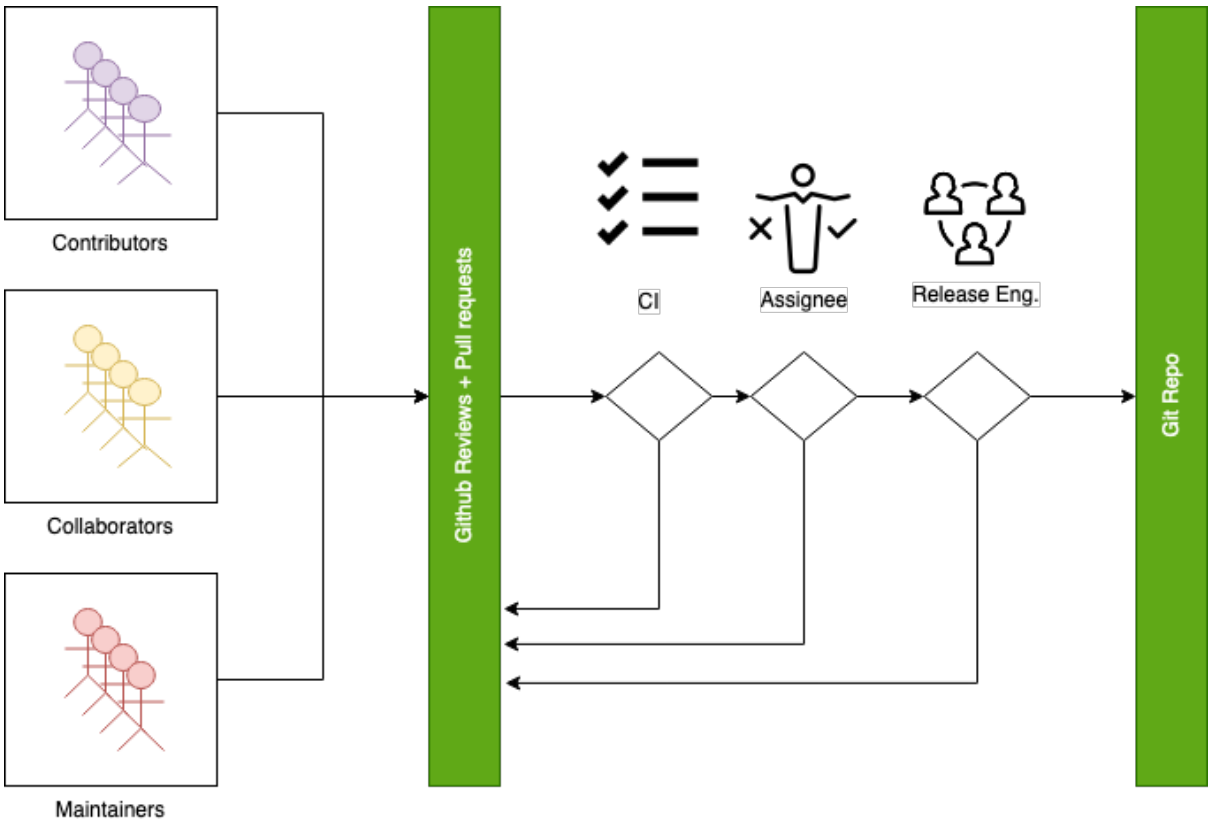
Changes accepted during stabilization **shall** be:

- Minimal in scope
- Low risk
- Adequately tested

Changes introducing significant risk **may** be deferred to a future release, even if they qualify as bug fixes.

8.4.3 Pull Request Merge Criteria

The following criteria apply during all release phases.



- All *PR Requirements* must be met.
- Minimal of 2 approvals, including an approval by the designated assignee.
- Pull requests should be reviewed by at least a maintainer or collaborator of each affected area; Unless the changes to a given area are considered trivial enough, in which case approvals by other affected subsystems maintainers/collaborators would suffice.
- Four eye principle on the organisation level. We already require at least 2 approvals (basic four eye principle), however, such reviews and approvals might be unintentionally biased in the case where the submitter is from the same organisation as the approvers. To allow for project wide review and approvals, the merge criteria is extended with the guidelines below:
  - Changes or additions to common and shared code shall have approvals from different organisations (at least one approval from an organisation different than the submitters'). Common and shared code is defined as anything that does not fall under `soc`, `boards` and `drivers/*/*`.

  - Changes or additions to hardware support (driver, SoC, boards) shall at least have the merger be from a different organisation. This applies only to implementation of an API supporting vendor specific hardware and not the APIs.
  - Release engineers may make exceptions for areas with contributions primarily coming from one organisation and where reviews from other organisations are not possible, however, merges shall be completed by a person from a different organisation. In such cases, the minimum review period of at least 2 days shall be strictly followed to allow for additional reviews.
  - Release engineers shall not merge code changes originating and reviewed only by their own organisation. To be able to merge such changes, at least one review shall be from a different organisation.
- A minimum review period of 2 business days, 4 hours for trivial changes (see *Give reviewers time to review before code merge*).
- Hotfixes can be merged at any time after CI has passed and are excluded from most of the conditions listed above.
- All required checks are passing:
  - Device Tree
  - Documentation
  - Code linters (Gitlint, Pylint, Ruff, Sphinx, etc.)
  - Identity/Emails
  - Kconfig
  - License checks
  - Checkpatch (Coding Style)
  - Integration Tests (Via twister) on emulation/simulation platforms
  - Simulated Bluetooth Tests

8.4.4 Release Quality Criteria

The main motivation is to clearly have the criteria in place that must be met for a release. This will help define when a release is "done" in terms that most people can understand and in ways that help new people to understand the process and participate in creating successful releases:

- The release criteria documents all the requirements of our target audience for each Zephyr release
- The target audiences for each release can be different, and may overlap
- The criteria at any given time are not set in stone: there may be requirements that have been overlooked, or that are new, and in these cases, the criteria should be expanded to ensure all needs are covered.

Below is the high level criteria to be met for each release:

- No blocker bugs / blocking issues
- All relevant tests shall pass on `Tier 0` platforms
- All relevant tests shall pass on Tier 0 and 1 platforms (at least 1 per architecture/architecture variant/Hardware features)
- All applicable samples/tests shall build on Tiers 0, 1 and 2
- All high and critical static analysis and security issues addressed
- Release Notes are up-to-date.

Blocker Bugs

Blocker bug process kicks in during the release process and is in effect after the feature freeze milestone. An issue labeled as a blocker practically blocks a release from happening. All blocker bugs shall be resolved before a release is created.

A fix for a bug that is granted `blocker` status can be merged to 'main' and included in the release all the way until the final release date.

Bugs of moderate severity and higher that have impact on all users are typically the candidates to be promoted to blocker bugs

Contributors and member of the release engineering team shall follow these guidelines for release blocker bugs:

- Only mark bugs as blockers if the software (Zephyr) must not be released with the bug present.
- All collaborators can add or remove blocking labels.
- Evaluate bugs as potential blockers based on their severity and prevalence.
- Provide detailed rationale whenever adding or removing a blocking label.
- Ensure all blockers have the milestone tagged.
- Release managers have final say on blocking status; contact them with any questions.

8.4.5 Release Milestones

This table documents milestones associated with each release.

Table 2: Release Milestones

Timeline	Checkpoint	Description	Owner
T-5M	Planning	Finalize dates for release, Assign release owner and agree on project wide goals for this release.	TSC
T-7W	Review target milestones	Finalize target milestones for features in flight.	Release Engineering
T-5W	Release Announcement	Release owner announces feature freeze and timeline for release.	Release Manager
T-4W	Release Timeline reminder	Release owner sends a reminder of the feature freeze and timeline for release.	Release Manager
T-3W	Feature Freeze (RC1)	After RC1, no new features may be introduced. Only stabilization, cosmetic updates, bug fixes, documentation improvements, and new tests for existing features are permitted.	Release Engineering
T-2W	2nd Release Candidate	No new features after RC2, ONLY stabilization and cosmetic changes, bug and doc fixes are allowed.	Release Manager
T-1W	Hard Freeze (RC3)	Only blocker bug fixes after RC3, documentation improvements and changes to release notes are allowed. Release notes need to be complete by this checkpoint. Release Criteria is met.	Release Manager
T-0W	Release		Release Manager

8.4.6 Long Term Support (LTS) Releases

Long-term support releases are designed to be supported and maintained for an extended period and are the recommended release for products and the auditable branch used for certification.

An LTS release is defined as:

- **Product focused**
- **Extended Stabilisation period**: Allow for more testing and bug fixing
- **Stable APIs**
- **Quality Driven Process**
- **Long Term**: Maintained for an extended period of time (at least 5 years).

Product Focused

Zephyr LTS is the recommended release for product makers with an extended support and maintenance which includes general stability and bug fixes, security fixes.

An LTS includes both mature and new features. API and feature maturity is documented and tracked. The footprint and scope of mature and stable APIs expands as we move from one LTS to the next giving users access to bleeding edge features and new hardware while keeping a stable foundation that evolves over time.

Extended Stabilisation Period

Zephyr LTS development cycle differs from regular releases and has an extended stabilization period. Feature freeze of regular releases happens 3-4 weeks before the scheduled release date. The stabilization period for LTS is extended by 3 weeks with the feature freeze occurring 6-7 weeks before the anticipated release date. The time between code freeze and release date is extended in this case.

Stable APIs

Zephyr LTS provides a stable and long-lived foundation for developing products. To guarantee stability of the APIs and the implementation of such APIs it is required that any release software that makes the core of the OS went through the Zephyr API lifecycle and stabilized over at least 2 releases. This guarantees that we release many of the highlighted and core features with mature and well-established implementations with stable APIs that are supported during the lifetime of the release LTS.

- API Freeze (LTS - 2)
  - All stable APIs need to be frozen 2 releases before an LTS. APIs can be extended with additional features, but the core implementation is not modified. This is valid for the following subsystems for example:
    - Device Drivers (i2c.h, spi.h)...
    - Kernel (k_*):
    - OS services (logging,debugging, ..)
    - DTS: API and bindings stability
    - Kconfig
  - New APIs for experimental features can be added at any time as long as they are standalone and documented as experimental or unstable features/APIs.
- Feature Freeze (LTS - 1) - No new features or overhaul/restructuring of code covering major LTS features.
  - Kernel + Base OS

  - Additional advertised LTS features
  - Auxiliary features on top of and/or extending the base OS and advertised LTS features can be added at any time and should be marked as experimental if applicable

Quality Driven Process

The Zephyr project follows industry standards and processes with the goal of providing a quality oriented releases. This is achieved by providing the following products to track progress, integrity and quality of the software components provided by the project:

- Compliance with published coding guidelines, style guides and naming conventions and documentation of deviations.
- Static analysis reports
  - Regular static analysis on the complete tree using available commercial and open-source tools, and documentation of deviations and false positives.
- Documented components and APIS
- Requirements Catalog
- Verification Plans
- Verification Reports
- Coverage Reports
- Requirements Traceability Matrix (RTM)
- SPDX License Reports

Each release is created with the above products to document the quality and the state of the software when it was released.

Long Term Support and Maintenance

LTS releases are published every 2.5 to 3 years and are branched and maintained independently from the main tree for approximately 5 years after they were released.

Support is provided in three main phases:

- **Phase 1 (first 2 years):** General bug fixes and security fixes, including platform and driver fixes.
- **Phase 2 (following 3+ years):** Security and OS stability fixes only.
- **Phase 3:** Extended support may be available through third parties (details to be determined).

Support for a given LTS release (LTS *N*) continues until the initial release of the LTS two versions ahead (LTS *N+2*). A final release of LTS *N* occurs shortly after the initial release of LTS *N+2*.

The list of currently supported LTS releases and their EOL dates can be found here.

Fig. 2: Long Term Support Release

Changes and fixes flow in both directions. However, changes from main branch to an LTS branch will be limited to fixes that apply to both branches and for existing features only.

All fixes for an LTS branch that apply to the mainline tree shall be submitted to mainline tree as well.

Auditable Code Base

An auditable code base is to be established from a defined subset of Zephyr OS features and will be limited in scope. The LTS, development tree, and the auditable code bases shall be kept in sync after the audit branch is created, but with a more rigorous process in place for adding new features into the audit branch used for certification.

This process will be applied before new features move into the auditable code base.

The initial and subsequent certification targets will be decided by the Zephyr project governing board.

Processes to achieve selected certification will be determined by the Security and Safety Working Groups and coordinated with the TSC.

8.4.7 Host Tools Support Policy

This section documents policies related to host tools required to develop zephyr applications.

Python Version Policy

Zephyr tracks the Python upstream release schedule when determining its minimum supported Python version.

The minimum supported version is selected to balance long-term support and practical availability in mainstream, actively supported Linux distributions (e.g. current Ubuntu LTS and Debian Stable releases).

A Python version is considered for removal when:

- It is approaching or has reached upstream end-of-life (EOL), and
- Newer Python versions are broadly available in current, actively supported major Linux distributions.

8.4.8 Hardware Support Tiers

This section documents a rough set of tiers associated with hardware platforms. These criteria are not currently formally enforced or evaluated at a board, architecture, or SoC level.

Tier 0: Emulation Platforms

- Tests are both built and run in these platforms in CI, and therefore runtime failures can block Pull Requests.
- Supported by the Zephyr project itself, commitment to fix bugs in releases.
- One Tier 0 platform is required for each new architecture.
- Bugs reported against platforms of this tier are to be evaluated and treated as a general bug in Zephyr and should be dealt with the highest priority.

Tier 1: Supported Platforms

- Commitment from a specific team to run tests using twister device testing for the “Zephyr compatibility test suite” (details TBD) on a regular basis using open-source and publicly available drivers.
- Commitment to fix bugs in time for releases. Not supported by “Zephyr Project” itself.
- General availability for purchase
- Bugs reported against platforms of this tier are to be evaluated and treated as a general bug in Zephyr and should be dealt with medium to high priority.

Tier 2: Community Platforms

- Platform implementation is available in upstream, no commitment to testing, may not be generally available.
- Has a dedicated maintainer who commits to respond to issues / review patches.
- Bugs reported against platforms of this tier are NOT considered as a general bug in Zephyr.

Tier 3: Deprecated and unsupported Platforms

- Platform implementation is available, but no owner or unresponsive owner.
- No commitment to support is available.
- May be removed from upstream if no one works to bring it up to tier 2 or better.
- Bugs reported against platforms of this tier are NOT considered as a general bug in Zephyr.

8.4.9 Release Procedure

This section documents the Release manager responsibilities so that it serves as a knowledge repository for Release managers.

Release Checklist

Each release has a GitHub issue associated with it that contains the full checklist. After a release is complete, a checklist for the next release is created.

Tagging

The following syntax should be used for releases and tags in Git:

- Release [Major].[Minor].[Patch Level]
- Release Candidate [Major].[Minor].[Patch Level]-rc[RC Number]
- Tagging:
  - v[Major].[Minor].[Patch Level]-rc[RC Number]
  - v[Major].[Minor].[Patch Level]
  - v[Major].[Minor].99 - A tag applied to main branch to signify that work on v[Major].[Minor+1] has started. For example, v1.7.99 will be tagged at the start of v1.8 process. The tag corresponds to VERSION_MAJOR/VERSION_MINOR/PATCHLEVEL macros as defined for a work-in-progress main branch version. Presence of this tag allows generation of sensible output for "git describe" on main branch, as typically used for automated builds and CI tools.

The final release and each release candidate shall be tagged using the following steps:

> **Note**
>
> Tagging needs to be done via explicit git commands and not via GitHub's release interface. The GitHub release interface does not generate annotated tags (it generates 'lightweight' tags regardless of release or pre-release). You should also upload your gpg public key to your GitHub account, since the instructions below involve creating signed tags. Alternatively, if you don't have a gpg key, you can also use your ssh key for signing.

Release Candidate

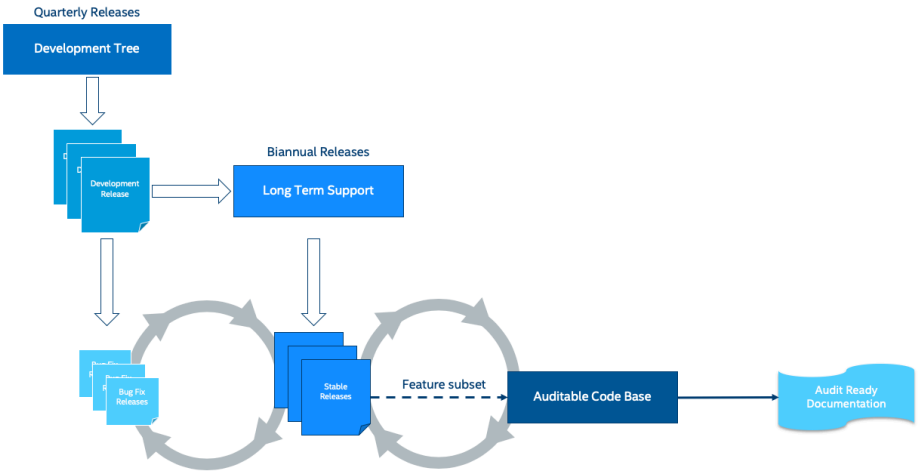


Fig. 3: Zephyr Code and Releases

> **Note**
>
> This section uses tagging 1.11.0-rc1 as an example, replace with the appropriate release candidate version.

1. Update the version variables in the VERSION file located in the root of the Git repository to match the version for this release candidate. The `EXTRAVERSION` variable is used to identify the rc[RC Number] value for this candidate:

```
EXTRAVERSION = rc1
```

2. Post a PR with the updated VERSION file using `release: Zephyr 1.11.0-rc1` as the commit subject. Merge the PR after successful CI.
3. Tag and push the version, using an annotated tag:

```
$ git pull
$ git tag -s -m "Zephyr 1.11.0-rc1" v1.11.0-rc1
```

4. Verify that the tag has been signed correctly, `git show` for the tag must contain a signature (look for the `BEGIN PGP SIGNATURE` or `BEGIN SSH SIGNATURE` marker in the output):

```
$ git show v1.11.0-rc1
```

5. Push the tag:

```
$ git push git@github.com:zephyrproject-rtos/zephyr.git v1.11.0-rc1
```

6. Send an email to the mailing lists (`announce` and `devel`) with a link to the release

Final Release

> **Note**
>
> This section uses tagging 1.11.0 as an example, replace with the appropriate final release version.

When all final release criteria has been met and the final release notes have been approved and merged into the repository, the final release version will be set and repository tagged using the following procedure:

1. Update the version variables in the VERSION file located in the root of the Git repository. Set `EXTRAVERSION` variable to an empty string to indicate final release:

```
EXTRAVERSION =
```

2. Post a PR with the updated VERSION file using `release: Zephyr 1.11.0` as the commit subject. Merge the PR after successful CI.
3. Tag and push the version, using two annotated tags:

```
$ git pull
$ git tag -s -m "Zephyr 1.11.0" v1.11.0
```

4. Verify that the tag has been signed correctly, `git show` for the tag must contain a signature (look for the `BEGIN PGP SIGNATURE` or `BEGIN SSH SIGNATURE` marker in the output):

```
$ git show v1.11.0
```

5. Push the tag:

```
$ git push git@github.com:zephyrproject-rtos/zephyr.git v1.11.0
```

6. Find the new `v1.11.0` tag at the top of the releases page and edit the release with the `Edit tag` button with the following:
   - Copy the overview of `docs/releases/release-notes-1.11.rst` into the release notes textbox and link to the full release notes file on docs.zephyrproject.org.
7. Send an email to the mailing lists (`announce` and `devel`) with a link to the release

8.5 Feature Tracking

To propose something new to the Zephyr Project:

1. Start by searching existing issues to see if there is prior work on the area.
2. If not, feel free to create a new issue of the appropriate type (Enhancement, RFC/Proposal, etc.)

You should contact an assigned owner if you'd like to discuss or contribute to an existing issue's implementation.

> **Note**
>
> Previously, the project used GitHub labels to track feature-related work. You may find issues with labels like "Feature" or "Enhancement".
>
> These labels are deprecated for new issues, which should be of the appropriate type instead. For example, create an issue of type Enhancement instead of adding the Enhancement label to a new issue.

8.5.1 Roadmap and Release Plans

Project roadmaps and release plans are both important tools for the project, but they have very different purposes and should not be confused. A project roadmap communicates the high-level overview of a project's strategy, while a release plan is a tactical document designed to capture and track the features planned for upcoming releases.

- The project roadmap communicates the why; a release plan details the what
- A release plan spans only a few months; a product roadmap might cover a year or more

Project Roadmap

The project roadmap should serve as a high-level, visual summary of the project's strategic objectives and expectations.

If built properly, the roadmap can be a valuable tool for several reasons. It can help the project present its plan in a compelling way to existing and new stakeholders, to help recruit new members and it can be a helpful resource the team and community can refer to throughout the project's development, to ensure they are still executing according to plan.

As such, the roadmap should contain only strategic-level details, major project themes, epics, and goals.

Release Plans

The release plan comes into play when the project roadmap's high-level strategy is translated into an actionable plan built on specific features, enhancements, and fixes that need to go into a specific release or milestone.

The release plan communicates those features and enhancements slated for your project' next release (or the next few releases). So it acts as more of a project plan, breaking the big ideas down into smaller projects the community and main stakeholders of the project can make progress on.

8.6 Code Flow and Branches

8.6.1 Introduction

The zephyr Git repository has three types of branches:

main
: Which contains the latest state of development

collab-*
: Collaboration branches that are used for shared development of new features to be introduced into the main branch when ready. Creating a new collaboration branch requires a justification and TSC approval. Collaboration branches shall be based off the main branch and any changes developed in the collab branch shall target the main development branch. For released versions of Zephyr, the introduction of fixes and new features, if approved by the TSC, shall be done using backport pull requests.

vx.y-branch
: Branches which track maintenance releases based on a major release

Development in collaboration branches before features go to mainline allows teams to work independently on a subsystem or a feature, improves efficiency and turnaround time, and encourages collaboration and streamlines communication between developers.

Changes submitted to a collaboration branch can evolve and improve incrementally in a branch, before they are submitted to the mainline tree for final integration.

By dedicating an isolated branch to complex features, it's possible to initiate in-depth discussions around new additions before integrating them into the official project.

Collaboration branches are ephemeral and shall be removed once the collaboration work has been completed. When a branch is requested, the proposal should include the following:

- Define exit criteria for merging the collaboration branch changes back into the main branch.
- Define a timeline for the expected life cycle of the branch. It is recommended to select a Zephyr release to set the timeline. Extensions to this timeline requires TSC approval.

8.6.2 Roles and Responsibilities

Collaboration branch owners have the following responsibilities:

- Use the infrastructure and tools provided by the project (GitHub, Git)
- All changes to collaboration branches shall come in form of github pull requests.
- Force pushing a collaboration branch is only allowed when rebasing against the main branch.
- Review changes coming from team members and request review from branch owners when submitting changes.
- Keep the branch in sync with upstream and update on a regular basis.
- Push changes frequently to upstream using the following methods:
  - GitHub pull requests: for example, when reviews have not been done in the local branch (one-man branch).
  - Merge requests: When a set of changes has been done in a local branch and has been reviewed and tested in a collaboration branch.

8.7 Development Environment and Tools

8.7.1 Code Review

GitHub is intended to provide a framework for reviewing every commit before it is accepted into the code base. Changes, in the form of Pull Requests (PR) are uploaded to GitHub but don't actually become a part of the project until they've been reviewed, passed a series of checks (CI), and are approved by maintainers. GitHub is used to support the standard open source practice of submitting patches, which are then reviewed by the project members before being applied to the code base.

Pull requests should be appropriately *labeled*, and linked to any relevant *bug or feature tracking issues* .

The Zephyr project uses GitHub for code reviews and Git tree management. When submitting a change or an enhancement to any Zephyr component, a developer should use GitHub. GitHub Actions automatically assigns a responsible reviewer on a component basis, as defined in the MAINTAINERS.yml file stored with the code tree in the Zephyr project repository. A limited set of release managers are allowed to merge a pull request into the main branch once reviews are complete.

Give reviewers time to review before code merge

The Zephyr project is a global project that is not tied to a certain geography or timezone. We have developers and contributors from across the globe. When changes are proposed using pull request, we need to allow for a minimal review time to give developers and contributors the opportunity to review and comment on changes. There are different categories of changes and we know that some changes do require reviews by subject matter experts and owners of the subsystem being changed. Many changes fall under the "trivial" category that can be addressed with general reviews and do not need to be queued for a maintainer or code-owner review. Additionally, some changes might require further discussions and a decision by the TSC or the Security working group. To summarize the above, the diagram below proposes minimal review times for each category:

Workflow

- An author of a change can suggest in his pull-request which category a change should belong to. A project maintainers or TSC member monitoring the inflow of changes can change

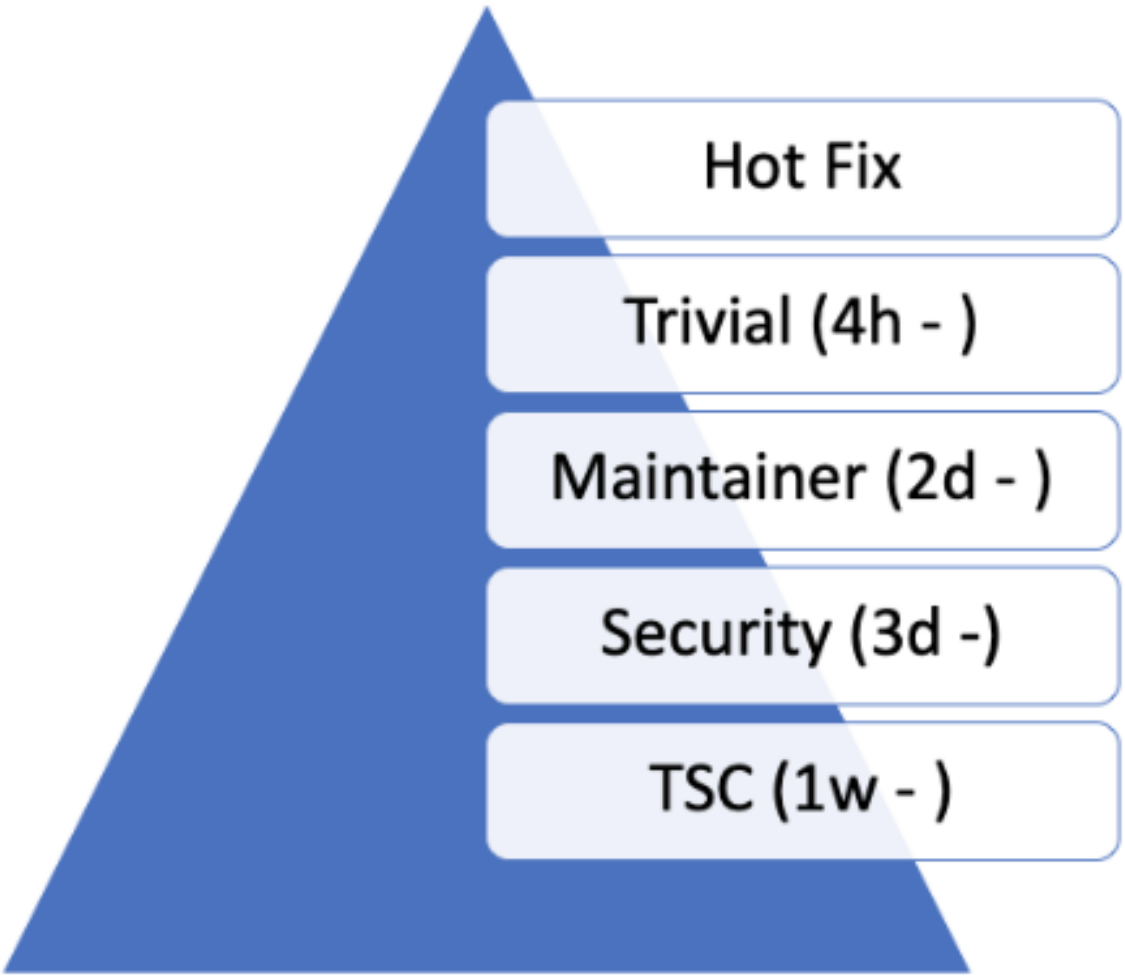


Fig. 4: Pull request classes

the label of a pull request by adding a comment justifying why a change should belong to another category.

- The project will use the label system to categorize the pull requests.
- Changes should not be merged before the minimal time has expired.

Categories/Labels

Hotfix
Any change that is a fix to an issue that blocks developers from doing their daily work, for example CI breakage, Test breakage, Minor documentation fixes that impact the user experience.

Such fixes can be merged at any time after they have passed CI checks. Depending on the fix, severity, and availability of someone to review them (other than the author) they can be merged with justification without review by one of the project owners.

Trivial
Trivial changes are those that appear obvious enough and do not require maintainer or code-owner involvement. Such changes should not change the logic or the design of a subsystem or component. For example a trivial change can be:

- Documentation changes
- Configuration changes
- Minor Build System tweaks
- Minor optimization to code logic without changing the logic
- Test changes and fixes
- Sample modifications to support additional configuration or boards etc.

Maintainer
Any changes that touch the logic or the original design of a subsystem or component will need to be reviewed by the code owner or the designated subsystem maintainer. If the code changes is initiated by a contributor or developer other than the owner the pull request needs to be assigned to the code owner who will have to drive the pull request to a mergeable state by giving feedback to the author and asking for more reviews from other developers.

Security
Changes that appear to have an impact to the overall security of the system need to be reviewed by a security expert from the security working group.

TSC and Working Groups
Changes that introduce new features or functionality or change the way the overall system works need to be reviewed by the TSC or the responsible Working Group. For example for *breaking API changes*, the proposal needs to be presented in the Architecture meeting so that the relevant stakeholders are made aware of the change.

A Pull-Request should have an Assignee

- An assignee to a pull request should not be the same as the author of the pull-request
- An assignee to a pull request is responsible for driving the pull request to a mergeable state
- An assignee is responsible for dismissing stale reviews and seeking reviews from additional developers and contributors
- Pull requests should not be merged without an approval by the assignee.

Pull Request should not be merged by author without review

All pull requests need to be reviewed and should not be merged by the author without a review. The following exceptions apply:

- Hot fixes: Fixing CI issues, reverts, and system breakage
- Release related changes: Changing version file, applying tags and release related activities without any code changes.

Developers and contributors should always seek review, however there are cases when reviewers are not available and there is a need to get a code change into the tree as soon as possible.

Reviewers shall not 'Request Changes' without comments or justification

Any change requests (-1) on a pull request have to be justified. A reviewer should avoid blocking a pull-request with no justification. If a reviewer feels that a change should not be merged without their review, then: Request change of the category: for example:

- Trivial -> Maintainer
- Assign Pull Request to yourself, this will mean that a pull request should not be merged without your approval.

Pull Requests should have at least 2 approvals before they are merged

A pull-request shall be merged only with two positive reviews (approval). Beside the person merging the pull-request (merging != approval), two additional approvals are required to be able to merge a pull request. The person merging the request can merge without approving or approve and merge to get to the 2 approvals required.

Reviewers should keep track of pull requests they have provided feedback to

If a reviewer has requested changes in a pull request, he or she should monitor the state of the pull request and/or respond to mention requests to see if his feedback has been addressed. Failing to do so, negative reviews shall be dismissed by the assignee or an owner of the repository. Reviews will be dismissed following the criteria below:

- The feedback or concerns were visibly addressed by the author
- The reviewer did not revisit the pull request after 2 week and multiple pings by the author
- The review is unrelated to the code change or asking for unjustified structural changes such as:
  - Split the PR
  - Can you fix this unrelated code that happens to appear in the diff
  - Can you fix unrelated issues
  - Etc.

Closing Stale Issues and Pull Requests

- The Pull requests and issues sections on Github are NOT discussion forums. They are items that we need to execute and drive to closure. Use the mailing lists for discussions.
- In case of both issues and pull-requests the original poster needs to respond to questions and provide clarifications regarding the issue or the change. After one week without a response to a request, a second attempt to elicit a response from the contributor will be made. After one more week without a response the item may be closed (draft and DNM tagged pull requests are excluded).

8.7.2 Continuous Integration

All changes submitted to GitHub are subject to tests that are run on emulated platforms and architectures to identify breakage and regressions that can be immediately identified. Testing using Twister additionally performs build tests of all boards and platforms. Documentation changes are also verified through review and build testing to verify doc generation will be successful.

Any failures found during the CI test run will result in a negative review assigned automatically by the CI system. Developers are expected to fix issues and rework their patches and submit again.

The CI infrastructure currently runs the following tests:

- Run `checkpatch` for code style issues (can vote -1 on errors; see note)
- Gitlint: Git commit style based on project requirements
- License Check: Check for conflicting licenses
- Run `twister` script
  - Run kernel tests in QEMU (can vote -1 on errors)
  - Build various samples for different boards (can vote -1 on errors)
- Verify documentation builds correctly.

Note

`checkpatch` is a Perl script that uses regular expressions to extract information that requires a C language parser to process accurately. As such it sometimes issues false positives. Known cases include constructs like:

```
static uint8_t __aligned(PAGE_SIZE) page_pool[PAGE_SIZE * POOL_PAGES];
IOPCTL_Type *base = config->base;
```

Both lines produce a diagnostic regarding spaces around the `*` operator: the first is misidentified as a pointer type declaration that would be correct as `PAGE_SIZE *POOL_PAGES` while the second is misidentified as a multiplication expression that would be correct as `IOPCTL_Type * base`.

Maintainers can override the -1 in cases where the CI infrastructure gets the wrong answer.

8.7.3 Labeling issues and pull requests in GitHub

The project uses GitHub issues and pull requests (PRs) to track and manage daily and long-term work and contributions to the Zephyr project. We use GitHub **labels** to classify and organize these issues and PRs by area, type, priority, and more, making it easier to find and report on relevant items.

All GitHub issues or pull requests must be appropriately labeled. Issues and PRs often have multiple labels assigned, to help classify them in the different available categories. When reviewing a PR, if it has missing or incorrect labels, maintainers shall fix it.

This saves us all time when searching, reduces the chances of the PR or issue being forgotten, speeds up reviewing, avoids duplicate issue reports, etc.

These are the labels we currently have, grouped by applicability:

Labels applicable to issues only

Label	Description
priority: {high|medium|low}	To classify the impact and importance of a bug. Note: Issue priorities are generally set or changed during the bug-triage or TSC meetings.
Regression	Something, which was working, but does not anymore (bug subtype).
Duplicate	This issue is a duplicate of another issue (please specify).
Good first issue	Good for a first time contributor to take.
Release Notes	Issues that need to be mentioned in release notes as known issues with additional information.

Labels applicable to pull requests only

The issue or PR describes a change to a stable API.

Label	Description
Hotfix	Fix for an issue blocking development.
Trivial	Simple changes that can have shorter review time and be reviewed by anyone, i.e. typos, straightforward one-liner bug fixes, etc.
Maintainer	Maintainer review required.
Security Review	To be reviewed by a security expert.
DNM	This PR should not be merged (Do Not Merge). For work in progress, GitHub "draft" PRs are preferred.
Needs review	The PR needs attention from the maintainers.
Backport	The PR is a backport or should be backported.
Licensing	The PR has licensing issues which require a licensing expert to review it.

> **Note**
>
> For all labels applicable to PRs: Please note that the label, together with PR complexity, affects how long a merge should be held to ensure proper review. See *review process* for details.

Labels applicable to both pull requests and issues

Label	Description
area: {area-name}	Indicates Zephyr subsystems (e.g, *area: Kernel*, *area: I2C*, *area: Memory Management*), project functions (e.g., *area: Debugging*, *area: Documentation*, *area: Process*), or other categories (e.g., *area: Coding Style*, *area: MISRA-C*) affected by the bug or the pull request. An area maintainer should be able to filter by an area label and find all issues and PRs which relate to that area.
platform: {platform name}	An issue or PR which affects only a particular platform.
TSC	TSC stands for Technical Steering Committee. The issue is to be discussed in the following TSC meeting if time permits.
Breaking API Change	The issue or PR describes a breaking change to a stable API. See additional information in *Introducing breaking API changes*.
bug	The issue is a bug, or the PR is fixing a bug.
Coverity	A Coverity detected issue or its fix.
Waiting for response	The Zephyr developers are waiting for the submitter to respond to a question, or address an issue.
Blocked	Blocked by another PR or issue.
Stale	An issue or a PR which seems abandoned, and requires attention by the author.
In progress	For PRs: is work in progress and should not be merged yet. For issues: Is being worked on.
RFC	The author would like input from the community. For a PR it should be considered a draft.
LTS	Long term release branch related.
EXT	Related to an external component.

8.8 Bug Reporting

To maintain traceability and relation between proposals, changes, features, and issues, it is recommended to cross-reference source code commits with the relevant GitHub issues and vice versa. Any changes that originate from a tracked feature or issue should contain a reference to the feature by mentioning the corresponding issue or pull-request identifiers.

At any time it should be possible to establish the origin of a change and the reason behind it by following the references in the code.

8.8.1 Reporting a regression issue

It could happen that the issue being reported is identified as a regression, as the use case is known to be working on earlier commit or release. In this case, providing directly the guilty commit when submitting the bug gains a lot of time in the eventual bug fixing.

To identify the commit causing the regression, several methods could be used, but tree bisecting method is an efficient one that doesn't require deep code expertise and can be used by every

one.

For this, git bisect is the recommended tool.

Recommendations on the process:

- Run `west update` on each bisection step.
- Once the bisection is over and a culprit identified, verify manually the result.

8.9 Communication and Collaboration

The Zephyr Discord Server is the primary chat forum used by Zephyr developers, contributors, and users.

The Zephyr project mailing lists are used as an additional communication tool by project members, contributors, and the community. There are specialized mailing lists for specific interests. Several lists are public and open. Mailing lists are always available for use in situations where Discord is unavailable or an unsuitable forum.

In general, bug reports and other issues should be reported as GitHub Issues and not broadcasted to the mailing list. The same applies to code reviews.

8.10 Code Documentation

8.10.1 API Documentation

Well documented APIs enhance the experience for developers and are an essential requirement for defining an API's success. Doxygen is a general purpose documentation tool that the zephyr project uses for documenting APIs. It generates either an on-line documentation browser (in HTML) and/or provides input for other tools that is used to generate a reference manual from documented source files. In particular, doxygen's XML output is used as an input when producing the Zephyr project's online documentation.

8.10.2 Reference to Requirements

APIs for the most part document the implementation of requirements or advertised features and can be traced back to features. We use the API documentation as the main interface to trace implementation back to documented features. This is done using doxygen's native requirements traceability commands (`@satisfies` and `@verifies`) that reference requirements maintained somewhere else in a requirement catalogue.

8.10.3 Test Documentation

To help understand what each test does and which functionality it tests we also document all test code using the same tools and in the same context and generate documentation for all unit and integration tests maintained in the same environment. Tests are documented using references to the APIs or functionality they validate by creating a link back to the APIs and by adding a reference to the original requirements.

8.10.4 Documentation Guidelines

Test Code

The Zephyr project uses several test methodologies, the most common being the *Ztest framework*. Test documentation should only be done on the entry test functions (usually prefixed with test_) and those that are called directly by the Ztest framework. Those tests are going to appear in test reports and using their name and identifier is the best way to identify them and trace back to them from requirements.

Test documentation should not interfere with the actual API documentation and needs to follow a new structure to avoid confusion. Using a consistent naming scheme and following a well-defined structure we will be able to group this documentation in its own module and identify it uniquely when parsing test data for traceability reports. Here are a few guidelines to be followed:

- All test code documentation should be grouped under the `all_tests` doxygen group
- All test documentation should be under doxygen groups that are prefixed with tests_

Documenting a Requirement

A requirement is documented with the doxygen `@requirement` command, which takes a unique identifier and an optional title. Doxygen collects all requirements on a single page and allows them to be referenced from anywhere by their identifier. Use a comment block dedicated to the requirement, typically maintained in the requirement catalogue:

```
/**
* @requirement ZEPH-015 (Give a semaphore)
*
* The kernel shall provide a mechanism to give a semaphore, increasing its
* count up to its configured maximum.
*/
```

The identifier used here (for example `ZEPH-015`) is the same one referenced by the `@satisfies` and `@verifies` commands below, which is what links the implementation and tests back to the requirement.

The doxygen `@verifies` command signifies that a test verifies a requirement:

```
/**
* @brief Tests for the Semaphore kernel object
* @defgroup kernel_semaphore_tests Semaphore
* @ingroup all_tests
* @{
*/

...
/**
* @brief A brief description of the tests
* Some details about the test
* more details
*
* @verifies ZEPH-015
*/
void test_sema_thread2thread(void)
{
...
}
...

/**
* @}
*/
```

To get coverage of how an implementation or a piece of code satisfies a requirement, we use the doxygen `@satisfies` command:

```
/**
* @brief Give a semaphore.
```

(continues on next page)

(continued from previous page)

```
*
* This routine gives @a sem, unless the semaphore is already at its maximum
* permitted count.
*
* @note Can be called by ISRs.
*
* @param sem Address of the semaphore.
*
* @satisfies ZEPH-015
*/
__syscall void k_sem_give(struct k_sem *sem);
```

To generate the matrix, you will first need to build the documentation, specifically you will need to build the doxygen XML output:

```
$ make doxygen
```

Parse the generated XML data from doxygen to generate the traceability matrix.

The Zephyr project defines a development process workflow using GitHub **Issues** to track feature, enhancement, and bug reports together with GitHub **Pull Requests** (PRs) for submitting and reviewing changes. Zephyr community members work together to review these Issues and PRs, managing feature enhancements and quality improvements of Zephyr through its regular releases, as described in *Release Process*.

We can only manage the volume of Issues and PRs, by requiring timely reviews, feedback, and responses from the community and contributors, both for initial submissions and for followup questions and clarifications. Read about the project's *development processes and tools* and specifics about *review timelines* to learn about the project's goals and guidelines for our active developer community.

TSC Project Roles describes in detail the Zephyr project roles and associated permissions with respect to the development process workflow.

8.11 Terminology

- mainline: The main tree where the core functionality and core features are being developed.
- subsystem/feature branch: is a branch within the same repository. In our case, we will use the term branch also when referencing branches not in the same repository, which are a copy of a repository sharing the same history.
- upstream: A parent branch the source code is based on. This is the branch you pull from and push to, basically your upstream.
- LTS: Long Term Support

Chapter 9

Security

These documents describe the requirements, processes, and developer guidelines for ensuring security is addressed within the Zephyr project.

9.1 Zephyr Security Overview

9.1.1 Introduction

This document outlines the steps of the Zephyr Security Subcommittee towards a defined security process that helps developers build more secure software while addressing security compliance requirements. It presents the key ideas of the security process and outlines which documents need to be created. After the process is implemented and all supporting documents are created, this document is a top-level overview and entry point.

Overview and Scope

We begin with an overview of the Zephyr development process, which mainly focuses on security functionality.

In subsequent sections, the individual parts of the process are treated in detail. As depicted in Figure 1, these main steps are:

1. **Secure Development:** Defines the system architecture and development process that ensures adherence to relevant coding principles and quality assurance procedures.
2. **Secure Design:** Defines security procedures and implement measures to enforce them. A security architecture of the system and relevant sub-modules is created, threats are identified, and countermeasures designed. Their correct implementation and the validity of the threat models are checked by code reviews. Finally, a process shall be defined for reporting, classifying, and mitigating security issues.
3. **Security Certification:** Defines the certifiable part of the Zephyr RTOS. This includes an evaluation target, its assets, and how these assets are protected. Certification claims shall be determined and backed with appropriate evidence.

Intended Audience

This document is a guideline for the development of a security process by the Zephyr Security Subcommittee and the Zephyr Technical Steering Committee. It provides an overview of the Zephyr security process for (security) engineers and architects.

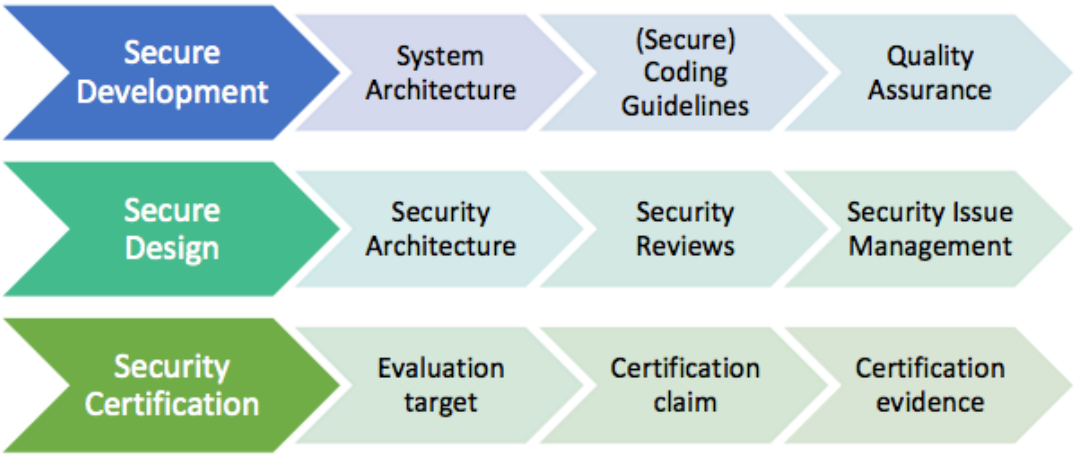


Fig. 1: Figure 1. Security Process Steps

Nomenclature

In this document, the keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in [**?**].

These words are used to define absolute requirements (or prohibitions), highly recommended requirements, and truly optional requirements. As noted in RFC-2119, "These terms are frequently used to specify behavior with security implications. The effects on security of not implementing a MUST or SHOULD, or doing something the specification says MUST NOT or SHOULD NOT be done may be very subtle. Document authors should take the time to elaborate the security implications of not following recommendations or requirements as most implementers will not have had the benefit of the experience and discussion that produced the specification."

Security Document Update

This document is a living document. As new requirements, features, and changes are identified, they will be added to this document through the following process:

1. Changes will be submitted from the interested party(ies) via pull requests to the Zephyr documentation repository.
2. The Zephyr Security Subcommittee will review these changes and provide feedback or acceptance of the changes.
3. Once accepted, these changes will become part of the document.

9.1.2 Current Security Definition

This section recapitulates the current status of secure development within the Zephyr RTOS. Currently, focus is put on functional security and code quality assurance, although additional security features are scoped.

The three major security measures currently implemented are:

- **Security Functionality** with a focus on cryptographic algorithms and protocols. Support for cryptographic hardware is scoped for future releases. The Zephyr runtime architecture is a monolithic binary and removes the need for dynamic loaders, thereby reducing the exposed attack surface.

- **Quality Assurance** is driven by using a development process that requires all code to be reviewed before being committed to the common repository. Furthermore, the reuse of proven building blocks such as network stacks increases the overall quality level and guarantees stable APIs. Static code analyses provide additional quality checks.
- **Execution Protection** including thread separation, stack and memory protection is currently available in the upstream Zephyr RTOS starting with version 1.9.0 (stack protection). Memory protection and thread separation were added in version 1.10.0 for X86 and in version 1.11.0 for ARM and ARC.

These topics are discussed in more detail in the following subsections.

Security Functionality

The security functionality in Zephyr hinges mainly on the inclusion of cryptographic algorithms, and on its monolithic system design.

The cryptographic features are provided through PSA Crypto, with Mbed TLS as the underlying implementation. Applications leverage PSA Crypto APIs, ensuring a standardized and secure approach to cryptographic operations. Mbed TLS, as the implementation of PSA Crypto, supports a wide range of cryptographic algorithms, making it suitable for various application requirements.

APIs for vendor specific cryptographic IPs in both hardware and software are planned, including secure key storage in the form of secure access modules (SAMs), Trusted Platform Modules (TPMs), and Trusted Execution Environments (TEEs).

The security architecture is based on a monolithic design where the Zephyr kernel and all applications are compiled into a single static binary. System calls are implemented as function calls without requiring context switches. Static linking eliminates the potential for dynamically loading malicious code.

Additional protection features are available in later releases. Stack protection mechanisms are provided to protect against stack overruns. In addition, applications can take advantage of thread separation features to split the system into privileged and unprivileged execution environments. Memory protection features provide the capability to partition system resources (memory, peripheral address space, etc.) and assign resources to individual threads or groups of threads. Stack, thread execution level, and memory protection constraints are enforced at the time of context switch.

Quality Assurance

The Zephyr project uses an automated quality assurance process. The goal is to have a process including mandatory code reviews, feature and issue management/tracking, and static code analyses.

Code reviews are documented and enforced using a voting system before getting checked into the repository by the responsible subsystem's maintainer. The main goals of the code review are:

- Verifying correct functionality of the implementation
- Increasing the readability and maintainability of the contributed source code
- Ensuring appropriate usage of string and memory functions
- Validation of the user input
- Reviewing the security relevant code for potential issues

The current coding principles focus mostly on coding styles and conventions. Functional correctness is ensured by the build system and the experience of the reviewer. Especially for security relevant code, concrete and detailed guidelines need to be developed and aligned with the developers (see: *Secure Coding*).

Static code analyses are run on the Zephyr code tree on a regular basis, see *Static Code Analysis*.

Bug and issue tracking and management is performed using Github. The term "survivability" was coined to cover pro-active security tasks such as security issue categorization and management. A problem identified as vulnerability is managed within Github security advisories.

Issues determined by static analyses should have more stringent reviews before they are closed as non-issues (at least another person educated in security processes need to agree on non-issue before closing).

A security subcommittee has been formed to develop a security process in more detail; this document is part of that process.

Execution Protection

Execution protection is supported and can be categorized into the following tasks:

- **Memory separation:** Memory will be partitioned into regions and assigned attributes based on the owner of that region of memory. Threads will only have access to regions they control.
- **Stack protection:** Stack guards would provide mechanisms for detecting and trapping stack overruns. Individual threads should only have access to their own stacks.
- **Thread separation:** Individual threads should only have access to their own memory resources. As threads are scheduled, only memory resources owned by that thread will be accessible. Topics such as program flow protection and other measures for tamper resistance are currently not in scope.

System Level Security (Ecosystem, ...)

System level security encompasses a wide variety of categories. Some examples of these would be:

- Secure/trusted boot
- Over the air (OTA) updates
- External Communication
- Device authentication
- Access control of onboard resources
  - Flash updating
  - Secure storage
  - Peripherals
- Root of trust
- Reduction of attack surface

Some of these categories are interconnected and rely on multiple pieces to be in place to produce a full solution for the application.

9.1.3 Secure Development Process

The development of secure code shall adhere to certain criteria. These include coding guidelines and development processes that can be roughly separated into two categories related to software quality and related to software security. Furthermore, a system architecture document shall be created and kept up-to-date with future development.

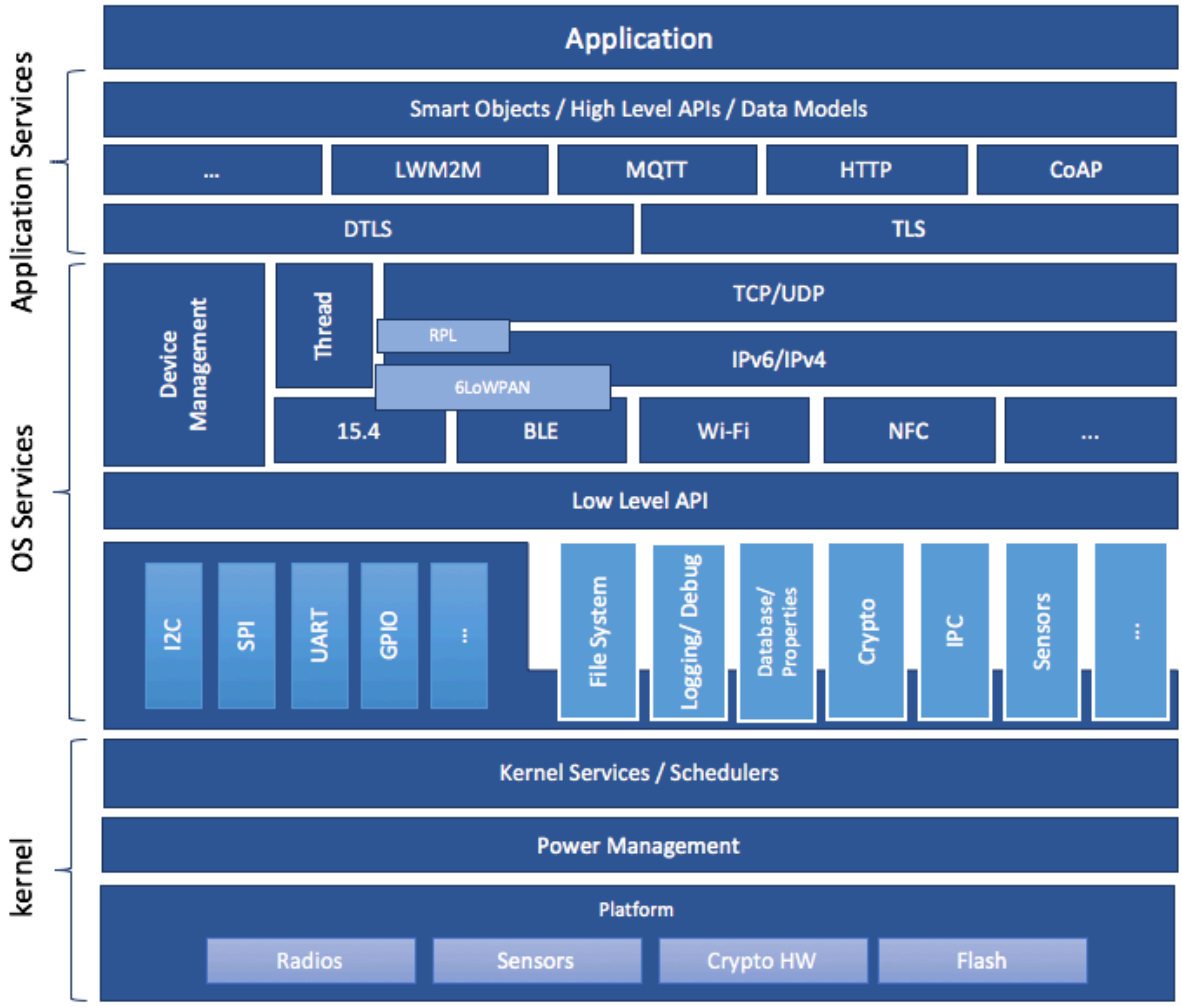


Fig. 2: Figure 2: Zephyr System Architecture

System Architecture

A high-level schematic of the Zephyr system architecture is given in Figure 2. It separates the architecture into an OS part (*kernel + OS Services*) and a user-specific part (*Application Services*). The OS part itself contains low-level, platform specific drivers and the generic implementation of I/O APIs, file systems, kernel-specific functions, and the cryptographic library.

A document describing the system architecture and design choices shall be created and kept up to date with future development. This document shall include the base architecture of the Zephyr OS and an overview of important submodules. For each of the modules, a dedicated architecture document shall be created and evaluated against the implementation. These documents shall serve as an entry point to new developers and as a basis for the security architecture. Please refer to the *Zephyr subsystem documentation* for detailed information.

Secure Coding

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. These standards are included in the Zephyr Project documentation, specifically in its *Secure Coding* section. In [**?**], the following, widely accepted principles for protection mechanisms are defined to prevent security violations and limit their impact:

- **Open design** as a design principle incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [**?**] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [**?**].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [**?**].
- **Least privilege** describes an access model in which each user, program and thread shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [**?**] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure its usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth:** do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [**?**] paradigm shall be used.
- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely

used (a threshold of 80% is given in [**?**]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.

- **Change management:** to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

Based on these design principles and commonly accepted best practices, a secure development guide shall be developed, published, and implemented into the Zephyr development process. Further details on this are given in the *Secure Design* section.

Quality Assurance

The quality assurance part encompasses the following criteria:

- **Adherence to the Coding Conventions** with respect to coding style, naming schemes of modules, functions, variables, and so forth. This increases the readability of the Zephyr code base and eases the code review. These coding conventions are enforced by automated scripts prior to check-in.
- **Adherence to Deployment Guidelines** is required to ensure consistent releases with a well-documented feature set and a trackable list of security issues.
- **Code Reviews** ensure the functional correctness of the code base and shall be performed on each proposed code change prior to check-in. Code reviews shall be performed by at least one independent reviewer other than the author(s) of the code change. These reviews shall be performed by the subsystem maintainers and developers on a functional level and are to be distinguished from security reviews as laid out in the *Secure Design* section. Refer to the *Project and Governance* documentation for more information.
- **Static Code Analysis** tools efficiently detect common coding mistakes in large code bases. All code shall be analyzed using an appropriate tool prior to merges into the main repository. This is not per individual commit, but is to be run on some interval on specific branches. It is mandatory to remove all findings or waive potential false-positives before each release. Waivers shall be documented centrally and in the form of a comment inside the source code itself. The documentation shall include the employed tool and its version, the date of the analysis, the branch and parent revision number, the reason for the waiver, the author of the respective code, and the approver(s) of the waiver. This shall as a minimum run on the main release branch and on the security branch. It shall be ensured that each release has zero issues with regard to static code analysis (including waivers). Refer to the *Project and Governance* documentation for more information.
- **Complexity Analyses** shall be performed as part of the development process and metrics such as cyclomatic complexity shall be evaluated. The main goal is to keep the code as simple as possible.
- **Automation:** the review process and checks for coding rule adherence are a mandatory part of the precommit checks. To ensure consistent application, they shall be automated as part of the precommit procedure. Prior to merging large pieces of code in from subsystems, in addition to review process and coding rule adherence, all static code analysis must have been run and issues resolved.

Release and Lifecycle Management

Lifecycle management contains several aspects:

- **Device management** encompasses the possibility to update the operating system and/or security related sub-systems of Zephyr enabled devices in the field.
- **Lifecycle management:** system stages shall be defined and documented along with the transactions between the stages in a system state diagram. For security reasons, this shall include locking of the device in case an attack has been detected, and a termination if the end of life is reached.
- **Release management** describes the process of defining the release cycle, documenting releases, and maintaining a record of known vulnerabilities and mitigations. Especially for certification purposes the integrity of the release needs to be ensured in a way that later manipulation (e.g., inserting of backdoors, etc.) can be easily detected.
- **Rights management and NDAs:** if required by the chosen certification, the confidentiality and integrity of the system needs to be ensured by an appropriate rights management (e.g., separate source code repository) and non-disclosure agreements between the relevant parties. In case of a repository shared between several parties, measures shall be taken that no malicious code is checked in.

These points shall be evaluated with respect to their impact on the development process employed for the Zephyr project.

9.1.4 Secure Design

In order to obtain a certifiable system or product, the security process needs to be clearly defined and its application needs to be monitored and driven. This process includes the development of security related modules in all of its stages and the management of reported security issues. Furthermore, threat models need to be created for currently known and future attack vectors, and their impact on the system needs to be investigated and mitigated. Please refer to the *Secure Coding* outlined in the Zephyr project documentation for detailed information.

The software security process includes:

- **Adherence to the Secure Development Coding** is mandatory to avoid that individual components breach the system security and to minimize the vulnerability of individual modules. While this can be partially achieved by automated tests, it is inevitable to investigate the correct implementation of security features such as countermeasures manually in security-critical modules.
- **Security Reviews** shall be performed by a security architect in preparation of each security-targeted release and each time a security-related module of the Zephyr project is changed. This process includes the validation of the effectiveness of implemented security measures, the adherence to the global security strategy and architecture, and the preparation of audits towards a security certification if required.
- **Security Issue Management** encompasses the evaluation of potential system vulnerabilities and their mitigation as described in *Security Issue Management*.

These criteria and tasks need to be integrated into the development process for secure software and shall be automated wherever possible. On system level, and for each security related module of the secure branch of Zephyr, a directly responsible security architect shall be defined to guide the secure development process.

Security Architecture

The general guidelines above shall be accompanied by an architectural security design on system- and module-level. The high level considerations include

- The identification of **security and compliance requirements**
- **Functional security** such as the use of cryptographic functions whenever applicable
- Design of **countermeasures** against known attack vectors

- Recording of security relevant **auditable events**
- Support for **Trusted Platform Modules (TPM)** and **Trusted Execution Environments (TEE)**
- Mechanisms to allow for **in-the-field updates** of devices using Zephyr
- Task scheduler and separation

The security architecture development is based on assets derived from the structural overview of the overall system architecture. Based on this, the individual steps include:

1. **Identification of assets** such as user data, authentication and encryption keys, key generation data (obtained from RNG), security relevant status information.
2. **Identification of threats** against the assets such as breaches of confidentiality, manipulation of user data, etc.
3. **Definition of requirements** regarding security and protection of the assets, e.g., countermeasures or memory protection schemes.

The security architecture shall be harmonized with the existing system architecture and implementation to determine potential deviations and mitigate existing weaknesses. Newly developed sub-modules that are integrated into the secure branch of the Zephyr project shall provide individual documents describing their security architecture. Additionally, their impact on the system level security shall be considered and documented.

Security Vulnerability Reporting

Please see *Security Vulnerability Reporting* for information on reporting security vulnerabilities.

Threat Modeling and Mitigation

The modeling of security threats against the Zephyr RTOS is required for the development of an accurate security architecture and for most certification schemes. The first step of this process is the definition of assets to be protected by the system. The next step then models how these assets are protected by the system and which threats against them are present. After a threat has been identified, a corresponding threat model is created. This model contains the asset and system vulnerabilities, as well as the description of the potential exploits of these vulnerabilities. Additionally, the impact on the asset, the module it resides in, and the overall system is to be estimated. This threat model is then considered in the module and system security architecture and appropriate countermeasures are defined to mitigate the threat or limit the impact of exploits.

In short, the threat modeling process can be separated into these steps (adapted from **[?]**):

1. Definition of assets
2. Application decomposition and creation of appropriate data flow diagrams (DFDs)
3. Threat identification and categorization using the **[?]** and **[?]** approaches
4. Determination of countermeasures and other mitigation approaches

This procedure shall be carried out during the design phase of modules and before major changes of the module or system architecture. Additionally, new models shall be created, or existing ones shall be updated whenever new vulnerabilities or exploits are discovered. During security reviews, the threat models and the mitigation techniques shall be evaluated by the responsible security architect.

From these threat models and mitigation techniques tests shall be derived that prove the effectiveness of the countermeasures. These tests shall be integrated into the continuous integration workflow to ensure that the security is not impaired by regressions.

Vulnerability Analyses

In order to find weak spots in the software implementation, vulnerability analyses (VA) shall be performed. Of special interest are investigations on cryptographic algorithms, critical OS tasks, and connectivity protocols.

On a pure software level, this encompasses

- **Penetration testing** of the RTOS on a particular hardware platform, which involves testing the respective Zephyr OS configuration and hardware as one system.
- **Side channel attacks** (timing invariance, power invariance, etc.) should be considered. For instance, ensuring **timing invariance** of the cryptographic algorithms and modules is required to reduce the attack surface. This applies to both the software implementations and when using cryptographic hardware.
- **Fuzzing tests** shall be performed on both exposed APIs and protocols.

The list given above serves primarily illustration purposes. For each module and for the complete Zephyr system (in general on a particular hardware platform), a suitable VA plan shall be created and executed. The findings of these analyses shall be considered in the security issue management process, and learnings shall be formulated as guidelines and incorporated into the secure coding guide.

If possible (as in case of fuzzing analyses), these tests shall be integrated into the continuous integration process.

9.1.5 Security Certification

One goal of creating a secure branch of the Zephyr RTOS is to create a certifiable system or certifiable submodules thereof. The certification scope and scheme are yet to be decided. However, many certifications such as Common Criteria [**?**] require evidence that the evaluation claims are indeed fulfilled, so a general certification process is outlined in the following. Based on the final choices for the certification scheme and evaluation level, this process needs to be refined.

Generic Certification Process

In general, the steps towards a certification or precertification (compare [**?**]) are:

1. The **definition of assets** to be protected within the Zephyr RTOS. Potential candidates are confidential information such as cryptographic keys, user data such as communication logs, and potentially IP of the vendor or manufacturer.
2. Developing a **threat model** and **security architecture** to protect the assets against exploits of vulnerabilities of the system. As a complete threat model includes the overall product including the hardware platform, this might be realized by a split model containing a precertified secure branch of Zephyr which the vendor could use to certify their Zephyr-enabled product.
3. Formulating an **evaluation target** that includes the **certification claims** on the security of the assets to be evaluated and certified, as well as assumptions on the operating conditions.
4. Providing **proof** that the claims are fulfilled. This includes consistent documentation of the security development process, etc.

These steps are partially covered in previous sections as well. In contrast to these sections, the certification process only requires to consider those components that shall be covered by the certification. The security architecture, for example, considers assets on system level and might include items not relevant for the certification.

Certification Options

For the security certification as such, the following options can be pursued:

1. **Abstract precertification of Zephyr as a pure software system:** this option requires assumptions on the underlying hardware platform and the final application running on top of Zephyr. If these assumptions are met by the hardware and the application, a full certification can be more easily achieved. This option is the most flexible approach but puts the largest burden on the product vendor.
2. **Certification of Zephyr on specific hardware platform without a specific application in mind:** this scenario describes the enablement of a secure platform running the Zephyr RTOS. The hardware manufacturer certifies the platform under defined assumptions on the application. If these are met, the final product can be certified with little effort.
3. **Certification of an actual product:** in this case, a full product including a specific hardware, the Zephyr RTOS, and an application is certified.

In all three cases, the certification scheme (e.g., FIPS 140-2 [**?**] or Common Criteria [**?**]), the scope of the certification (main-stream Zephyr, security branch, or certain modules), and the certification/assurance level need to be determined.

In case of partial certifications (options 1 and 2), assumptions on hardware and/or software are required for certifications. These can include [**?**]

- **Appropriate physical security** of the hardware platform and its environment.
- **Sufficient protection of storage and timing channels** on the hardware platform itself and all connected devices. (No mentioning of remote connections.)
- Only **trusted/assured applications** running on the device
- The device and its software stack is configured and operated by **properly trained and trusted individuals** with no malicious intent.

These assumptions shall be part of the security claim and evaluation target documents.

9.2 Security Vulnerability Reporting

9.2.1 Introduction

Vulnerabilities in the Zephyr project should preferably be reported by opening a draft security advisory on the Zephyr repository's security advisories page. Alternatively, reports may be submitted to the project security incident response team via email to vulnerabilities@zephyrproject.org. These reports will be acknowledged and analyzed by the security response team within 1 week.

All vulnerabilities are tracked and managed through the Zephyr project's security advisories page on GitHub, where the original submitter will be granted access to view the issues that they have reported.

9.2.2 Vulnerability Reporting Guidelines

Security research and vulnerability reporting are valued contributions to the Zephyr project. To keep this process safe and productive for everyone, all reports and related activities must comply with the Zephyr project Code of Conduct.

The guidelines below describe what is expected when researching and reporting security vulnerabilities in Zephyr.

Reporting expectations

To help the security response team triage and resolve issues efficiently, a report should:

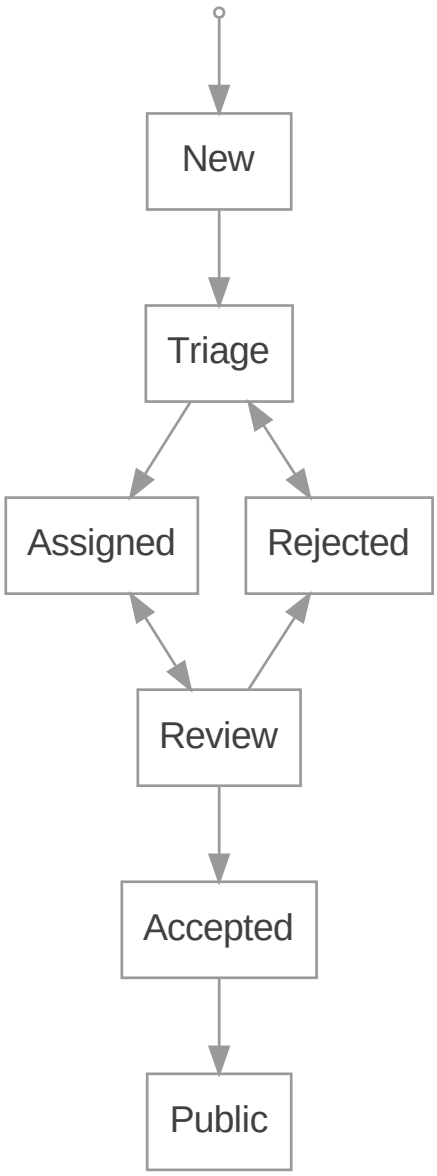
- Describe the affected component, version, board or platform, and configuration needed to reproduce the issue.

- Include clear, step-by-step reproduction instructions and a minimal proof-of-concept where possible.
- Explain the security impact (for example, loss of confidentiality, integrity, or availability) and any known mitigations.
- Avoid including destructive payloads or components that could cause harm to users or systems if the report were inadvertently disclosed.

Reporters are expected to make a good-faith effort to give the project a reasonable opportunity to investigate and remediate the issue before any public disclosure, consistent with the embargo and disclosure process described in the following sections.

9.2.3 Security Issue Management

Issues within this bug tracking system will transition through a number of states according to this diagram:



- New: This state represents new reports that have been entered directly by a reporter. When entered by the response team in response to an email, the issue shall be transitioned directly to Triage.
- Triage: This issue is awaiting Triage by the response team. The response team will analyze the issue, determine a responsible entity, assign it to that individual, and move the issue to the Assigned state. Part of triage will be to set the issue's priority.
- Assigned: The issue has been assigned, and is awaiting a fix by the assignee.
- Review: Once there is a Zephyr pull request for the issue, the PR link will be added to a comment in the issue, and the issue moved to the Review state.
- Accepted: Indicates that this issue has been merged into the appropriate branch within Zephyr.
- Public: The embargo period has ended. The issue will be made publicly visible, the associ-

ated CVE updated, and the vulnerabilities page in the docs updated to include the detailed information.

The security advisories created are kept private, due to the sensitive nature of security reports. The issues are only visible to certain parties:

- Members of the PSIRT mailing list
- the reporter
- others, as proposed and ratified by the Zephyr Security Subcommittee. In the general case, this will include:
  - The code owner responsible for the fix.
  - The Zephyr release owners for the relevant releases affected by this vulnerability.

The Zephyr Security Subcommittee shall review the reported vulnerabilities during any meeting with more than three people in attendance. During this review, they shall determine if new issues need to be embargoed.

The guideline for embargo will be based on: 1. Severity of the issue, and 2. Exploitability of the issue. Issues that the subcommittee decides do not need an embargo will be reproduced in the regular Zephyr project bug tracking system.

Security sensitive vulnerabilities shall be made public after an embargo period of at most 90 days. The intent is to allow 30 days within the Zephyr project to fix the issues, and 60 days for external parties building products using Zephyr to be able to apply and distribute these fixes.

Fixes to the code shall be made through pull requests PR in the Zephyr project github. Developers shall make an attempt to not reveal the sensitive nature of what is being fixed, and shall not refer to CVE numbers that have been assigned to the issue. The developer instead should merely describe what has been fixed.

The security subcommittee will maintain information mapping embargoed CVEs to these PRs (this information is within the Github security advisories), and produce regular reports of the state of security issues.

Each issue that is considered a security vulnerability shall be assigned a CVE number. As fixes are created, it may be necessary to allocate additional CVE numbers, or to retire numbers that were assigned.

9.2.4 Vulnerability Notification

Each Zephyr release shall contain a report of CVEs that were fixed in that release. Because of the sensitive nature of these vulnerabilities, the release shall merely include a list of CVEs that have been fixed. After the embargo period, the vulnerabilities page shall be updated to include additional details of these vulnerabilities. The vulnerability page shall give credit to the reporter(s) unless a reporter specifically requests anonymity.

The Zephyr project shall maintain a vulnerability-alerts mailing list. This list will be seeded initially with a contact from each project member. Additional parties can request to join this list by filling out the form at the Vulnerability Registry. These parties will be vetted by the project director to determine that they have a legitimate interest in knowing about security vulnerabilities during the embargo period.

Periodically, the security subcommittee will send information to this mailing list describing known embargoed issues, and their backport status within the project. This information is intended to allow them to determine if they need to backport these changes to any internal trees.

When issues have been triaged, this list will be informed of:

- The Zephyr Project security advisory link (GitHub).
- The CVE number assigned.
- The subsystem involved.

- The severity of the issue.

After acceptance of a PR fixing the issue (merged), in addition to the above, the list will be informed of:

- The association between the CVE number and the PR fixing it.
- Backport plans within the Zephyr project.

9.2.5 Backporting of Security Vulnerabilities

Each security issue fixed within zephyr shall be backported to the following releases:

- The current Long Term Stable (LTS) release.
- The most recent two releases.

The developer of the fix shall be responsible for any necessary backports, and apply them to any of the above listed release branches, unless the fix does not apply (the vulnerability was introduced after this release was made). All recommendations for *vulnerability fixes* apply for backport pull requests (and associated issues). Additionally, it is recommended that the developer privately informs the responsible release manager that the backport pull request and issue are addressing a vulnerability.

Backports will be tracked on the security advisory.

9.2.6 Need to Know

Due to the sensitive nature of security vulnerabilities, it is important to share details and fixes only with those parties that have a need to know. The following parties will need to know details about security vulnerabilities before the embargo period ends:

- Maintainers will have access to all information within their domain area only.
- The current release manager, and the release manager for historical releases affected by the vulnerability (see backporting above).
- The Project Security Incident Response (PSIRT) team will have full access to information. The PSIRT is made up of representatives from platinum members, and volunteers who do work on triage from other members.
- As needed, release managers and maintainers may be invited to attend additional security meetings to discuss vulnerabilities.

9.3 Secure Coding

Traditionally, microcontroller-based systems have not placed much emphasis on security. They have usually been thought of as isolated, disconnected from the world, and not very vulnerable, just because of the difficulty in accessing them. The Internet of Things has changed this. Now, code running on small microcontrollers often has access to the internet, or at least to other devices (that may themselves have vulnerabilities). Given the volume they are often deployed at, uncontrolled access can be devastating[1].

This document describes the requirements and process for ensuring security is addressed within the Zephyr project. All code submitted should comply with these principles.

Much of this document comes from [**?**].

[1] An attack resulted in a significant portion of DNS infrastructure being taken down.

9.3.1 Introduction and Scope

This document covers guidelines for the Zephyr Project, from a security perspective. Many of the ideas contained herein are captured from other open source efforts.

We begin with an overview of secure design as it relates to Zephyr. This is followed by a section on *Secure development knowledge*, which gives basic requirements that a developer working on the project will need to have. This section gives references to other security documents, and full details of how to write secure software are beyond the scope of this document. This section also describes vulnerability knowledge that at least one of the primary developers should have. This knowledge will be necessary for the review process described below this.

Following this is a description of the review process used to incorporate changes into the Zephyr codebase. This is followed by documentation about how security-sensitive issues are handled by the project.

Finally, the document covers how changes are to be made to this document.

9.3.2 Secure Coding

Designing an open software system such as Zephyr to be secure requires adhering to a defined set of design standards. In [**?**], the following, widely accepted principles for protection mechanisms are defined to help prevent security violations and limit their impact:

- **Open design** as a design guideline incorporates the maxim that protection mechanisms cannot be kept secret on any system in widespread use. Instead of relying on secret, custom-tailored security measures, publicly accepted cryptographic algorithms and well established cryptographic libraries shall be used.
- **Economy of mechanism** specifies that the underlying design of a system shall be kept as simple and small as possible. In the context of the Zephyr project, this can be realized, e.g., by modular code [**?**] and abstracted APIs.
- **Complete mediation** requires that each access to every object and process needs to be authenticated first. Mechanisms to store access conditions shall be avoided if possible.
- **Fail-safe defaults** defines that access is restricted by default and permitted only in specific conditions defined by the system protection scheme, e.g., after successful authentication. Furthermore, default settings for services shall be chosen in a way to provide maximum security. This corresponds to the "Secure by Default" paradigm [**?**].
- **Separation of privilege** is the principle that two conditions or more need to be satisfied before access is granted. In the context of the Zephyr project, this could encompass split keys [**?**].
- **Least privilege** describes an access model in which each user, program, and thread, shall have the smallest possible subset of permissions in the system required to perform their task. This positive security model aims to minimize the attack surface of the system.
- **Least common mechanism** specifies that mechanisms common to more than one user or process shall not be shared if not strictly required. The example given in [**?**] is a function that should be implemented as a shared library executed by each user and not as a supervisor procedure shared by all users.
- **Psychological acceptability** requires that security features are easy to use by the developers in order to ensure their usage and the correctness of its application.

In addition to these general principles, the following points are specific to the development of a secure RTOS:

- **Complementary Security/Defense in Depth**: do not rely on a single threat mitigation approach. In case of the complementary security approach, parts of the threat mitigation are performed by the underlying platform. In case such mechanisms are not provided by the platform, or are not trusted, a defense in depth [**?**] paradigm shall be used.

- **Less commonly used services off by default**: to reduce the exposure of the system to potential attacks, features or services shall not be enabled by default if they are only rarely used (a threshold of 80% is given in [**?**]). For the Zephyr project, this can be realized using the configuration management. Each functionality and module shall be represented as a configuration option and needs to be explicitly enabled. Then, all features, protocols, and drivers not required for a particular use case can be disabled. The user shall be notified if low-level options and APIs are enabled but not used by the application.
- **Change management**: to guarantee a traceability of changes to the system, each change shall follow a specified process including a change request, impact analysis, ratification, implementation, and validation phase. In each stage, appropriate documentation shall be provided. All commits shall be related to a bug report or change request in the issue tracker. Commits without a valid reference shall be denied.

9.3.3 Secure development knowledge

Secure designer

The Zephyr project must have at least one primary developer who knows how to design secure software.

This requires understanding the following design principles, including the 8 principles from [**?**]:

- economy of mechanism (keep the design as simple and small as practical, e.g., by adopting sweeping simplifications)
- fail-safe defaults (access decisions shall deny by default, and projects' installation shall be secure by default)
- complete mediation (every access that might be limited must be checked for authority and be non-bypassable)
- open design (security mechanisms should not depend on attacker ignorance of its design, but instead on more easily protected and changed information like keys and passwords)
- separation of privilege (ideally, access to important objects should depend on more than one condition, so that defeating one protection system won't enable complete access. For example, multi-factor authentication, such as requiring both a password and a hardware token, is stronger than single-factor authentication)
- least privilege (processes should operate with the least privilege necessary)
- least common mechanism (the design should minimize the mechanisms common to more than one user and depended on by all users, e.g., directories for temporary files)
- psychological acceptability (the human interface must be designed for ease of use - designing for "least astonishment" can help)
- limited attack surface (the set of the different points where an attacker can try to enter or extract data)
- input validation with allowlists (inputs should typically be checked to determine if they are valid before they are accepted; this validation should use allowlists (which only accept known-good values), not blocklists (which attempt to list known-bad values)).

Vulnerability Knowledge

A "primary developer" in a project is anyone who is familiar with the project's code base, is comfortable making changes to it, and is acknowledged as such by most other participants in the project. A primary developer would typically make a number of contributions over the past year (via code, documentation, or answering questions). Developers would typically be considered primary developers if they initiated the project (and have not left the project more than three years ago), have the option of receiving information on a private vulnerability reporting channel

(if there is one), can accept commits on behalf of the project, or perform final releases of the project software. If there is only one developer, that individual is the primary developer.

At least one of the primary developers **must** know of common kinds of errors that lead to vulnerabilities in this kind of software, as well as at least one method to counter or mitigate each of them.

Examples (depending on the type of software) include SQL injection, OS injection, classic buffer overflow, cross-site scripting, missing authentication, and missing authorization. See the CWE/SANS top 25 or OWASP Top 10 for commonly used lists.

A free class from the nonprofit OpenSecurityTraining2 for C/C++ developers is available at OST2_1001. It teaches how to prevent, detect, and mitigate linear stack/heap buffer overflows, non-linear out of bound writes, integer overflows, and other integer issues. The follow-on class, OST2_1002, covers uninitialized data access, race conditions, use-after-free, type confusion, and information disclosure vulnerabilities.

Zephyr Security Subcommittee

There shall be a "Zephyr Security Subcommittee", responsible for enforcing this guideline, monitoring reviews, and improving these guidelines.

This team will be established according to the Zephyr Project charter.

9.3.4 Code Review

The Zephyr project shall use a code review system that all changes are required to go through. Each change shall be reviewed by at least one primary developer that is not the author of the change. This developer shall determine if this change affects the security of the system (based on their general understanding of security), and if so, shall request the developer with vulnerability knowledge, or the secure designer to also review the code. Any of these individuals shall have the ability to block the change from being merged into the mainline code until the security issues have been addressed.

9.3.5 Issues and Bug Tracking

The Zephyr project shall have an issue tracking system (such as GitHub) that can be used to record and track defects that are found in the system.

Because security issues are often sensitive, this issue tracking system shall have a field to indicate a security issue. Setting this field shall result in the issue only being visible to the Zephyr Security Subcommittee. In addition, there shall be a field to allow the Zephyr Security Subcommittee to add additional users that will have visibility to a given issue.

This embargo, or limited visibility, shall only be for a fixed duration, with a default being a project-decided value. However, because security considerations are often external to the Zephyr project itself, it may be necessary to increase this embargo time. The time necessary shall be clearly annotated in the issue itself.

The list of issues shall be reviewed at least once a month by the Zephyr Security Subcommittee. This review should focus on tracking the fixes, determining if any external parties need to be notified or involved, and determining when to lift the embargo on the issue. The embargo should **not** be lifted via an automated means, but the review team should avoid unnecessary delay in lifting issues that have been resolved.

9.3.6 Modifications to This Document

Changes to this document shall be reviewed by the Zephyr Security Subcommittee, and approved by consensus.

9.4 Sensor Device Threat Model

This document describes a threat model for an IoT sensor device. Spelling out a threat model helps direct development effort, and can be used to help prioritize these efforts as well.

This device contains a sensor of some type (for example temperature, or a pressure in a pipe), which sends this data to an SoC running a microcontroller. This microcontroller connects to a cloud service, and relays this sensor data to this service. The cloud service is also able to send configuration data to the device, as well as software update images. A general diagram can be seen in Figure 1:

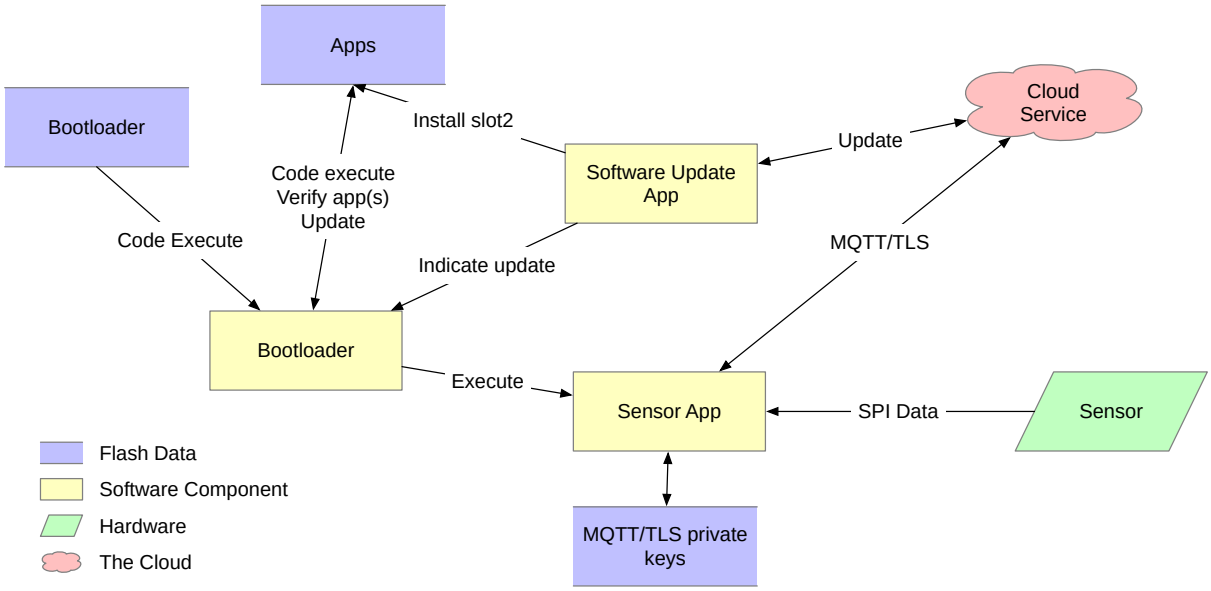


Fig. 3: Figure 1. Sensor General Diagram

In this sensor device, the sensor connects with the SoC via an SPI bus, and the SoC has a network interface that it uses to communicate with the cloud service. The particulars of these interfaces can impact the threat model in unexpected ways, and variants on this will need to be considered (for example, using a separate network interface SoC connected via some type of bus).

This model also focuses on communicating via the MQTT-over-TLS protocol, as this seems to be in wide use[1].

9.4.1 Assets

One aspect of the threat model to consider are assets involved in the operation of the device. The following list enumerates the assets included in this model:

1. **The bootloader**. This is a small code/data image contained in on-device flash that is the first code to run. In order to establish a root of trust, this image must be immutable. This model assumes that the SoC provides a mechanism to protect a region of the flash from future writes, and that this will be done after this image is programmed into the device, early in production [th-imboot].
2. **The application firmware image**. This asset consists of the remainder of the firmware run by the microcontroller. The distinction is made because this part of the image will need to be updated periodically as security vulnerabilities are discovered. Requirements for updates to this image are:
   a. The image shall only be replaced with an authorized image [th-authrepl].

[1] See https://www.slideshare.net/kartben/iot-developer-survey-2018. As of this writing, the three major cloud IoT service providers, AWS IoT, Google Cloud IoT, and Microsoft Azure IoT all provide MQTT over TLS. Some feedback has suggested that some find difficulty with UDP protocols and routing issues on various networks.

b. When an authorized replacement image is available, the update shall be done in a timely manner [th-timely-update].

c. The image update shall be seen as atomic, meaning that when the image is run, the flash shall contain either the update image in its entirety, or the old image in its entirety [th-atomic-update].

3. **Root certificate list**. In order to authenticate the cloud service (server), the IoT device must have a list of root certificates that are allowed to sign the certificate on the server. For cloud-provider based services, this list will generally be provided by the service provider. Because the root certificates can expire, and possibly be revoked, this list will need to be periodically updated [th-root-certs], [th-root-check].

4. **Client secrets**. To authenticate the client to the service, the client must possess some kind of secret. This is generally a private key, usually either an RSA key or an EC private key. When establishing communication with the server, the device will use this secret either as part of the TLS establishment, or to sign a message used in the communication.

   This secret is generally generated by the service provider, or by software running elsewhere, and must be securely installed on the device. Policy may dictate that this secret be replaced periodically, which will require a way to update the client secret. Typically, the service will allow two or three active keys to allow this update to proceed while the old key is used.

   These secrets must be protected from read, and the smallest amount of code necessary shall have access to them. [th-secret-storage]

5. **Current date/time**. TLS certificate verification requires knowledge of the current date and time in order to determine if the current time falls within the certificate's current validity time. Also, token based client authentication will generally require the client to sign a message containing a time window that the token is valid. Certificate validation requires the device's notion of date and time to be accurate within a day or so. Token generation generally requires the time to be accurate within 5-10 minutes.

   It may be possible to approximate secure time by querying an external time server. Secure NTP is possibly beyond the capabilities of an IoT device. The main risks of having incorrect time are denial of service (the device rejects valid certificates), and the generation of tokens with invalid times. It could be possible to trick the device into generating tokens that are valid in the future, but the attacker would also have to spoof the server's certificate to be able to intercept this. [th-time]

6. **Sensor data**. The data received from the sensor itself, and delivered to the service shall be delivered without modification or tampering.

7. **Device configuration**. Various configuration data, such as the hostname of the service to connect to, the address of a time server, frequency and parameters of when sensor data is sent to the service, and other need to be kept by the device. This configuration data will need to be updated periodically as the configuration changes. Updates should be allowed only from authorized parties. [th-conf]

8. **Logs**. In order to assist with analysis of security issues, the device shall log information about security-pertinent events. IoT devices generally have limited storage, and as such, these logs need to be carefully selected. It may also be possible to send these log events to the cloud service where they can be stored in a more resource-available environment. Types of events that should be logged include:

   a. **Firmware image updates**. The system should log the download of new images, and when an image is successfully updated.

   b. **Client secret changes**. Changes and new client secrets should be logged.

   c. **Changes to the device configuration**.

   [th-logs]

9.4.2 Communication

In addition to assets, the threat model also considers the locations where data or assets are communicated between entities of the system.

1. **Flash contents**. The flash device contains several regions. The contents of flash can be modified programmatically by the SoC's CPU.

   a. **The bootloader**. As described in the Assets section, the bootloader is a small section of the flash device containing the code initially run. This section shall be written early in the lifecycle of the device, and the flash device then configured to permanently disallow modification of this section. This configuration should also prevent modification via external interfaces, such as JTAG or SWD debuggers.

      The bootloader is responsible for verifying the signature of the application image as well as updating the application image from the update image when an update is needed.

      The bootloader shall verify the signature of the update image before installing it.

      The bootloader shall only accept an update image with a newer version number than the current image.

   b. **The application image**. The application image contains the code executed during normal operation of the device. Before running this image, the bootloader shall verify a digital signature of the image, to avoid running an image that has been tampered with. The flash/system shall be configured such that after the bootloader has completed, the CPU will be unable to write to the application image.

   c. **The update image**. This is an area of flash that holds a new version of the application image. This image will be downloaded and stored by the application during normal operation. When this has completed, the application can trigger a reboot, and the bootloader can install the new image.

   d. **Secret storage**. An area of the flash will be used to store client secrets. This area is written and read by a subset of the application image. The application shall be configured to protect this area from both reads and writes by code that does not need to have access to it, giving consideration to possible exploits found within a majority of the application code. Revealing the contents of the secrets would allow the attacker to spoof this device.

      Initial secrets shall be placed in the device during a provisioning activity, distinct from normal operation of the device. Later updates can be made under the direction of communication received over a secured channel to the service.

   e. **Configuration storage**. There shall be an area to store other configuration information. On resource-constrained devices, it is allowed for this to be stored in the same region as the secret storage, however, this adds additional code that has access to the secret storage area, and as such, more code that must be scrutinized.

   f. **Log storage**. The device may have an area of flash where log events can be written.

2. **Sensor/Actuator interface**. In this design, the sensor or actuator communicates with the SoC via a bus, such as SPI. The hardware design shall be made to make intercepting this bus difficult for an attack. Required techniques depend on the sensitivity and use of the sensor data, and can range from having the sensor mounted on the same PCB as the MCU to epoxy potting the entire device.

3. **Communication with cloud service**. Communication between the device, and the cloud service will be done over the general internet. As such, it shall be assumed that an attacker can arbitrarily intercept this channel and, for example, return spoofed DNS results or attempt man-in-the-middle attacks against communication with cloud services.

   The device shall use TLS for all communication with the cloud service [th-all-tls]. The TLS

stack shall be configured to use only cipher suites that are generally considered secure[2], including forward secrecy. The communication shall be secured by the following:

a. **Cipher suite selection**. The device shall only allow communication with generally agreed secure cipher suites [th-tls-ciphers].

b. **Server certificate verification**. The server presented by the server shall be verified [th-root-check].

   i. **Naming**. The certificate shall name the host and service the cloud service server is providing. **RFC 6125** describes best practices for this. It is permissible for the device to require the certificate to be more restrictive than as described in this RFC, provided the service can use a certificate that can comply.

   ii. **Path validation**. The device shall verify that the certificate chain has a valid signature path from a root certificate contained within the device, to the certificate presented by the service. **RFC 4158** describes this in general. The device is permitted to require a more restricted path, provided the server certificate used complies with this restriction.

   iii. **Validity period**. The validity period of all presented certificates shall be checked against the device's best notion of the current time.

c. **Client authentication**. The client shall authenticate itself to the service using a secret known only to that particular device. There are several options, and the technique used is generally mandated by the particular service provider being used [th-tls-client-auth].

   i. **TLS client certificates**. The TLS protocol allows the client to present a certificate, and assert its knowledge of the secret described by that certificate. Generally, these certificates will be stored within the service provider. These certificates can be self-signed, or signed by a CA. Since the service provider maintains a list of valid certificates (mapping them to a device identity), having these certificates signed by a CA does not add any additional security, but may be useful in the management of these certificates.

   ii. **Token-based authentication**. It is also possible for the client to authenticate itself using the *password* field of the MQTT CONNECT packet. However, the secret itself must not be transmitted in this packet. Instead, a token-based protocol, such as **RFC 7519**'s JSON Web Token (JWT) can be used. These tokens will generally have a small validity period (e.g. 1 hour), to prevent them from being reused if they are intercepted. The token shall not be sent until the device has verified the identity of the server.

d. **Random/Entropy source**. Cryptographic communication requires the generation of secure pseudorandom numbers. The device shall use a modern, accepted cryptographic random-bit generator to generate these random numbers. It shall use either a Non-Deterministic Random Bit Generator (True RBG) implemented in hardware within the SoC, or a Deterministic Random Bit Generator (Pseudo RBG) seeded by an entropy source within the SoC. Please see NIST SP 800-90A for information on approved RBGs and NIST SP 800-90B for information on testing a device's entropy source [th-entropy].

4. **Communication with the time service**. Ideally, the device shall contain hardware that maintains a secure time. However, most SoCs in use do not have support for this, and it will be necessary to consult an external time service. **RFC 4330** and referenced RFCs describe the Simple Network Time Protocol that can be used to query the current time from a network time server.

5. **Device lifecycle**. An IoT device will have a lifecycle from production to destruction and

[2] As new exploits are discovered, what is considered secure can change. Organizations such as https://www.ssllabs.com/ provide information on current ideas of how TLS must be configured to be secure.

disposal of the device. Aspects of this lifecycle that impact security include initial provisioning, normal operation, re-provisioning, and destruction.

a. **Initial provisioning**. During the initial provisioning stage, it is necessary to program the bootloader, an initial application image, a device secret, and initial configuration data [th-initial-provision]. In addition, the bootloader flash protection shall be installed. Of this information, only the device secret needs to differ per device. This secret shall be securely maintained, and destroyed in all locations outside of the device once it has been programmed [th-initial-secret].

b. **Normal operation**. Normal operation includes the behavior described by the rest of this document.

c. **Re-provisioning**. Sometimes it is necessary to re-provision a device, such as for a different application. One way to do this is to keep the same device secret, and replace the configuration data, as well as the cloud service data associated with the device. It is also possible to program a new device secret, but if this is done it shall be done securely, and the new secret destroyed externally once programmed into the device [th-reprovision].

d. **Destruction**. To prevent the device secret from being used to spoof the device, upon decommissioning, the secret for a particular device shall be rendered ineffective [th-destruction]. Possibilities include:

   i. Hardware destruction of the device.

   ii. Securely wiping the flash area containing the secret[3].

   iii. Removing the device identity and certificate from the service.

9.4.3 Other Considerations

In addition to the above, network connected devices generally will need a way to configure them to connect to the network environment they are placed in. There are numerous ways of doing this, and it is important for these configuration methods to not circumvent the security requirements described above.

9.4.4 Threats

9.4.5 Notes

9.5 Hardening Tool

Before launching a product, it's crucial to ensure that your software is as secure as possible. This process, known as "hardening", involves strengthening the security of a system to protect it from potential threats and vulnerabilities.

At a high-level, hardening a Zephyr application can be seen as a two-fold process:

1. Disabling features and compilation flags that might lead to security vulnerabilities (ex. making sure that no "experimental" features are being used, disabling features typically used for debugging purposes such as assertions, shell, etc.).
2. Enabling optional features that can lead to improve security (ex. stack sentinel, hardware stack protection, etc.). Some of these features might be hardware-dependent.

To simplify this process, Zephyr offers a **hardening tool** designed to analyze an application's configuration against the **hardening database**, a set of recommendations curated by the Zephyr **Security Working Group**. The tool looks at the Kconfig options in the build target and provides tailored suggestions and recommendations to adjust security-related options, along with the rationale behind each recommendation.

[3] Note that merely erasing this flash area is unlikely to be sufficient.

9.5.1 Usage

Using west:

```
# From the root of the zephyr repository
west build -b reel_board samples/hello_world
west build -t hardenconfig
```

Using CMake and ninja:

```
# From the root of the zephyr repository
# Use cmake to configure a Ninja-based buildsystem:
cmake -Bbuild -GNinja -DBOARD=reel_board samples/hello_world

# Now run the build tool on the generated build system:
ninja -Cbuild hardenconfig
```

The output should be similar to the table below. For each configuration option set to a value that could lead to a security vulnerability, the table will propose a recommended value that should be used instead, together with the reason the option matters.

```
Hardening report for profile: strict
+-----------------------------+-----------+---------------+----------------+--------------------
| Name                        | Current   | Recommended   | Check result   |␣
↪Rationale                                              |
+=============================+===========+===============+================+====================
| CONFIG_BUILD_OUTPUT_STRIPPED | n         | y             | FAIL           |␣
↪Produces a stripped copy of the ELF next to the  |
|                             |           |               |                |␣
↪unstripped one, so a product that distributes or |
|                             |           |               |                |␣
↪flashes the ELF itself ships no symbol names or  |
|                             |           |               |                |␣
↪debug information for an attacker to work from.  |
|                             |           |               |                | Both␣
↪files are built; deploying the stripped one |
|                             |           |               |                |␣
↪remains the user's responsibility.               |
+-----------------------------+-----------+---------------+----------------+--------------------
| CONFIG_STACK_SENTINEL       | n         | y             | FAIL           |␣
↪Places a software sentinel value at the end of   |
|                             |           |               |                | each␣
↪thread stack and checks it at context       |
|                             |           |               |                |␣
↪switches, catching overflows on hardware without |
|                             |           |               |                |␣
↪MPU/MMU stack protection.                        |
+-----------------------------+-----------+---------------+----------------+--------------------
```

Options that are not applicable to the current target (for example, MPU-based protections on hardware without an MPU, or options with no visible prompt in the current configuration) are not reported.

In addition to the database-driven checks, the tool flags any enabled option that is marked in Kconfig itself as experimental, deprecated or not secure (i.e. options selecting `CONFIG_EXPERIMENTAL`, `CONFIG_DEPRECATED` or `CONFIG_NOT_SECURE`).

9.5.2 Profiles

Recommendations are grouped into **profiles**. Two profiles are provided in-tree:

`base`
: Baseline hardening every production build should satisfy: enable available memory-protection and exploit-mitigation features, and disable inherently insecure options.

`strict` **(default)**
: Everything in base, plus removal of debugging, tracing and observability features (logging, shell, assertions, etc.) that enlarge the attack surface or disclose internal state. Some products legitimately keep a subset of these enabled — if that is your case, check against the base profile instead:

```
west build -t hardenconfig -- -DHARDENCONFIG_PROFILE=base
```

9.5.3 Configuration options

The tool is controlled through CMake cache variables (passed after `--` on the `west build` command line, as above) or environment variables of the same name:

Option	Effect
`HARDENCONFIG_PROFILE`	Hardening profile to check against. Default: `strict`.
`HARDENCONFIG_SHOW_ALL`	When set, also list passing and non-applicable options instead of only failures.
`HARDENCONFIG_STRICT`	When set, exit with a non-zero code if any check fails. Useful to gate CI pipelines or release processes on a clean hardening report.
`HARDENCONFIG_JSON`	Path to a file where results are additionally written as JSON, for consumption by scripts and dashboards.
`HARDENCONFIG_EXTRA_SOURCE`	Semicolon-separated list of additional hardening database files (see below).

9.5.4 The hardening database

The database is distributed so that each rule lives with the code it hardens and is reviewed by that code's maintainers:

- scripts/kconfig/hardening.yaml — the central file, owned by the Security Working Group. It defines the hardening **profiles**, plus rules for symbols defined in the top-level Kconfig files.
- `hardening.yaml` **fragments** next to the Kconfig files they relate to (for example subsys/bluetooth/hardening.yaml or arch/x86/hardening.yaml), containing rules only. Fragments are discovered automatically under `arch/`, `boards/`, `drivers/`, `kernel/`, `lib/`, `modules/`, `share/`, `soc/` and `subsys/`. Defining the same symbol in two files is an error, as is defining profiles in a fragment.

All files use the same JSON schema, scripts/schemas/hardening-schema.yaml. Each rule is keyed by the Kconfig symbol it applies to and recommends either an exact value or an integer constraint:

```yaml
rules:
  BOOT_BANNER:
    value: n
    rationale: |
      The boot banner prints the exact Zephyr version to the console,
      letting anyone with console access fingerprint the firmware and
      match it against known vulnerabilities for that release.
    references: [CWE-200]

  STACK_POINTER_RANDOM:
```

(continues on next page)

(continued from previous page)

```
    min: 100
    rationale: |
      Randomizing each thread's initial stack pointer makes stack
      addresses unpredictable. The offset is taken out of the thread's
      own stack, which keeps the recommended value small.
    references: [CWE-121]
```

Contributing a new rule means adding it to the `hardening.yaml` fragment next to the relevant Kconfig file (creating the fragment if the subsystem does not have one yet) and requires a `rationale` — that requirement is enforced by the schema. Continuous integration additionally verifies that every file parses and matches the schema, that every rule references an existing Kconfig symbol with a value coherent with the symbol's type, that no rule is defined twice, and that no file looks like a misnamed fragment — so entries cannot silently go stale or silently fail to load.

The `rationale` is what the report shows the reader, wrapped into a narrow column, and is usually all they have to go on. State the security consequence of the option in one or two sentences, present tense, with the feature as the subject:

`value: n`
: The exposure enabling it creates and, where it matters, who can reach it: "Verbose modem logging dumps all data exchanged with the modem, which can include credentials such as APN settings, PINs and server addresses."

`value: y`
: The attack it defeats, or what leaving it off permits, whichever reads better.

`value: 0`, `min` and `max`
: What the other values emit or fail to prevent, and what the boundary value itself does.

Do not:

- Say where the option belongs. The profile is what decides whether a product should accept the consequence; write "…discloses internal transfer state and addresses", not "…and belongs only in development builds".
- Claim an effect that is not complete without further action. For example, an option may enable the generation of a stripped binary, but making sure that is the one deployed is up to the user.
- Restate a column the report already shows. In "0 disables randomization entirely; at least 100 bytes of random offset is recommended", the second half is the Recommended column.
- Repeat another rule's wording. Two rules that can fail in the same report should read differently enough for the reader to tell them apart.

Do not add an `n` rule for an option that can only be enabled while another `n` rule covering at least the same profiles is already violated; for example, a rule for a shell command module is redundant once `CONFIG_SHELL` itself is flagged, and CI rejects it. A rule for an option that merely `selects` a flagged one is not redundant: it names what the user must actually change.

Rules do not carry applicability conditions: whether a recommendation applies to a given target is already encoded in Kconfig itself (dependencies, hardware support), and the tool reports options that are not applicable to the current target as not applicable rather than as failures.

9.5.5 Extending the database out of tree

Product teams can layer their own recommendations on top of the in-tree database with `HARDENCONFIG_EXTRA_SOURCES`. Each additional file uses the same schema and may define new profiles (optionally extending in-tree ones) and new rules, as well as override in-tree rules. Redefining an existing profile is rejected, so an in-tree profile keeps the meaning documented here whatever a product layers on top of it:

```yaml
profiles:
  acme-production:
    extends: strict
    description: ACME product security policy.

rules:
  MY_VENDOR_DEBUG_INTERFACE:
    value: n
    profiles: [acme-production]
    rationale: |
      The vendor debug interface bypasses the product's authentication.
```

```
west build -t hardenconfig -- \
  -DHARDENCONFIG_EXTRA_SOURCES=/path/to/acme-hardening.yaml \
  -DHARDENCONFIG_PROFILE=acme-production
```

9.6 Control Flow Integrity

Control Flow Integrity (CFI) is a security feature that ensures that the control flow of a program follows a predefined path, preventing attackers from diverting execution to malicious code. CFI is particularly useful in defending against control flow hijacking attacks, such as return-oriented programming (ROP) and jump-oriented programming (JOP).

CFI works by validating the control flow of a program at runtime, ensuring that function calls and returns are made to legitimate targets. This is typically achieved through instrumentation of the code, which adds checks to verify that the control flow adheres to the expected paths defined by the program's control flow graph.

9.6.1 Forward and return edges

In the context of CFI, control flow edges can be categorized into two types:

1. **Forward edges**: These represent the flow of control from one function to another, such as a function call. Forward edges are typically validated to ensure that the target of a call is a legitimate function within the program.
2. **Return edges**: These represent the return from a function back to the caller. Return edges are validated to ensure that the return address is legitimate and corresponds to a valid call site.

Forward edges can be supported by compiler instrumentation, which adds checks to verify that the target of a function call is valid. Return edges can be supported by maintaining a shadow stack or using other mechanisms to ensure that return addresses are legitimate.

Zephyr does support maintaining a shadow stack which can be enabled via `CONFIG_HW_SHADOW_STACK`. Then, the kernel will use macros like `K_THREAD_HW_SHADOW_STACK_DEFINE`, in tandem with other thread stack related macros, to provide the area for shadow stacks used by threads. Usually, applications only need to enable `CONFIG_HW_SHADOW_STACK` (and related ones, such as `CONFIG_HW_SHADOW_STACK_PERCENTAGE_SIZE` and `CONFIG_HW_SHADOW_STACK_MIN_SIZE`) to enable shadow stack support. The kernel will then automatically manage the shadow stack for each thread.

Implementation details

The `K_THREAD_HW_SHADOW_STACK*` family of macros does a minimal setup of the shadow stack parameters. Then, they invoke arch-specific `ARCH_THREAD_HW_SHADOW_STACK*` macros to perform the actual setup.

9.6.2 Hardware support

While CFI can be implemented in software, hardware support can significantly enhance its effectiveness and performance. Currently, Zephyr supports Intel Control-flow Enforcement Technology (CET), which provides hardware-based support for CFI.

Intel CET

Intel Control-flow Enforcement Technology (CET) is a set of hardware features that enhance the security of applications by providing support for CFI. CET includes two main components:

1. **Shadow Stack**: This feature maintains a separate stack for return addresses, ensuring that return addresses cannot be tampered with. When a function returns, the return address is popped from the shadow stack and compared to that of the regular stack, providing an additional layer of protection against control flow hijacking.
2. **Indirect Branch Tracking (IBT)**: This feature tracks indirect branches (such as function pointers) and ensures that they only target valid locations in the code. It prevents attackers from redirecting execution to arbitrary code through techniques like ROP.

These two features proved the return and forward edge validation, respectively. To enable shadow stack support in Zephyr, on supported hardware, you can use the `CONFIG_HW_SHADOW_STACK` Kconfig option. To enable IBT, use `CONFIG_X86_CET_IBT`.

As IBT is effectively implemented by the compiler, toolchain support is necessary. Currently, Zephyr SDK x86 toolchain can be used to build applications with IBT support. However, their precompiled artifacts, such as `libc` and `libgcc`, do not have IBT enabled. Therefore, for `libc`, one needs to build it as a module, for example, by using `CONFIG_PICOLIBC_USE_MODULE`. For other bits, one would need a custom toolchain.

Intel CET Limitations Currently, the shadow stacks that are created behind the scenes live on the global namespace. Thus, one *cannot* reuse thread stack names, even across different compilation units.

ARM Pointer Authentication and Branch Target Identification

ARM platforms provide hardware-based CFI features through Pointer Authentication (PAC) and Branch Target Identification (BTI), available on both Cortex-M and ARM64 (Cortex-A/R) architectures:

1. **Pointer Authentication (PAC)**: Signs return addresses and other pointers using cryptographic signatures. When a return address is pushed to the stack, it is signed with a key unique to each thread. Before returning, the signature is verified, ensuring the return address has not been tampered with. This protects against Return-Oriented Programming (ROP) attacks. Available in ARMv8.1-M Mainline (Cortex-M) and ARMv8.3-A and later (ARM64).
2. **Branch Target Identification (BTI)**: Marks legitimate indirect branch targets with special BTI landing pad instructions. The processor verifies that indirect branches land only on valid targets marked with BTI instructions. This protects against Jump-Oriented Programming (JOP) attacks. Available in ARMv8.1-M Mainline (Cortex-M) and ARMv8.5-A and later (ARM64).

These features provide return and forward edge validation, respectively. PAC and BTI can be enabled via the `ARM_PACBTI` menu. Available options include `CONFIG_ARM_PACBTI_STANDARD` (both PAC and BTI), `CONFIG_ARM_PACBTI_PACRET` (PAC only), `CONFIG_ARM_PACBTI_BTI` (BTI only), and other variants. Both features can be used independently or combined for comprehensive control flow integrity.

Like x86 CET IBT, these features require compiler support and proper C library instrumentation. PAC requires that functions use the appropriate signing and verification instructions, while BTI

requires that all legitimate branch targets include BTI landing pad instructions. Precompiled C libraries from toolchains typically lack this instrumentation.

For BTI support, the C library must be compiled with `-mbranch-protection` flags to include BTI landing pads in all functions. Therefore, when using any option that enables BTI, only `CONFIG_MINIMAL_LIBC` or picolibc built as a module via `CONFIG_PICOLIBC_USE_MODULE` can be used. Newlib from toolchains is not supported with BTI.

PAC has less strict requirements as it primarily affects function prologues and epilogues, but for optimal security, building the C library with PAC support is recommended.

9.7 Vulnerabilities

This page collects all of the vulnerabilities that are discovered and fixed in each release. It will also often have more details than is available in the releases. Some vulnerabilities are deemed to be sensitive, and will not be publicly discussed until there is sufficient time to fix them. Because the release notes are locked to a version, the information here can be updated after the embargo is lifted.

Vulnerabilities from previous years are collected on separate pages:

9.7.1 CVE-2017

CVE 2017-14199

Buffer overflow in `getaddrinfo()`.

- Zephyr project bug tracker ZEPSEC-12
- PR6158 fix for 1.11.0

CVE 2017-14201

The shell DNS command can cause unpredictable results due to misuse of stack variables.

Use After Free vulnerability in the Zephyr shell allows a serial or telnet connected user to cause denial of service, and possibly remote code execution.

This has been fixed in release v1.14.0.

- Zephyr project bug tracker ZEPSEC-17
- PR13260 fix for v1.14.0

CVE 2017-14202

The shell implementation does not protect against buffer overruns resulting in unpredictable behavior.

Improper Restriction of Operations within the Bounds of a Memory Buffer vulnerability in the shell component of Zephyr allows a serial or telnet connected user to cause a crash, possibly with arbitrary code execution.

This has been fixed in release v1.14.0.

- Zephyr project bug tracker ZEPSEC-18
- PR13048 fix for v1.14.0

9.7.2 CVE-2019

CVE 2019-9506

The Bluetooth BR/EDR specification up to and including version 5.1 permits sufficiently low encryption key length and does not prevent an attacker from influencing the key length negotiation. This allows practical brute-force attacks (aka "KNOB") that can decrypt traffic and inject arbitrary ciphertext without the victim noticing.

- Zephyr project bug tracker ZEPSEC-20
- PR18702 fix for v1.14.0
- PR18659 fix for v2.0.0

9.7.3 CVE-2020

CVE 2020-10019

Buffer Overflow vulnerability in USB DFU of zephyr allows a USB connected host to cause possible remote code execution.

This has been fixed in releases v1.14.2, v2.2.0, and v2.1.1.

- Zephyr project bug tracker ZEPSEC-25
- PR23460 fix for 1.14.x
- PR23457 fix for 2.1.x
- PR23190 fix in 2.2.0

CVE 2020-10021

Out-of-bounds write in USB Mass Storage with unaligned sizes

Out-of-bounds Write in the USB Mass Storage memoryWrite handler with unaligned Sizes.

See NCC-ZEP-024, NCC-ZEP-025, NCC-ZEP-026

This has been fixed in releases v1.14.2, and v2.2.0.

- Zephyr project bug tracker ZEPSEC-26
- PR23455 fix for v1.14.2
- PR23456 fix for the v2.1 branch
- PR23240 fix for v2.2.0

CVE 2020-10022

UpdateHub Module Copies a Variable-Size Hash String Into a Fixed-Size Array

A malformed JSON payload that is received from an UpdateHub server may trigger memory corruption in the Zephyr OS. This could result in a denial of service in the best case, or code execution in the worst case.

See NCC-ZEP-016

This has been fixed in the below pull requests for main, branch from v2.1.0, and branch from v2.2.0.

- Zephyr project bug tracker ZEPSEC-28
- PR24154 fix for main
- PR24065 fix for branch from v2.1.0
- PR24066 fix for branch from v2.2.0

CVE 2020-10023

Shell Subsystem Contains a Buffer Overflow Vulnerability In shell_spaces_trim

The shell subsystem contains a buffer overflow, whereby an adversary with physical access to the device is able to cause a memory corruption, resulting in denial of service or possibly code execution within the Zephyr kernel.

See NCC-ZEP-019

This has been fixed in releases v1.14.2, v2.2.0, and in a branch from v2.1.0,

- Zephyr project bug tracker ZEPSEC-29
- PR23646 fix for v1.14.2
- PR23649 fix for branch from v2.1.0
- PR23304 fix for v2.2.0

CVE 2020-10024

ARM Platform Uses Signed Integer Comparison When Validating Syscall Numbers

The arm platform-specific code uses a signed integer comparison when validating system call numbers. An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

See NCC-ZEP-001

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0,

- Zephyr project bug tracker ZEPSEC-30
- PR23535 fix for v1.14.2
- PR23498 fix for branch from v2.1.0
- PR23323 fix for v2.2.0

CVE 2020-10027

ARC Platform Uses Signed Integer Comparison When Validating Syscall Numbers

An attacker who has obtained code execution within a user thread is able to elevate privileges to that of the kernel.

See NCC-ZEP-001

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0.

- Zephyr project bug tracker ZEPSEC-35
- PR23500 fix for v1.14.2
- PR23499 fix for branch from v2.1.0
- PR23328 fix for v2.2.0

CVE 2020-10028

Multiple Syscalls In GPIO Subsystem Performs No Argument Validation

Multiple syscalls with insufficient argument validation

See NCC-ZEP-006

This has been fixed in releases v1.14.2, and v2.2.0, and in a branch from v2.1.0.

- Zephyr project bug tracker ZEPSEC-32

- PR23733 fix for v1.14.2
- PR23737 fix for branch from v2.1.0
- PR23308 fix for v2.2.0 (gpio patch)

CVE 2020-10058

Multiple Syscalls In kscan Subsystem Performs No Argument Validation

Multiple syscalls in the Kscan subsystem perform insufficient argument validation, allowing code executing in userspace to potentially gain elevated privileges.

See NCC-ZEP-006

This has been fixed in a branch from v2.1.0, and release v2.2.0.

- Zephyr project bug tracker ZEPSEC-34
- PR23748 fix for branch from v2.1.0
- PR23308 fix for v2.2.0 (kscan patch)

CVE 2020-10059

UpdateHub Module Explicitly Disables TLS Verification

The UpdateHub module disables DTLS peer checking, which allows for a man in the middle attack. This is mitigated by firmware images requiring valid signatures. However, there is no benefit to using DTLS without the peer checking.

See NCC-ZEP-018

This has been fixed in a PR against Zephyr main.

- Zephyr project bug tracker ZEPSEC-36
- PR24954 fix on main (to be fixed in v2.3.0)
- PR24954 fix v2.1.0
- PR24954 fix v2.2.0

CVE 2020-10060

UpdateHub Might Dereference An Uninitialized Pointer

In updatehub_probe, right after JSON parsing is complete, objects[1] is accessed from the output structure in two different places. If the JSON contained less than two elements, this access would reference uninitialized stack memory. This could result in a crash, denial of service, or possibly an information leak.

Recommend disabling updatehub until such a time as a fix can be made available.

See NCC-ZEP-030

This has been fixed in a PR against Zephyr main.

- Zephyr project bug tracker ZEPSEC-37
- PR27865 fix on main (to be fixed in v2.4.0)
- PR27865 fix for v2.3.0
- PR27865 fix for v2.2.0
- PR27865 fix for v2.1.0

CVE 2020-10061

Error handling invalid packet sequence

Improper handling of the full-buffer case in the Zephyr Bluetooth implementation can result in memory corruption.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- Zephyr project bug tracker ZEPSEC-75
- PR23516 fix for v2.3 (split driver)
- PR23517 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23547 fix for branch from v2.2.0

CVE 2020-10062

Packet length decoding error in MQTT

CVE: An off-by-one error in the Zephyr project MQTT packet length decoder can result in memory corruption and possible remote code execution. NCC-ZEP-031

The MQTT packet header length can be 1 to 4 bytes. An off-by-one error in the code can result in this being interpreted as 5 bytes, which can cause an integer overflow, resulting in memory corruption.

This has been fixed in main for v2.3.

- Zephyr project bug tracker ZEPSEC-84
- commit 11b7a37d for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE 2020-10063

Remote Denial of Service in CoAP Option Parsing Due To Integer Overflow

A remote adversary with the ability to send arbitrary CoAP packets to be parsed by Zephyr is able to cause a denial of service.

This has been fixed in main for v2.3.

- Zephyr project bug tracker ZEPSEC-55
- PR24435 fix in main for v2.3
- PR24531 fix for branch from v2.2
- PR24535 fix for branch from v2.1
- PR24530 fix for branch from v1.14
- NCC-ZEP report (NCC-ZEP-032)

CVE 2020-10064

Improper Input Frame Validation in ieee802154 Processing

- Zephyr project bug tracker ZEPSEC-65
- PR24971 fix for v2.4
- PR33451 fix for v1.4

CVE 2020-10065

OOB Write after not validating user-supplied length (<= 0xffff) and copying to fixed-size buffer (default: 77 bytes) for HCI_ACL packets in bluetooth HCI over SPI driver.

- Zephyr project bug tracker ZEPSEC-66
- This issue has not been fixed.

CVE 2020-10066

Incorrect Error Handling in Bluetooth HCI core

In hci_cmd_done, the buf argument being passed as null causes nullpointer dereference.

- Zephyr project bug tracker ZEPSEC-67
- PR24902 fix for v2.4
- PR25089 fix for v1.4

CVE 2020-10067

Integer Overflow In is_in_region Allows User Thread To Access Kernel Memory

A malicious userspace application can cause a integer overflow and bypass security checks performed by system call handlers. The impact would depend on the underlying system call and can range from denial of service to information leak to memory corruption resulting in code execution within the kernel.

See NCC-ZEP-005

This has been fixed in releases v1.14.2, and v2.2.0.

- Zephyr project bug tracker ZEPSEC-27
- PR23653 fix for v1.14.2
- PR23654 fix for the v2.1 branch
- PR23239 fix for v2.2.0

CVE 2020-10068

Zephyr Bluetooth DLE duplicate requests vulnerability

In the Zephyr project Bluetooth subsystem, certain duplicate and back-to-back packets can cause incorrect behavior, resulting in a denial of service.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- Zephyr project bug tracker ZEPSEC-78
- PR23707 fix for v2.3 (split driver)
- PR23708 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23964 fix for v2.2.0

CVE 2020-10069

Zephyr Bluetooth unchecked packet data results in denial of service

An unchecked parameter in bluetooth data can result in an assertion failure, or division by zero, resulting in a denial of service attack.

This has been fixed in branches for v1.14.0, v2.2.0, and will be included in v2.3.0.

- Zephyr project bug tracker ZEPSEC-81
- PR23705 fix for v2.3 (split driver)
- PR23706 fix for v2.3 (legacy driver)
- PR23091 fix for branch from v1.14.0
- PR23963 fix for branch from v2.2.0

CVE 2020-10070

MQTT buffer overflow on receive buffer

In the Zephyr Project MQTT code, improper bounds checking can result in memory corruption and possibly remote code execution. NCC-ZEP-031

When calculating the packet length, arithmetic overflow can result in accepting a receive buffer larger than the available buffer space, resulting in user data being written beyond this buffer.

This has been fixed in main for v2.3.

- Zephyr project bug tracker ZEPSEC-85
- commit 0b39cbf3 for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE 2020-10071

Insufficient publish message length validation in MQTT

The Zephyr MQTT parsing code performs insufficient checking of the length field on publish messages, allowing a buffer overflow and potentially remote code execution. NCC-ZEP-031

This has been fixed in main for v2.3.

- Zephyr project bug tracker ZEPSEC-86
- commit 989c4713 fix for v2.3
- NCC-ZEP report (NCC-ZEP-031)

CVE 2020-10072

All threads can access all socket file descriptors

There is no management of permissions to network socket API file descriptors. Any thread running on the system may read/write a socket file descriptor knowing only the numerical value of the file descriptor.

- Zephyr project bug tracker ZEPSEC-87
- PR25804 fix for v2.4
- PR27176 fix for v1.4

CVE 2020-10136

IP-in-IP protocol routes arbitrary traffic by default zephyrproject

- Zephyr project bug tracker ZEPSEC-64

CVE 2020-13598

FS: Buffer Overflow when enabling Long File Names in FAT_FS and calling fs_stat

Performing fs_stat on a file with a filename longer than 12 characters long will cause a buffer overflow.

- Zephyr project bug tracker ZEPSEC-88
- PR25852 fix for v2.4
- PR28782 fix for v2.3
- PR33577 fix for v1.4

CVE 2020-13599

Security problem with settings and littlefs

When settings is used in combination with littlefs all security related information can be extracted from the device using MCUmgr and this could be used e.g in bt-mesh to get the device key, network key, app keys from the device.

- Zephyr project bug tracker ZEPSEC-57
- PR26083 fix for v2.4

CVE 2020-13600

Malformed SPI in response for eswifi can corrupt kernel memory

- Zephyr project bug tracker ZEPSEC-91
- PR26712 fix for v2.4

CVE 2020-13601

Possible read out of bounds in dns read

- Zephyr project bug tracker ZEPSEC-92
- PR27774 fix for v2.4
- PR30503 fix for v1.4

CVE 2020-13602

Remote Denial of Service in LwM2M do_write_op_tlv

In the Zephyr LwM2M implementation, malformed input can result in an infinite loop, resulting in a denial of service attack.

- Zephyr project bug tracker ZEPSEC-56
- PR26571 fix for v2.4
- PR33578 fix for v1.4

CVE 2020-13603

Possible overflow in mempool

- Zephyr offers pre-built 'malloc' wrapper function instead.
- The 'malloc' function is wrapper for the 'sys_mem_pool_alloc' function
- sys_mem_pool_alloc allocates 'size + WB_UP(sizeof(struct sys_mem_pool_block))' in an unsafe manner.

- Asking for very large size values leads to internal integer wrap-around.
- Integer wrap-around leads to successful allocation of very small memory.
- For example: calling malloc(0xffffffff) leads to successful allocation of 7 bytes.
- That leads to heap overflow.
- Zephyr project bug tracker ZEPSEC-111
- PR31796 fix for v2.4
- PR32808 fix for v1.4

9.7.4 CVE-2021

CVE 2021-3319

DOS: Incorrect 802154 Frame Validation for Omitted Source / Dest Addresses

Improper processing of omitted source and destination addresses in ieee802154 frame validation (ieee802154_validate_frame)

This has been fixed in main for v2.5.0

- Zephyr project bug tracker GHSA-94jg-2p6q-5364
- PR31908 fix for main

CVE 2021-3320

Mismatch between validation and handling of 802154 ACK frames, where ACK frames are considered during validation, but not during actual processing, leading to a type confusion.

- PR31908 fix for main

CVE 2021-3321

Incomplete check of minimum IEEE 802154 fragment size leading to an integer underflow.

- Zephyr project bug tracker ZEPSEC-114
- PR33453 fix for v2.4

CVE 2021-3323

Integer Underflow in 6LoWPAN IPHC Header Uncompression

This has been fixed in main for v2.5.0

- Zephyr project bug tracker GHSA-89j6-qpxf-pfpc
- PR 31971 fix for main

CVE 2021-3430

Assertion reachable with repeated LL_CONNECTION_PARAM_REQ.

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-46h3-hjcq-2jjr
- PR 33272 fix for main
- PR 33369 fix for 2.5
- PR 33759 fix for 1.14.2

CVE 2021-3431

BT: Assertion failure on repeated LL_FEATURE_REQ

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-7548-5m6f-mqv9
- PR 33340 fix for main
- PR 33369 fix for 2.5

CVE 2021-3432

Invalid interval in CONNECT_IND leads to Division by Zero

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-7364-p4wc-8mj4
- PR 33278 fix for main
- PR 33369 fix for 2.5

CVE 2021-3433

BT: Invalid channel map in CONNECT_IND results to Deadlock

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-3c2f-w4v6-qxrp
- PR 33278 fix for main
- PR 33369 fix for 2.5

CVE 2021-3434

L2CAP: Stack based buffer overflow in le_ecred_conn_req()

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-8w87-6rfp-cfrm
- PR 33305 fix for main
- PR 33419 fix for 2.5
- PR 33418 fix for 1.14.2

CVE 2021-3435

L2CAP: Information leakage in le_ecred_conn_req()

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-xhg3-gvj6-4rqh
- PR 33305 fix for main
- PR 33419 fix for 2.5
- PR 33418 fix for 1.14.2

CVE 2021-3436

Bluetooth: Possible to overwrite an existing bond during keys distribution phase when the identity address of the bond is known

During the distribution of the identity address information we don't check for an existing bond with the same identity address.This means that a duplicate entry will be created in RAM while the newest entry will overwrite the existing one in persistent storage.

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-j76f-35mc-4h63
- PR 33266 fix for main
- PR 33432 fix for 2.5
- PR 33433 fix for 2.4
- PR 33718 fix for 1.14.2

CVE 2021-3454

Truncated L2CAP K-frame causes assertion failure

For example, sending L2CAP K-frame where SDU length field is truncated to only one byte, causes assertion failure in previous releases of Zephyr. This has been fixed in master by commit 0ba9437 but has not yet been backported to older release branches.

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-fx88-6c29-vrp3
- PR 32588 fix for main
- PR 33513 fix for 2.5
- PR 33514 fix for 2.4

CVE 2021-3455

Disconnecting L2CAP channel right after invalid ATT request leads freeze

When Central device connects to peripheral and creates L2CAP connection for Enhanced ATT, sending some invalid ATT request and disconnecting immediately causes freeze.

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-7g38-3x9v-v7vp
- PR 35597 fix for main
- PR 36104 fix for 2.5
- PR 36105 fix for 2.4

CVE 2021-3510

Zephyr JSON decoder incorrectly decodes array of array

When using JSON_OBJ_DESCR_ARRAY_ARRAY, the subarray is has the token type JSON_TOK_LIST_START, but then assigns to the object part of the union. arr_parse then takes the offset of the array-object (which has nothing todo with the list) treats it as relative to the parent object, and stores the length of the subarray in there.

This has been fixed in main for v2.7.0

- Zephyr project bug tracker GHSA-289f-7mw3-2qf4
- PR 36340 fix for main

- PR 37816 fix for 2.6

CVE 2021-3581

HCI data not properly checked leads to memory overflow in the Bluetooth stack

In the process of setting SCAN_RSP through the HCI command, the Zephyr Bluetooth protocol stack did not effectively check the length of the incoming HCI data. Causes memory overflow, and then the data in the memory is overwritten, and may even cause arbitrary code execution.

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-8q65-5gqf-fmw5
- PR 35935 fix for main
- PR 35984 fix for 2.5
- PR 35985 fix for 2.4
- PR 35985 fix for 1.14

CVE 2021-3625

Buffer overflow in Zephyr USB DFU DNLOAD

This has been fixed in main for v2.6.0

- Zephyr project bug tracker GHSA-c3gr-hgvr-f363
- PR 36694 fix for main

CVE 2021-3835

Buffer overflow in Zephyr USB device class

This has been fixed in main for v3.0.0

- Zephyr project bug tracker GHSA-fm6v-8625-99jf
- PR 42093 fix for main
- PR 42167 fix for 2.7

CVE 2021-3861

Buffer overflow in the RNDIS USB device class

This has been fixed in main for v3.0.0

- Zephyr project bug tracker GHSA-hvfp-w4h8-gxvj
- PR 39725 fix for main

CVE 2021-3966

Usb bluetooth device ACL read cb buffer overflow

This has been fixed in main for v3.0.0

- Zephyr project bug tracker GHSA-hfxq-3w6x-fv2m
- PR 42093 fix for main
- PR 42167 fix for v2.7.0

9.7.5 CVE-2022

CVE 2022-0553

Possible to retrieve unencrypted firmware image

This has been fixed in main for v3.0.0

- Zephyr project bug tracker GHSA-wrj2-9vj9-rrcp
- PR 42424 fix for main

CVE 2022-1041

Out-of-bound write vulnerability in the Bluetooth Mesh core stack can be triggered during provisioning

This has been fixed in main for v3.1.0

- Zephyr project bug tracker GHSA-p449-9hv9-pj38
- PR 45136 fix for main
- PR 45188 fix for v3.0.0
- PR 45187 fix for v2.7.0

CVE 2022-1042

Out-of-bound write vulnerability in the Bluetooth Mesh core stack can be triggered during provisioning

This has been fixed in main for v3.1.0

- Zephyr project bug tracker GHSA-j7v7-w73r-mm5x
- PR 45066 fix for main
- PR 45135 fix for v3.0.0
- PR 45134 fix for v2.7.0

CVE 2022-1841

Out-of-Bound Write in tcp_flags

This has been fixed in main for v3.1.0

- Zephyr project bug tracker GHSA-5c3j-p8cr-2pgh
- PR 45796 fix for main

CVE 2022-2741

can: denial-of-service can be triggered by a crafted CAN frame

This has been fixed in main for v3.2.0

- Zephyr project bug tracker GHSA-hx5v-j59q-c3j8
- PR 47903 fix for main
- PR 47957 fix for v3.1.0
- PR 47958 fix for v3.0.0
- PR 47959 fix for v2.7.0

CVE 2022-2993

bt: host: Wrong key validation check

This has been fixed in main for v3.2.0

- Zephyr project bug tracker GHSA-3286-jgjx-8cvr
- PR 48733 fix for main

CVE 2022-3806

DoS: Invalid Initialization in le_read_buffer_size_complete()

- Zephyr project bug tracker GHSA-w525-fm68-ppq3

9.7.6 CVE-2023

CVE 2023-0396

Buffer Overreads in Bluetooth HCI

- Zephyr project bug tracker GHSA-8rpp-6vxq-pqg3

CVE 2023-0397

DoS: Invalid Initialization in le_read_buffer_size_complete()

- Zephyr project bug tracker GHSA-wc2h-h868-q7hj

This has been fixed in main for v3.3.0

- PR 54905 fix for main
- PR 47957 fix for v3.2.0
- PR 47958 fix for v3.1.0
- PR 47959 fix for v2.7.4

CVE 2023-0779

net: shell: Improper input validation

- Zephyr project bug tracker GHSA-9xj8-6989-r549

This has been fixed in main for v3.3.0

- PR 54371 fix for main
- PR 54380 fix for v3.2.0
- PR 54381 fix for v2.7.4

CVE 2023-1901

HCI send_sync Dangling Semaphore Reference Re-use

- Zephyr project bug tracker GHSA-xvvm-8mcm-9cq3

This has been fixed in main for v3.4.0

- PR 56709 fix for main

CVE 2023-1902

HCI Connection Creation Dangling State Reference Re-use

- Zephyr project bug tracker GHSA-fx9g-8fr2-q899

This has been fixed in main for v3.4.0

- PR 56709 fix for main

CVE 2023-3725

Potential buffer overflow vulnerability in the Zephyr CANbus subsystem.

- Zephyr project bug tracker GHSA-2g3m-p6c7-8rr3

This has been fixed in main for v3.5.0

- PR 61502 fix for main
- PR 61518 fix for 3.4
- PR 61517 fix for 3.3
- PR 61516 fix for 2.7

CVE 2023-4257

Unchecked user input length in the Zephyr WiFi shell module can cause buffer overflows.

- Zephyr project bug tracker GHSA-853q-q69w-gf5j

This has been fixed in main for v3.5.0

- PR 605377 fix for main
- PR 61383 fix for 3.4

CVE 2023-4258

bt: mesh: vulnerability in provisioning protocol implementation on provisionee side

- Zephyr project bug tracker GHSA-m34c-cp63-rwh7

This has been fixed in main for v3.5.0

- PR 59467 fix for main
- PR 60078 fix for 3.4
- PR 60079 fix for 3.3

CVE 2023-4259

Buffer overflow vulnerabilities in the Zephyr eS-WiFi driver

- Zephyr project bug tracker GHSA-gghm-c696-f4j4

This has been fixed in main for v3.5.0

- PR 63074 fix for main
- PR 63750 fix for main

CVE 2023-4260

Off-by-one buffer overflow vulnerability in the Zephyr FS subsystem

- Zephyr project bug tracker GHSA-gj27-862r-55wh

This has been fixed in main for v3.5.0

- PR 63079 fix for main

CVE 2023-4262

- This issue has been determined to be a false positive after further analysis.

CVE 2023-4263

Potential buffer overflow vulnerability in the Zephyr IEEE 802.15.4 nRF 15.4 driver.

- Zephyr project bug tracker GHSA-rf6q-rhhp-pqhf

This has been fixed in main for v3.5.0

- PR 60528 fix for main
- PR 61384 fix for 3.4
- PR 61216 fix for 2.7

CVE 2023-4264

Potential buffer overflow vulnerabilities in the Zephyr Bluetooth subsystem

- Zephyr project bug tracker GHSA-rgx6-3w4j-gf5j

This has been fixed in main for v3.5.0

- PR 58834 fix for main
- PR 60465 fix for main
- PR 61845 fix for main
- PR 61385 fix for 3.4

CVE 2023-4265

Two potential buffer overflow vulnerabilities in Zephyr USB code

- Zephyr project bug tracker GHSA-4vgv-5r6q-r6xh

This has been fixed in main for v3.4.0

- PR 59157 fix for main
- PR 59018 fix for main

CVE 2023-4424

bt: hci: DoS and possible RCE

- Zephyr project bug tracker GHSA-j4qm-xgpf-qjw3

This has been fixed in main for v3.5.0

- PR 61651 fix for main
- PR 61696 fix for 3.4
- PR 61695 fix for 3.3

- PR 61694 fix for 2.7

CVE 2023-5055

L2CAP: Possible Stack based buffer overflow in le_ecred_reconf_req()

- Zephyr project bug tracker GHSA-wr8r-7f8x-24jj

This has been fixed in main for v3.5.0

- PR 62381 fix for main

CVE 2023-5139

Potential buffer overflow vulnerability in the Zephyr STM32 Crypto driver.

- Zephyr project bug tracker GHSA-rhrc-pcxp-4453

This has been fixed in main for v3.5.0

- PR 61839 fix for main

CVE 2023-5184

Potential signed to unsigned conversion errors and buffer overflow vulnerabilities in the Zephyr IPM driver

- Zephyr project bug tracker GHSA-8x3p-q3r5-xh9g

This has been fixed in main for v3.5.0

- PR 63069 fix for main

CVE 2023-5563

The SJA1000 CAN controller driver backend automatically attempts to recover from a bus-off event when built with CONFIG_CAN_AUTO_BUS_OFF_RECOVERY=y. This results in calling k_sleep() in IRQ context, causing a fatal exception.

- Zephyr project bug tracker GHSA-98mc-rj7w-7rpv

This has been fixed in main for v3.5.0

- PR 63713 fix for main
- PR 63718 fix for 3.4
- PR 63717 fix for 3.3

CVE 2023-5753

Potential buffer overflow vulnerabilities in the Zephyr Bluetooth subsystem source code when asserts are disabled.

- Zephyr project bug tracker GHSA-hmpr-px56-rvww

This has been fixed in main for v3.5.0

- PR 63605 fix for main

CVE 2023-5779

Out of bounds issue in remove_rx_filter in multiple can drivers.

- Zephyr project bug tracker GHSA-7cmj-963q-jj47

This has been fixed in main for v3.6.0

- PR 64399 fix for main
- PR 64416 fix for 3.5
- PR 64415 fix for 3.4
- PR 64427 fix for 3.3
- PR 64431 fix for 2.7

CVE 2023-6249

Signed to unsigned conversion problem in esp32_ipm_send may lead to buffer overflow

- Zephyr project bug tracker GHSA-32f5-3p9h-2rqc

This has been fixed in main for v3.6.0

- PR 65546 fix for main

CVE 2023-6749

Potential buffer overflow due unchecked data coming from user input in settings shell.

- Zephyr project bug tracker GHSA-757h-rw37-66hw

This has been fixed in main for v3.6.0

- PR 66451 fix for main
- PR 66584 fix for 3.5

CVE 2023-6881

Potential buffer overflow vulnerability in Zephyr fuse file system.

- Zephyr project bug tracker GHSA-mh67-4h3q-p437

This has been fixed in main for v3.6.0

- PR 66592 fix for main

CVE 2023-7060

Missing Security Control in Zephyr OS IP Packet Handling

- Zephyr project bug tracker GHSA-fjc8-223c-qgqr

This has been fixed in main for v3.6.0

- PR 66645 fix for main
- PR 66739 fix for 3.5
- PR 66738 fix for 3.4
- PR 66887 fix for 2.7

9.7.7 CVE-2024

CVE 2024-1638

Bluetooth characteristic LESC security requirement not enforced without additional flags

- Zephyr project bug tracker GHSA-p6f3-f63q-5mc2

This has been fixed in main for v3.6.0

- PR 69170 fix for main

CVE 2024-3077

Bluetooth: Integer underflow in gatt_find_info_rsp. A malicious Bluetooth LE device can crash Bluetooth LE victim device by sending malformed gatt packet.

- Zephyr project bug tracker GHSA-gmfv-4vfh-2mh8

This has been fixed in main for v3.7.0

- PR 69396 fix for main

CVE 2024-3332

Bluetooth: DoS caused by null pointer dereference.

A malicious Bluetooth LE device can send a specific order of packet sequence to cause a DoS attack on the victim Bluetooth LE device.

- Zephyr project bug tracker GHSA-jmr9-xw2v-5vf4

This has been fixed in main for v3.7.0

- PR 71030 fix for main

CVE 2024-4785

Bluetooth: Missing Check in LL_CONNECTION_UPDATE_IND Packet Leads to Division by Zero

- Zephyr project bug tracker GHSA-xcr5-5g98-mchp

This has been fixed in main for v3.7.0

- PR 72608 fix for main

CVE 2024-5754

BT: Encryption procedure host vulnerability

- Zephyr project bug tracker GHSA-gvv5-66hw-5qrc

This has been fixed in main for v3.7.0

- PR 7395 fix for main
- PR 74124 fix for 3.6
- PR 74123 fix for 3.5
- PR 74122 fix for 2.7

CVE 2024-5931

BT: Unchecked user input in bap_broadcast_assistant

- Zephyr project bug tracker GHSA-r8h3-64gp-wv7f

This has been fixed in main for v3.7.0

- PR 74062 fix for main
- PR 77966 fix for 3.6

CVE 2024-6135

BT:Classic: Multiple missing buf length checks

- Zephyr project bug tracker GHSA-2mp4-4g6f-cqcx

This has been fixed in main for v3.7.0

- PR 74283 fix for main
- PR 77964 fix for 3.6

CVE 2024-6137

BT: Classic: SDP OOB access in get_att_search_list

- Zephyr project bug tracker GHSA-pm38-7g85-cf4f

This has been fixed in main for v3.7.0

- PR 75575 fix for main

CVE 2024-6258

BT: Missing length checks of net_buf in rfcomm_handle_data

- Zephyr project bug tracker GHSA-7833-fcpm-3ggm

This has been fixed in main for v3.7.0

- PR 74640 fix for main

CVE 2024-6259

BT: HCI: adv_ext_report Improper discarding in adv_ext_report

- Zephyr project bug tracker GHSA-p5j7-v26w-wmcp

This has been fixed in main for v3.7.0

- PR 74639 fix for main
- PR 77960 fix for 3.6

CVE 2024-6442

Bluetooth: ASCS Unchecked tailroom of the response buffer

- Zephyr project bug tracker GHSA-m22j-ccg7-4v4h

This has been fixed in main for v3.7.0

- PR 74976 fix for main
- PR 77958 fix for 3.6

CVE 2024-6443

zephyr: out-of-bound read in utf8_trunc

- Zephyr project bug tracker GHSA-gg46-3rh2-v765

This has been fixed in main for v3.7.0

- PR 74949 fix for main
- PR 78286 fix for 3.6

CVE 2024-6444

Bluetooth: ots: missing buffer length check

- Zephyr project bug tracker GHSA-qj4r-chj6-h7qp

This has been fixed in main for v3.7.0

- PR 74944 fix for main
- PR 77954 fix for 3.6

CVE 2024-8798

Bluetooth: classic: avdtp: missing buffer length check

- Zephyr project bug tracker GHSA-r7pm-f93f-f7fp

This has been fixed in main for v4.0.0

- PR 77969 fix for main
- PR 78409 fix for 3.7

CVE 2024-10395

net: lib: http_server: Buffer Under-read

No proper validation of the length of user input in http_server_get_content_type_from_extension could cause a segmentation fault or crash by causing memory to be read outside of the bounds of the buffer.

- Zephyr project bug tracker GHSA-hfww-j92m-x8fv

This has been fixed in main for v4.0.0

- PR 80396 fix for main

CVE 2024-11263

arch: riscv: userspace: potential security risk when CONFIG_RISCV_GP=y

A rogue thread can corrupt the gp reg and cause the entire system to hard fault at best, at worst, it can potentially trick the system to access another set of random global symbols.

- Zephyr project bug tracker GHSA-jjf3-7x72-pqm9

This has been fixed in main for v4.0.0

- PR 81155 fix for main
- PR 81370 fix for 3.7

9.7.8 CVE-2025

CVE 2025-1673

Out of bounds read when calling crc16_ansi and strlen in dns_validate_msg

A malicious or malformed DNS packet without a payload can cause an out-of-bounds read, resulting in a crash (denial of service) or an incorrect computation.

- Zephyr project bug tracker GHSA-jjhx-rrh4-j8mx

This has been fixed in main for v4.1.0

- PR 82072 fix for main
- PR 82289 fix for 4.0
- PR 82288 fix for 3.7

CVE 2025-1674

Out of bounds read when unpacking DNS answers

A lack of input validation allows for out of bounds reads caused by malicious or malformed packets.

- Zephyr project bug tracker GHSA-x975-8pgf-qh66

This has been fixed in main for v4.1.0

- PR 82072 fix for main
- PR 82289 fix for 4.0
- PR 82288 fix for 3.7

CVE 2025-1675

Out of bounds read in dns_copy_qname

The function dns_copy_qname in dns_pack.c performs performs a memcpy operation with an untrusted field and does not check if the source buffer is large enough to contain the copied data.

- Zephyr project bug tracker GHSA-2m84-5hfw-m8v4

This has been fixed in main for v4.1.0

- PR 82072 fix for main
- PR 82289 fix for 4.0
- PR 82288 fix for 3.7

CVE 2025-2962

Infinite loop in dns_copy_qname

A denial-of-service issue in the dns implementation could cause an infinite loop.

- Zephyr project bug tracker GHSA-2qp5-c2vq-g2ww

This has been fixed in main for v4.2.0

- PR 87753 fix for main
- PR 87925 fix for 4.1
- PR 87949 fix for 3.7

CVE 2025-7403

Bluetooth: bt_conn_tx_processor unsafe handling

Unsafe handling in bt_conn_tx_processor causes a use-after-free, resulting in a write-before-zero. The written 4 bytes are attacker-controlled, enabling precise memory corruption.

- Zephyr project bug tracker GHSA-9r46-cqqw-6j2j

This has been fixed in main for v4.2.0

- PR 90975 fix for main

CVE 2025-10456

Bluetooth: Semi-Arbitrary ability to make the BLE Target send disconnection requests

A vulnerability was identified in the handling of Bluetooth Low Energy (BLE) fixed channels (such as SMP or ATT). Specifically, an attacker could exploit a flaw that causes the BLE target (i.e., the device under attack) to attempt to disconnect a fixed channel, which is not allowed per the Bluetooth specification. This leads to undefined behavior, including potential assertion failures, crashes, or memory corruption.

- Zephyr project bug tracker GHSA-hcc8-3qr7-c9m8

This has been fixed in main for v4.2.0

- PR 93576 fix for main
- PR 100474 fix for 3.7

CVE 2025-10457

Bluetooth: Out-Of-Context le_conn_rsp handling

The function responsible for handling BLE connection responses does not verify whether a response is expected—that is, whether the device has initiated a connection request. Instead, it relies solely on identifier matching.

- Zephyr project bug tracker GHSA-xqj6-vh76-2vv8

This has been fixed in main for v4.2.0

- PR 94080 fix for main

CVE 2025-10458

Bluetooth: le_conn_rsp does not sanitize CID, MTU, MPS values

Parameters are not validated or sanitized, and are later used in various internal operations.

- Zephyr project bug tracker GHSA-vmww-237q-2fwp

This has been fixed in main for v4.2.0

- PR 93174 fix for main

CVE 2025-9408

Userspace privilege escalation vulnerability on Cortex M

System call entry on Cortex M (and possibly R and A, but I think not) has a race which allows very practical privilege escalation for malicious userspace processes.

- Zephyr project bug tracker GHSA-3r6j-5mp3-75wr

This has been fixed in main for v4.3.0

- PR 95101 fix for main
- PR 96850 fix for main
- PR 96014 fix for 4.2
- PR 97306 fix for 4.2
- PR 96015 fix for 4.1
- PR 97305 fix for 4.1
- PR 96030 fix for 3.7
- PR 97313 fix for 3.7

CVE 2025-9557

Bluetooth: Mesh: Out-of-Bound Write in gen_prov_cont

An out-of-bound write can lead to an arbitrary code execution. Even on devices with some form of memory protection, this can still lead to a crash and a resultant denial of service.

- Zephyr project bug tracker GHSA-r3j3-c5v7-2ppf

This has been fixed in main for v4.3.0

- PR 95061 fix for main
- PR 97518 fix for 4.2
- PR 97517 fix for 4.1

CVE 2025-9558

Bluetooth: Mesh: Out-of-Bound Write in gen_prov_start

There is a potential OOB Write vulnerability in the gen_prov_start function in pb_adv.c. The full length of the received data is copied into the link.rx.buf receiver buffer without any validation on the data size.

- Zephyr project bug tracker GHSA-8wvr-688x-68vr

This has been fixed in main for v4.3.0

- PR 95064 fix for main
- PR 97520 fix for 4.2
- PR 97519 fix for 4.1

CVE 2025-12035

Bluetooth: Integer Overflow in Bluetooth Classic (BR/EDR) L2CAP

An integer overflow condition exists in Bluetooth Host stack, within the bt_br_acl_recv routine a critical path for processing inbound BR/EDR L2CAP traffic.

- Zephyr project bug tracker GHSA-p793-3456-h7w3

This has been fixed in main for v4.3.0

- PR 97370 fix for main

CVE 2025-12890

Bluetooth: peripheral: Invalid handling of malformed connection request

Improper handling of malformed Connection Request with the interval set to be 1 (which supposed to be illegal) and the chM 0x7CFFFFFFFF triggers a crash. The peripheral will not be connectable after it.

- Zephyr project bug tracker GHSA-8hrf-pfww-83v9

This has been fixed in main for v4.2.0

- PR 89955 fix for main

CVE 2025-12899

net: icmp: Out of bound memory read

A flaw in Zephyr's network stack allows an IPv4 packet containing ICMP type 128 to be misclassified as an ICMPv6 Echo Request. This results in an out-of-bounds memory read and creates a potential information-leak vulnerability in the networking subsystem.

- Zephyr project bug tracker GHSA-c2vg-hj83-c2vg

This has been fixed in main for v4.3.0

- PR 98780 fix for main
- PR 98983 fix for 4.2
- PR 98984 fix for 4.1
- PR 98985 fix for 3.7

9.7.9 CVE-2026

CVE 2026-0849

crypto: ATAES132A response length allows stack buffer overflow

Malformed ATAES132A responses with an oversized length field overflow a 52-byte stack buffer in the Zephyr crypto driver, allowing a compromised device or bus attacker to corrupt kernel memory and potentially hijack execution.

- Zephyr project bug tracker GHSA-ff4p-3ggg-prp6

This has been fixed in main for v4.4.0

- PR 103163 fix for main

CVE 2026-1677

net: TLS 1.2 connections allowed on TLS 1.3 sockets

Zephyr sockets created with `IPPROTO_TLS_1_3` can still negotiate a TLS 1.2 connection when both TLS versions are enabled in Kconfig, because the socket-level protocol selection is not propagated to mbedTLS (e.g. via `mbedtls_ssl_conf_min_tls_version`). The ClientHello advertises both versions and the peer can establish TLS 1.2, so applications that assumed `IPPROTO_TLS_1_3` enforces TLS 1.3 may silently use TLS 1.2 and remain exposed to TLS 1.2-specific weaknesses.

- Zephyr project bug tracker GHSA-23r2-m5wx-4rvq

This has been fixed in main for v4.4.0

- PR 102570 fix for main

CVE 2026-1678

dns: memory-safety issue in the DNS name parser

`dns_unpack_name()` caches the buffer tailroom once and reuses it while appending DNS labels. As the buffer grows, the cached size becomes incorrect, and the final null terminator can be written past the buffer. With assertions disabled (default), a malicious DNS response can trigger an out-of-bounds write when `CONFIG_DNS_RESOLVER` is enabled.

- Zephyr project bug tracker GHSA-536f-h63g-hj42

This has been fixed in main for v4.4.0

- PR 99683 fix for main
- PR 99830 fix for 4.3
- PR 99829 fix for 4.2
- PR 99828 fix for 3.7

CVE 2026-1679

The eswifi socket offload driver copies user-provided payloads into a fixed buffer without checking available space; oversized sends overflow eswifi->buf, corrupting kernel memory (CWE-120). Exploit requires local code that can call the socket send API; no remote attacker can reach it directly.

- Zephyr project bug tracker GHSA-qx3g-5g22-fq5w

This has been fixed in main for v4.4.0

- PR 102119 fix for main

CVE 2026-1681

net: Stack Overflow with Ping (to own IP Address) via Shell

Issuing an ICMP ping via the `net ping` shell command to a device's own IPv4 address causes the network stack to recursively re-enter the input path on the same system work-queue stack. Because the destination is recognized as a local address, both the echo request and the resulting echo reply are processed inline before the current frame returns. The nested input-path frames exceed the work-queue stack and trigger a stack overflow.

- Zephyr project bug tracker GHSA-6fcc-8rwr-w7xx

This has been fixed in main for v4.4.0

- PR 102268 fix for main

CVE 2026-4179

stm32: usb: Infinite while loop in Interrupt Handler

Issues in stm32 USB device driver can lead to an infinite while loop.

- Zephyr project bug tracker GHSA-9xg7-g3q3-9prf

This has been fixed in main for v4.4.0

- PR 104390 fix for main

CVE 2026-5066

net: sockets: tls: Potential out-of-bounds write/read in socket_op_vtable::connect function

A potential out-of-bounds write/read exists in the TLS socket connect path of the network sockets subsystem (`subsys/net/lib/sockets/sockets_tls.c`). When the TLS session cache is enabled, `tls_session_store()` and `tls_session_restore()` `memcpy` the caller-supplied address into a fixed-size buffer using the caller-controlled `addrlen` value without validating it against the destination size. Since `struct net_sockaddr` is an opaque type, an application can pass an `addrlen` larger than `sizeof(struct net_sockaddr)` (for example 128 bytes into a 24-byte stack buffer), causing the `memcpy` to read and write past the end of the address memory used by the TLS session cache. This can lead to a crash and denial of service, and potentially to arbitrary code execution.

- Zephyr project bug tracker GHSA-wgrc-jrf6-24f3

This has been fixed in main for v4.4.0

- PR 104871 fix for main
- PR 105044 fix for 4.3
- PR 105043 fix for 4.2
- PR 105042 fix for 3.7

CVE 2026-5067

Out-of-bounds read/write in HTTP WebSocket upgrade via non-null-terminated Sec-WebSocket-Key

A remote, unauthenticated attacker can trigger memory corruption in Zephyr's HTTP server WebSocket upgrade path by sending a crafted `Sec-WebSocket-Key` header that is copied without guaranteed NUL termination and then passed to `strlen()`. This can cause out-of-bounds read and out-of-bounds write on stack memory, leading to a crash (denial of service) and potentially code execution. The path is reachable when `CONFIG_HTTP_SERVER_WEBSOCKET` is enabled.

- Zephyr project bug tracker GHSA-wgr4-9pwq-94vj

This has been fixed in main for v4.4.0

- PR 104740 fix for main
- PR 107927 fix for 4.3
- PR 107926 fix for 3.7

CVE 2026-5068

Bluetooth: L2CAP LE CoC: remote out-of-bounds write via segmentation counter stored in net_buf user_data

A remote, unauthenticated BLE peer can trigger a 2-byte out-of-bounds write in the Bluetooth host during L2CAP LE CoC SDU reassembly. When the application enables segmentation (via `chan_ops.alloc_buf`) and the chosen RX pool has a `user_data_size` smaller than 2 bytes, the segmentation counter stored in the `net_buf` user_data area is written out of bounds in `l2cap_chan_le_recv_seg` (`subsys/bluetooth/host/l2cap.c`). This can lead to heap corruption and a fatal error.

- Zephyr project bug tracker GHSA-qrcq-hxwj-mqxm

This has been fixed in main for v4.4.0

- PR 104913 fix for main
- PR 108335 fix for 4.3
- PR 108336 fix for 3.7

CVE 2026-5071

can: Local Denial of Service via SocketCAN Send

The SocketCAN send path (`zcan_sendto_ctx`) validated the caller-supplied buffer length with a `NET_ASSERT` instead of a real runtime check. In production builds where assertions are compiled out, a userspace app could pass a buffer shorter than `struct socketcan_frame`, and `socketcan_to_can_frame()` would dereference fields past the end of that buffer — an out-of-bounds read that can crash the system (local DoS) or, because the parsed frame is then transmitted, potentially leak adjacent memory.

- Zephyr project bug tracker GHSA-c3w6-x7m3-3c58

This has been fixed in main for v4.4.0

- PR 104654 fix for main
- PR 104679 fix for 4.3
- PR 104678 fix for 4.2
- PR 104677 fix for 3.7

CVE 2026-5072

net: ptp: Potential Denial of Service via PTP Interval Shift

A bitwise shift vulnerability allows a remote attacker to cause undefined behavior and potential crashes in the PTP subsystem by sending a crafted PTP Management or Delay Response packet containing a large, unvalidated, negative log_announce_interval used in the bitwise shift operation.

- Zephyr project bug tracker GHSA-3v98-458v-388r

This has been fixed in main for v4.4.0

- PR 104613 fix for main
- PR 108337 fix for v4.3
- PR 108338 fix for v3.7

CVE 2026-5589

Bluetooth: Mesh: Out-of-bounds write caused by an integer underflow

An integer underflow in `bt_mesh_sol_recv()` in the Bluetooth Mesh solicitation handling (`subsys/bluetooth/mesh/solicitation.c`) leads to an out-of-bounds write. When `CONFIG_BT_MESH_OD_PRIV_PROXY_SRV` is enabled, the function parses solicitation PDUs from raw BLE advertising payloads. The AD parsing loop reads an attacker-controlled length byte and computes `reported_len - 3` without checking that `reported_len` is at least 3. When the value is smaller, the signed subtraction yields a negative number that bypasses the length guard and is then implicitly converted to a very large `size_t`, advancing the buffer pointer far out of bounds so that subsequent reads dereference invalid memory. A nearby BLE device can trigger this with a non-connectable advertisement carrying a UUID16 AD structure and a crafted length byte, with no pairing or prior association required, potentially leading to denial of service or arbitrary code execution.

- Zephyr project bug tracker GHSA-4pm9-4v7f-x6gr

This has been fixed in main for v4.4.0

- PR 105585 fix for main
- PR 108334 fix for 4.3
- PR 108333 fix for 3.7

CVE 2026-5590

net: ip/tcp: Null pointer dereference can be triggered by a race condition

A race condition during TCP connection teardown can cause tcp_recv() to operate on a connection that has already been released. If tcp_conn_search() returns NULL while processing a SYN packet, a NULL pointer derived from stale context data is passed to tcp_backlog_is_full() and dereferenced without validation, leading to a crash.

- Zephyr project bug tracker GHSA-4vqm-pw24-g9jp

This has been fixed in main for v4.4.0

- PR 102110 fix for main

CVE 2026-8718

Out-of-bounds write in DTLS peer Connection ID getsockopt (`TLS_DTLS_PEER_CID_VALUE`) in Zephyr net sockets/TLS

`tls_opt_dtls_peer_connection_id_value_get()` in `subsys/net/lib/sockets/sockets_tls.c`, which handles `getsockopt(SOL_TLS, TLS_DTLS_PEER_CID_VALUE)`, passed the caller-supplied `optval` directly to `mbedtls_ssl_get_peer_cid()` without verifying the buffer was at least `MBEDTLS_SSL_CID_OUT_LEN_MAX` (default 32) bytes. `mbedtls_ssl_get_peer_cid()` copies the peer-negotiated DTLS Connection ID (length 1..`MBEDTLS_SSL_CID_OUT_LEN_MAX`) into that buffer without a destination-size parameter, so a caller-supplied `optlen` smaller than the CID causes a write of up to 31 bytes past the buffer end.

In `CONFIG_USERSPACE` builds the getsockopt syscall verifier (`z_vrfy_zsock_getsockopt`) bounce-buffers the user's `optval` into a kernel allocation of exactly `optlen` bytes (`k_usermode_alloc_from_copy -> z_thread_malloc`), so an unprivileged user thread that passes a small `optlen` on a connected DTLS socket with Connection ID enabled induces a kernel-heap buffer overflow, with the overflowing content being the remote peer's CID.

The defect requires `CONFIG_MBEDTLS_SSL_DTLS_CONNECTION_ID`, an established DTLS session with a negotiated peer CID, and (for the kernel-crossing case) `CONFIG_USERSPACE`. Introduced when the `TLS_DTLS_CID` option was added (v3.5.0).

The fix rejects callers whose `optlen` is below `MBEDTLS_SSL_CID_OUT_LEN_MAX` with -EINVAL.

- Zephyr project bug tracker GHSA-p3r6-mx6c-33gq

This has been fixed in main for v4.5.0

- PR 109244 fix for main
- PR 109624 fix for v4.4
- PR 113749 fix for v4.3
- PR 113750 fix for v3.7

CVE 2026-9263

Out-of-bounds read in Bluetooth Controller ISOAL framed RX reassembly leaks adjacent memory into host HCI ISO packets

The Zephyr Bluetooth controller ISO Adaptation Layer (`subsys/bluetooth/controller/ll_sw/isoal.c`) fails to validate the length field of a framed ISO PDU start segment. Per the Bluetooth specification a start segment (`sc=0`) always carries a 3-byte `time_offset`, so its segment-header `len` must be at least `PDU_ISO_SEG_TIMEOFFSET_SIZE` (3). `isoal_check_seg_header()` accepted start segments with `len < 3` as valid, and `isoal_rx_framed_consume()` then computed `length = seg_hdr->len - 3` in a `uint8_t`, underflowing to 253-255 when `len` is 0-2. That oversized length is passed to `isoal_rx_append_to_sdu()`, whose copy is clamped only against the destination SDU buffer size, not the source PDU length, so up to ~255 bytes of controller memory beyond the received PDU are copied (via `sink_sdu_write_hci()`/`net_buf_add_mem`) into an HCI

ISO data packet and delivered to the host. The PDU and its segment headers are entirely attacker-controlled and arrive over the air, reachable through both the CIS and BIS-sync HCI data paths (`hci_driver.c`) and the vendor data path (`ull_iso.c`), so a remote CIS peer or a broadcaster the device is synced to can trigger an out-of-bounds read causing information disclosure to the host and potential denial of service (faults or malformed oversized HCI ISO packets). The flaw affects all Zephyr releases since framed ISO reception was introduced in v3.0.0. The fix rejects `sc=0` segments with `len` < 3 in `isoal_check_seg_header()` and adds a guard before the subtraction in `isoal_rx_framed_consume()`.

- Zephyr project bug tracker GHSA-6gvp-pmh8-fjh2

This has been fixed in main for v4.5.0

- PR 109369 fix for main
- PR 109617 fix for v4.4
- PR 109619 fix for v4.3
- PR 109618 fix for v3.7

CVE 2026-10593

Remotely triggerable NULL-pointer dereference in Bluetooth LE Audio BAP unicast client QoS-state handling

The Zephyr Bluetooth LE Audio Basic Audio Profile (BAP) unicast client mishandles peer-supplied ASE state notifications. In `unicast_client_ep_qos_state()` (`subsys/bluetooth/audio/bap_unicast_client.c`), the handler writes attacker-controlled QoS fields (`interval`, `framing`, `phy`, `sdu`, `rtn`, `latency`, `pd`) through the `stream->qos` pointer with only a `stream != NULL` guard. `stream->qos` is `NULL` for any stream that has been codec-configured via `bt_bap_stream_config()` but not yet added to a unicast group (it is set only by `unicast_group_add_stream()`).

A malicious or buggy remote ASCS server, to which the local device is connected as a BAP unicast client, can send a GATT notification announcing the ASE has entered the QoS Configured state while the local endpoint is still in the Codec Configured state — a transition the dispatcher explicitly permits — during that window, causing a write through a NULL pointer and a crash (denial of service). The data written is itself remote-controlled.

The defect shipped in v4.3.0 and v4.4.0 (and earlier). The fix re-points all BAP QoS storage to the always-valid embedded `ep->qos` struct, eliminating the NULL dereference.

- Zephyr project bug tracker GHSA-22q8-m94g-2pwh

This has been fixed in main for v4.5.0

- PR 104887 fix for main
- PR 110779 fix for v4.4
- PR 110777 fix for v4.3

CVE 2026-10634

Use-after-free in Zephyr native TCP `net_tcp_foreach()` due to dropping `tcp_lock` during the callback

Zephyr's native TCP stack iterates the global connection list in `net_tcp_foreach()` (`subsys/net/ip/tcp.c`) using the `SYS_SLIST_FOR_EACH_CONTAINER_SAFE` macro, which caches a pointer to the next list node. Prior to this fix the function released `tcp_lock` while invoking the per-connection callback and re-acquired it afterwards. During that window a concurrent `tcp_conn_release()`, running on the dedicated TCP work-queue thread when a connection's reference count drops to zero (e.g. a remote peer closing or resetting the connection), can remove and `k_mem_slab_free()` the cached next connection. When the iterator advances it dereferences the freed (and possibly reallocated) slab memory — a use-after-free that can crash the system (denial of service) and, if

the slot has been reused, cause the callback to operate on an attacker-influenced object (potential information disclosure or further fault). `net_tcp_foreach()` is reached in production via the `net conn` network shell command and via `net_tcp_close_all_for_iface()` on interface-down; the freeing side is driven by ordinary TCP traffic. The fix moves the connection/context teardown in `tcp_conn_release()` inside the `tcp_lock` critical section and keeps `tcp_lock` held across the callback in `net_tcp_foreach()`. The defect was introduced with the modern (TCP2) stack in 2020 and affects releases up to and including v4.4.0.

- Zephyr project bug tracker GHSA-6c57-xfhw-j26x

This has been fixed in main for v4.5.0

- PR 106992 fix for main
- PR 107287 fix for v3.7
- PR 107288 fix for v4.3
- PR 107289 fix for v4.4

CVE 2026-10635

Dangling memory-domain pointer (use-after-free) in Xtensa MMU page-table code on memory-domain de-init

On Xtensa targets with `CONFIG_USERSPACE` and `CONFIG_XTENSA_MMU`, the page-table code (`arch/xtensa/core/ptables.c`) maintains a global list, `xtensa_domain_list`, of active memory domains using a list node embedded inside the caller-owned `struct k_mem_domain`. When a domain is destroyed via `k_mem_domain_deinit()` -> `arch_mem_domain_deinit()`, the page tables are torn down and `domain->arch.ptables` is set to `NULL`, but the domain's node was not removed from `xtensa_domain_list`. The freed/deinitialized domain therefore remained linked into the global list as a dangling pointer into caller-owned storage that may then be freed or reused.

Any subsequent `arch_mem_map()/arch_mem_unmap()` operation (widely invoked by kernel memory-mapping and demand-paging code) traverses the stale node and dereferences `domain->ptables`: at minimum a NULL pointer dereference causing a fatal MMU exception (denial of service), and if the `k_mem_domain` storage has been freed or reused, a use-after-free in which a stale/controlled `ptables` value is dereferenced and written through during the page-table walk (`l2_page_table_map` writes `l1_table[...]` and `l2_table[...]`, and `xtensa_mmu_compute_domain_regs` writes into the domain struct and the L1 table), yielding page-table memory corruption that can undermine userspace isolation.

The vulnerable path is reachable only from privileged kernel/supervisor code (`k_mem_domain_deinit` is not a syscall), not directly from unprivileged user threads or remotely. Affected: Zephyr v4.4.0 (the Xtensa memory-domain de-initialization feature was introduced in commit 3032b58f52d and first shipped in v4.4.0); fixed on `main` by adding `sys_slist_find_and_remove()` in `arch_mem_domain_deinit()`. The Xtensa MPU path is unaffected.

- Zephyr project bug tracker GHSA-39v7-cx8j-gq82

This has been fixed in main for v4.5.0

- PR 106923 fix for main
- PR 110758 fix for v4.4

CVE 2026-10636

Use-after-free in Zephyr IPv4 IGMP send path (`igmp_send`)

In Zephyr's IPv4 IGMP implementation, `igmp_send()` in `subsys/net/ip/igmp.c` read the network interface back out of the packet via `net_pkt_iface(pkt)` after the packet had been handed to `net_send_data()`. On the successful-send path the packet's last reference may already have

been released by the L2 driver or by the network stack's TX handling (synchronously in the default `NET_TC_TX_COUNT=0` immediate-transmit configuration), returning the `net_pkt` slab block to its free list. The subsequent `net_pkt_iface(pkt)` dereferences the freed packet, a use-after-free read; with `CONFIG_NET_STATISTICS_PER_INTERFACE` the resulting dangling interface pointer is further dereferenced for a statistics-counter write. The IGMP send path is reachable without authentication from inbound IPv4 IGMP membership queries addressed to 224.0.0.1 (`net_ipv4_igmp_input` -> `send_igmp_report`/`send_igmp_v3_report` -> `igmp_send`), as well as from local multicast join/leave/rejoin operations. Realistic impact is undefined behavior and potential denial of service (sporadic crash or stats corruption); a controllable write requires the asynchronous TX path plus a concurrent slab reuse. The flaw was introduced with IGMPv2 support and affects releases from v2.6.0 through v4.4.0. The fix caches the interface pointer before sending. Note the analogous IPv6 MLD path (`mld_send` in `subsys/net/ip/ipv6_mld.c`) retains the same unfixed pattern.

- Zephyr project bug tracker GHSA-fj6q-975v-65c9

This has been fixed in main for v4.5.0

- PR 107100 fix for main
- PR 110659 fix for v3.7
- PR 110658 fix for v4.3
- PR 107369 fix for v4.4

CVE 2026-10637

Use-after-free of `net_pkt` in IPv6 MLD send path triggerable by a link-local MLD Query

`subsys/net/ip/ipv6_mld.c:mld_send()` read the packet interface via `net_pkt_iface(pkt)` after `net_send_data(pkt)` returned successfully. Per the network stack's ownership contract (`include/zephyr/net/net_core.h`, and the explicit warning in `subsys/net/ip/net_core.c`:453-460 'do not use pkt after that call'), a successful send transfers ownership of the `net_pkt` and the L2 driver frees it (e.g. `ethernet_send()` unrefs the packet on success, `subsys/net/l2/ethernet/ethernet.c`:790), returning it to its `k_mem_slab`. The subsequent `net_pkt_iface(pkt)` is therefore a read of a freed object; the recovered interface pointer is then dereferenced and incremented by the per-interface statistics path (`net_stats.h` `UPDATE_STAT/SET_STAT`) when `CONFIG_NET_STATISTICS_PER_INTERFACE` is enabled. If the freed slot is concurrently reallocated, `pkt->iface` may read back as `NULL` (NULL-pointer dereference / crash) or as a stale/garbage pointer (stray increment write / memory corruption). The path is reachable remotely on the local link without authentication: `handle_mld_query()` (registered for `NET_ICMPV6_MLD_QUERY`) responds to a valid MLDv2 General Query (unspecified multicast address, hop limit 1) by calling `send_mld_report()` -> `mld_send()`. The result is a remotely triggerable denial of service of the networking stack, with a narrow possibility of memory corruption. The fix caches the interface in a local before sending and no longer touches the packet after `net_send_data()`. The IPv4/IGMP sibling (`igmp_send`) already used the corrected pattern.

- Zephyr project bug tracker GHSA-m23w-34pp-4h92

This has been fixed in main for v4.5.0

- PR 107100 fix for main
- PR 110659 fix for v3.7
- PR 110658 fix for v4.3
- PR 107369 fix for v4.4

CVE 2026-10638

Use-after-free in Zephyr ICMPv6 RX path when updating statistics after sending an echo reply or error

subsys/net/ip/icmpv6.c reads the network interface from a net_pkt after that packet has been handed to net_try_send_data(). In icmpv6_handle_echo_request() and net_icmpv6_send_error(), the post-send statistics update calls net_pkt_iface(reply)/net_pkt_iface(pkt) on the just-sent packet. The send path (net_try_send_data -> net_if_tx) unreferences and may free the packet back to its memory slab before returning — synchronously in the RX thread when no TX queue is configured (CONFIG_NET_TC_TX_COUNT == 0), and asynchronously the driver/L2 may already have freed it otherwise. net_pkt_iface() therefore dereferences a freed (and possibly reused) net_pkt; with CONFIG_NET_STATISTICS_PER_INTERFACE the stale iface pointer is further dereferenced and written through (iface->stats.icmp.sent++), turning the use-after-free read into a write through an attacker-influenceable pointer. The core stack already documents this hazard in net_core.c ("do not use pkt after that call") and caches iface before sending; the ICMPv6 callers did not. An unauthenticated remote attacker triggers the flaw simply by sending an ICMPv6 Echo Request (ping) or an IPv6 packet that elicits an ICMPv6 error (unknown next header, fragment reassembly timeout, destination unreachable), leading to denial of service via crash and potential memory corruption. Affected: Zephyr networking with CONFIG_NET_NATIVE_IPV6, roughly v4.2.0 through v4.4.0. The fix caches the interface pointer before sending and uses it for all statistics updates; the sibling commit 86e21665d46 fixes the identical bug in ICMPv4.

- Zephyr project bug tracker GHSA-m92g-94xv-wvw2

This has been fixed in main for v4.5.0

- PR 107100 fix for main
- PR 110659 fix for v3.7
- PR 110658 fix for v4.3
- PR 107369 fix for v4.4

CVE 2026-10639

Use-after-free reading net_pkt_iface() of a sent ICMPv4 echo-reply packet in icmpv4_handle_echo_request()

In Zephyr's native IPv4 stack, icmpv4_handle_echo_request() in subsys/net/ip/icmpv4.c builds an echo-reply packet (reply), hands it to net_try_send_data(), and then, on success, calls net_stats_update_icmp_sent(net_pkt_iface(reply)). net_try_send_data() transfers ownership of reply to the TX path (net_if_try_queue_tx -> net_if_tx -> L2/driver send, or the asynchronous net_if_tx_thread), which can unref it to refcount 0 and return the struct net_pkt to its slab (net_pkt_unref -> k_mem_slab_free) before the stats line runs. net_core.c documents this exact contract ('the pkt might contain garbage already ... do not use pkt after that call').

The post-send net_pkt_iface(reply) therefore reads reply->iface out of a freed (and possibly already reallocated) net_pkt, a use-after-free read; with CONFIG_NET_STATISTICS_PER_INTERFACE the stats macro additionally increments a counter through that value, i.e. a dereference/write through a stale or recycled-slot pointer.

The path is reached unauthenticated by any remote host that pings the device (net_icmpv4_input -> net_icmp_call_ipv4_handlers -> icmpv4_handle_echo_request) and is gated on CONFIG_NET_STATISTICS_ICMP. Impact is a probabilistic read of recycled packet memory plus a possible wild-pointer write under a timing race, leading most likely to corrupted interface statistics or a remotely triggerable crash (DoS).

The defect was introduced in 2019 (v1.14) and is present through v4.4.0. The companion change in net_icmpv4_send_error() is not a use-after-free because it reads net_pkt_iface(orig), the caller-owned received packet, which stays alive across the send. The fix caches the interface pointer from the live received packet before sending and uses it for the post-send stats updates.

- Zephyr project bug tracker GHSA-qhrf-w466-qmpw

This has been fixed in main for v4.5.0

- PR 107100 fix for main
- PR 110659 fix for v3.7
- PR 110658 fix for v4.3
- PR 107369 fix for v4.4

CVE 2026-10640

Use-after-free reading `net_pkt iface` after send in IPv6 Neighbor Discovery (`ipv6_nbr.c`)

Zephyr's IPv6 Neighbor Discovery send paths (`net_ipv6_send_na`, `net_ipv6_send_ns`, `net_ipv6_send_rs` in `subsys/net/ip/ipv6_nbr.c`) updated the per-interface ICMP- sent statistics by calling `net_pkt_iface(pkt)` after `net_send_data(pkt)` had already returned successfully. On the success path the network stack owns and releases the packet's reference (the L2/driver send unrefs it, e.g. `ethernet_send` -> `net_pkt_unref`), so for a freshly allocated packet with refcount 1 the `net_pkt` slab block can be freed before the statistics line runs (synchronously when no TX queue thread is configured, or via a concurrent TX thread otherwise).

The subsequent `net_pkt_iface(pkt)` reads `pkt->iface` from the freed slab block, and with `CONFIG_NET_STATISTICS_PER_INTERFACE` enabled that loaded pointer is dereferenced to increment `iface->stats.icmp.sent`, a use-after-free (CWE-416). If the slab block was reallocated in the meantime the read/increment targets unrelated or attacker-influenced memory, yielding corrupted statistics, a fault/crash (denial of service), or potential limited memory corruption.

The vulnerable Neighbor Advertisement path is reachable by any unauthenticated on-link node simply by sending ICMPv6 Neighbor Solicitations to a Zephyr node with native IPv6 enabled (`handle_ns_input` -> `net_ipv6_send_na`).

Affected from v3.3.0 through v4.4.0; the fix uses the already-available `iface` argument instead of touching the sent packet. Configurations without per-interface statistics dereference only a global counter and are not affected by the memory-safety aspect.

- Zephyr project bug tracker GHSA-r74c-mr4m-7g9g

This has been fixed in main for v4.5.0

- PR 107100 fix for main
- PR 110659 fix for v3.7
- PR 110658 fix for v4.3
- PR 107369 fix for v4.4

CVE 2026-10641

Out-of-bounds write in Bluetooth HFP Hands-Free CIND indicator parsing (cind_handle_values)

Zephyr's Bluetooth Classic Hands-Free Profile (HFP) Hands-Free role parser (subsys/bluetooth/host/classic/hfp_hf.c) contains an out-of-bounds write. During Service Level Connection setup the HF sends AT+CIND=? and parses the AG's +CIND: response in cind_handle(), which assigns a per-entry counter `index` and calls cind_handle_values() for each list element. cind_handle_values() then wrote `hf->ind_table[index] = i` without verifying that `index` is within the 20-element int8_t ind_table[] array of struct bt_hfp_hf. Because the parser places no cap on the number of +CIND: list entries, a remote Attendant Gateway (a malicious, compromised, or spoofed peer the device connects to over Bluetooth) can send a response with more than 20 recognized indicator entries and drive `index` arbitrarily large, writing a small attacker-positioned value past the array into adjacent struct fields (feature masks, SDP/version state, the calls[] array, work/atomic bookkeeping) and potentially beyond the static connection pool slot. This yields memory corruption and at least denial of service of the Bluetooth host, triggered by a single malformed AT response with no user interaction. The sibling consumer ag_indicator_handle_values() already performed the equivalent bounds check; this commit adds the same `index >= ARRAY_SIZE(hf->ind_table)` guard to close the gap. Affects builds

with CONFIG_BT_HFP_HF enabled; introduced with the original HFP HF CIND parser (~v1.7) and present through v4.4.0.

- Zephyr project bug tracker GHSA-wx5j-q6f2-59p3

This has been fixed in main for v4.5.0

- PR 107331 fix for main
- PR 110765 fix for v3.7
- PR 110764 fix for v4.3
- PR 110763 fix for v4.4

CVE 2026-10642

Unbounded TX busy-loop DoS in Zephyr PL011 UART driver under CTS hardware flow control

The Zephyr PL011 UART driver (`drivers/serial/uart_pl011.c`) contains an unbounded software loop in `pl011_irq_tx_enable()` that repeatedly invokes the interrupt-driven application callback while the TX interrupt mask bit (`PL011_IMSC_TXIM`) is set, to work around the controller's level-transition TX-interrupt behavior.

When CTS hardware flow control is enabled (devicetree `hw-flow-control` or runtime `UART_CFG_FLOW_CTRL_RTS_CTS`) and the wired serial peer de-asserts CTS, the controller stops draining the TX FIFO; `pl011_fifo_fill()` then returns 0 on every call while the application still has pending data and therefore never disables the TX interrupt. The loop condition never clears, so the thread that called `uart_irq_tx_enable()` (e.g. `h4_send()` in the Bluetooth HCI H4 driver) spins indefinitely, hanging the executing context and stalling the transport — a denial of service (CWE-835).

An attacker controlling the device attached to the UART's CTS line can trigger the hang by withholding CTS during transmission. Because that peer is the device wired to the UART — which may be a removable or external module (e.g. an off-board Bluetooth controller on the HCI H4 link) rather than a permanently-bonded on-PCB part — the attack vector is scored Adjacent (AV:A) rather than Physical; the security subcommittee should confirm the vector against the specific deployment. Impact is availability only; there is no memory-safety, confidentiality, or integrity consequence.

The vulnerable loop was introduced in commit b783bc8448ef (Feb 2025) and shipped in releases v4.1.0 through v4.4.0. The fix breaks out of the loop when CTS is blocking and arms the CTS modem-status interrupt to resume transmission when CTS re-asserts.

- Zephyr project bug tracker GHSA-3fgh-73jh-2q5j

This has been fixed in main for v4.5.0

- PR 103684 fix for main
- PR 110768 fix for v4.4
- PR 110767 fix for v4.3

CVE 2026-10643

Out-of-bounds heap write in Zephyr `recvmsg()` ancillary-data path (`insert_pktinfo` undersizes the control-buffer capacity check)

Zephyr's IP socket `recvmsg()` implementation (`subsys/net/lib/sockets/sockets_inet.c`, `insert_pktinfo()`) validated the user-supplied ancillary (`msg_control`) buffer using only the payload length (`msg->msg_controllen < pktinfo_len`) before writing a full control message consisting of an aligned cmsg header plus the payload. Because the check omitted the cmsg header size, a control buffer whose length falls in the under-checked window (e.g. 16-27 bytes for IPv4 `IP_PKTINFO` on a 64-bit target, where a single element actually occupies 28 bytes) passes the guard

yet causes a fixed-size out-of-bounds write of up to one cmsg header (~12 bytes) past the end of the buffer.

Under `CONFIG_USERSPACE` the `recvmsg` verifier allocates a kernel-heap copy of the control buffer sized to `msg_controllen` and runs the implementation against it, so the overflow corrupts kernel heap memory and is triggerable from an unprivileged userspace thread; in supervisor mode it corrupts the caller's buffer.

The path is reachable on a UDP/IP socket with `IP_PKTINFO/IPV6_RECVPKTINFO` (or hoplimit/timestamping) enabled when the application calls `recvmsg()` with an undersized control buffer and a datagram is received; part of the overwritten bytes (the destination IP in `ipi_addr`) is influenced by the received packet.

The fix makes the capacity check use `NET_CMSG_SPACE(pktinfo_len)` (aligned header + aligned data) and returns `-ENOMEM` when the buffer is too small. Affected: v3.6.0 through v4.4.0.

- Zephyr project bug tracker GHSA-pvf7-7mrp-35w7

This has been fixed in main for v4.5.0

- PR 106464 fix for main
- PR 110668 fix for v4.4
- PR 110669 fix for v4.3
- PR 110670 fix for v3.7

CVE 2026-10644

Out-of-bounds write in Microchip SERCOM-G1 (PIC32CM-JH) async UART RX with 1-byte buffer

The Microchip SERCOM-G1 UART driver (`drivers/serial/uart_mchp_sercom_g1.c`), used by the PIC32CM-JH SoC family, contains an out-of-bounds write in its asynchronous (DMA) receive path. When `uart_rx_enable()` is invoked with a one-byte receive buffer (`len == 1`) and `CONFIG_UART_MCHP_ASYNC` is enabled, the RX-complete ISR starts a single-beat DMA transfer while a received byte is already pending in the SERCOM DATA register. On this SoC the peripheral-triggered DMA start sequencing then writes one byte past the end of the caller-supplied buffer (CWE-787).

The overflowed byte's value is the UART RX data supplied by the connected serial peer (adjacent attacker), while its size and location are fixed at one byte immediately after the buffer.

Exploitation requires the async UART config (not enabled by default on the in-tree PIC32CM-JH boards) and a consumer that enables RX with a one-byte buffer; impact is limited single-byte memory corruption adjacent to the RX buffer (possible crash / denial of service).

The defect shipped in v4.4.0. The fix reads the first byte with the CPU and, for one-byte buffers, performs no DMA at all; for larger buffers it sizes the DMA for the remaining `len-1` bytes.

- Zephyr project bug tracker GHSA-xv2x-56j7-6wc3

This has been fixed in main for v4.5.0

- PR 107400 fix for main
- PR 110750 fix for v4.4

CVE 2026-10645

Out-of-bounds read in Zephyr ext2 directory entry traversal from a crafted filesystem image

The Zephyr ext2 filesystem driver (`subsys/fs/ext2`) trusted the on-disk directory entry fields `de_rec_len` and `de_name_len` when walking a directory block. `ext2_fetch_direntry()` guarded only with `de_name_len > EXT2_MAX_FILE_NAME`, but `de_name_len` is a `uint8_t` and `EXT2_MAX_FILE_NAME` is 255, so the check is always false; the function then `memcpy`'d up to 255 name bytes and the lookup/readdir paths advanced traversal by an unvalidated `de_rec_len`.

Each directory block is read into a `block_size`-sized slab buffer, and `block_off` can be driven near the block end by preceding entries' `rec_len`, so the 8-byte header read and the subsequent name `memcpy` can read up to ~263 bytes past the end of the block buffer into adjacent heap/slab memory. On the readdir path those bytes are returned to the caller in `fs_dirent.name`, leaking adjacent kernel heap memory; a `de_rec_len` of 0 also causes a zero-progress infinite loop (denial of service), and the unlink path's `memmove(de, next, next_reclen)` over unvalidated records is an additional OOB read/write source. The defect is reached by any path-based operation (open, stat, unlink, rename, mkdir) or directory listing on a mounted ext2 volume, so a crafted or corrupted ext2 image on attacker-supplied storage (SD card, USB mass storage, or otherwise mounted image) triggers it. Affected: Zephyr ext2 from its introduction in v3.5.0 through v4.4.0. The fix validates `rec_len` and `name_len` in the parser and rejects entries whose header does not fit the remaining block or whose `rec_len` crosses the block boundary in every traversal caller.

- Zephyr project bug tracker GHSA-hwrh-9h3x-vccm

This has been fixed in main for v4.5.0

- PR 108226 fix for main
- PR 110031 fix for v4.4
- PR 110033 fix for v4.3
- PR 110030 fix for v3.7

CVE 2026-10646

Use-after-return in `zsock_getaddrinfo()` when a timed-out DNS query is retried without cancellation

Zephyr's BSD-sockets `getaddrinfo()` implementation (`subsys/net/lib/sockets/getaddrinfo.c`) passes a pointer to a stack-allocated state object (`struct getaddrinfo_state ai_state`) as the `user_data` of an asynchronous DNS resolver query. The socket layer waits on a semaphore with a timeout deliberately set slightly longer than the resolver's own per-query timeout. When that semaphore wait nonetheless times out (`-EAGAIN`) - which can occur when the resolver's timeout work is delayed by workqueue contention, or in the documented multi-retry configuration where `CONFIG_NET_SOCKETS_DNS_TIMEOUT` exceeds `CONFIG_NET_SOCKETS_DNS_BACKOFF_INTERVAL` - the pre-fix code retries the query (`goto again`) without cancelling the previous one and without resetting the semaphore.

The previous query slot remains active in the resolver with its callback and the stack pointer as `user_data`, and `ai_state->dns_id` is overwritten so the stale query can no longer be cancelled. A subsequent DNS response delivered over UDP and matched by its 16-bit transaction id (in `dispatcher_cb()`/`dns_read()`), or the resolver's own delayed query-timeout work, then invokes `dns_resolve_cb()` against the now out-of-scope stack frame, writing through the stale pointer (`state->status`, `state->idx`, `state->ai_arr[]`, and `k_sem_give()`).

Because the triggering response is network-delivered and its 16-bit id is spoofable/replayable by an on- or off-path attacker, this is a network-influenceable use-after-return that can corrupt reused stack memory, leading to crashes/denial of service or memory corruption.

The fix cancels the timed-out query by name and type before retrying and resets the local semaphore, eliminating the stale callback path. Affected: Zephyr v4.0.0 through v4.4.0.

- Zephyr project bug tracker GHSA-h752-vhmf-29w6

This has been fixed in main for v4.5.0

- PR 107609 fix for main
- PR 110774 fix for v4.4
- PR 110773 fix for v4.3

CVE 2026-10647

Deadlock denial of service in USB CDC-NCM device class on TX enqueue failure

The USB CDC-NCM device class (`subsys/usb/device_next/class/usbd_cdc_ncm.c`) ignores the return value of `usbd_ep_enqueue()` in its ethernet transmit callback `cdc_ncm_send()`. When the enqueue fails, the function still calls `k_sem_take(&data->sync_sem, K_FOREVER)`, blocking on a completion semaphore that is only ever signaled from the bulk-IN transfer-completion callback. Because nothing was enqueued, that callback never fires and the calling thread — a shared network traffic-class TX thread — deadlocks permanently while holding the interface TX lock, halting transmission until reboot (and leaking the transmit buffer).

The enqueue fails under conditions controlled by the attached USB host: `usbd_ep_enqueue()` returns `-EPERM` whenever the bus is suspended (a standard, persistent host operation), and the underlying `udc_ep_enqueue()` returns `-EPERM/-ENODEV` on disconnect, bus reset, or endpoint disable. The `cdc_ncm_send()` guard only checks the `DATA_IFACE_ENABLED` and `IFACE_UP` flags, not the suspended state, so a packet transmitted while the host holds the bus suspended reaches the failing enqueue and deadlocks the TX path.

The realistic trigger is a bus suspend that occurs while the exported network interface is active and has traffic to send — host sleep, USB selective/auto-suspend, or hub power management — after which any device-originated packet deadlocks the path, recoverable only by reboot. The impact is a persistent loss of the virtual network connection between the host's NCM interface and the Zephyr device; because the deadlocked thread is a shared traffic-class TX thread, egress on other network interfaces can stall as well. There is no memory corruption or information disclosure.

The defect was introduced with the CDC-NCM driver and shipped in releases through v4.4.0; it is fixed by checking the `usbd_ep_enqueue()` return value and freeing the buffer before the blocking wait.

- Zephyr project bug tracker GHSA-xcf7-r86m-5q9f

This has been fixed in main for v4.5.0

- PR 107126 fix for main
- PR 110652 fix for v4.4
- PR 110653 fix for v4.3

CVE 2026-10648

NULL-pointer dereference in MCUmgr serial/console SMP transport on buffer-pool exhaustion

`mcumgr_serial_process_frag()` in `subsys/mgmt/mcumgr/transport/src/serial_util.c` calls `net_buf_reset()` on the result of `smp_packet_alloc()` before checking it for `NULL`. `smp_packet_alloc()` uses `net_buf_alloc(K_NO_WAIT)` against the shared MCUmgr packet pool (`CONFIG_MCUMGR_TRANSPORT_NETBUF_COUNT`, default 4), which returns `NULL` when the pool is exhausted. In default builds the `__ASSERT_NO_MSG` in `net_buf_reset` is a no-op, so `net_buf_simple_reset` writes through the `NULL` pointer (`buf->len = 0; buf->data = buf->__buf`), causing a fault/crash.

The fragment data reaches this code from attacker-controlled bytes on the MCUmgr serial/UART/shell-console transports (`smp_uart.c`, `smp_raw_uart.c`, `smp_shell.c`), and a fresh buffer is allocated at the start of essentially every new packet. An attacker on the serial/console link can flood the transport to drive the 4-entry buffer pool to exhaustion and induce the `NULL` dereference, crashing the device (denial of service).

The defect was introduced after the original MCUmgr rework and shipped in Zephyr v4.4.0. The fix moves the `NULL` check ahead of `net_buf_reset`.

- Zephyr project bug tracker GHSA-j64f-h3ww-f32c

This has been fixed in main for v4.5.0

- PR 107812 fix for main
- PR 108026 fix for v4.4

CVE 2026-10651

Out-of-bounds read in Bluetooth Classic SDP attribute parsing (`bt_sdp_parse_attribute`)

`bt_sdp_parse_attribute()` in `subsys/bluetooth/host/classic/sdp.c` validated only that the SDP record buffer held the type-marker byte plus the 2-byte attribute ID (a check of `buf->len < 3`) but then read a fourth byte, the data-element descriptor (`type`), via `net_buf_simple_pull_u8()`. Because `net_buf_simple_pull_u8()` dereferences `buf->data[0]` before its only bounds guard (an `__ASSERT_NO_MSG` that compiles out when `CONFIG_ASSERT` is disabled, the production default), a record of exactly three bytes (0x09 followed by a 2-byte attribute ID) causes a one-byte read past the end of the logical buffer. The parser is reachable from inbound, remote-controlled data: a Bluetooth BR/EDR peer acting as an SDP server returns discovery-response records that are stored verbatim in the client receive buffer and parsed via the public `bt_sdp_get_attr()`/`bt_sdp_has_attr()`/`bt_sdp_record_parse()` helpers. The over-read is bounded to a single byte that is used only as an internal length selector and is never leaked to the attacker; subsequent length checks then reject the malformed record. Realistic impact is therefore limited to an edge-case denial of service (a fault only if the record ends exactly at a mapped-memory boundary, or a deterministic assert panic when `CONFIG_ASSERT=y`). Affects Zephyr v4.3.0 and v4.4.0; fixed by adding `sizeof(type)` to the length check.

- Zephyr project bug tracker GHSA-p93g-3r68-cj53

This has been fixed in main for v4.5.0

- PR 107325 fix for main
- PR 110850 fix for v4.4
- PR 110851 fix for v4.3

CVE 2026-10652

Out-of-bounds read in Zephyr DNS resolver TXT/SRV record parsing (unvalidated `rdlength`)

Zephyr's DNS resolver (`subsys/net/lib/dns`) parses resource records from DNS responses in `dns_unpack_answer()`, which validated only the fixed RR header (type, class, TTL, `rdlength`) and accepted any attacker-declared `rdlength`, including one extending past the end of the received datagram. The TXT and SRV consumers in `dns_validate_record()` (`resolve.c`) then read up to `rdlength` bytes (clamped only to a record-type maximum such as `DNS_MAX_TEXT_SIZE`, default 64, not to the packet) from the receive buffer via `memcpy` without their own bounds check, and pass the result to the application's resolve callback. A malicious or spoofed DNS server, an on-path attacker forging UDP DNS replies, or (with mDNS/LLMNR enabled) any LAN node can craft a truncated TXT or SRV response that causes an out-of-bounds read of adjacent receive-pool memory; the disclosed stale bytes (residual contents of prior DNS packets / uninitialized pool memory) are returned to the application as TXT/SRV record contents, an information leak, and may in some configurations cross the allocation boundary and fault, causing a denial of service. The read is bounded (~64 bytes for TXT, ~6 for SRV) and read-only (no write). The fix rejects any record whose declared rdata extends past `dns_msg->msg_size` at the single chokepoint in `dns_unpack_answer()`. Affected: v4.3.0 and v4.4.0.

- Zephyr project bug tracker GHSA-3jxq-xx8g-q8j2

This has been fixed in main for v4.5.0

- PR 107977 fix for main
- PR 108844 fix for v4.4
- PR 108843 fix for v4.3
- PR 108845 fix for v3.7

CVE 2026-10653

Non-atomic net_buf reference counts cause double-free / free-list corruption under concurrent unref

The Zephyr net_buf library (lib/net_buf/buf.c) manipulated both of its reference counts – the per-header buf->ref and the per-data-block ref_count at the start of each variable/heap data allocation – with plain non-atomic C operators (buf->ref++, if (--buf->ref > 0), if (--(*ref_count))).

The API is documented as self-synchronizing: callers may share one buffer across threads (e.g. via k_fifo) and each holder independently calls net_buf_unref() with no surrounding lock. Under true concurrency (SMP, or single-core preemption between the non-atomic load and store while another context unrefs the same buffer), two holders can both observe the same prior reference value and both conclude they are the last reference.

For heap/variable-data pools (mem_pool_data_unref/heap_data_unref, used by zbus message subscribers, the IP stack RX/TX buffers when CONFIG_NET_BUF_FIXED_DATA_SIZE=n, capture, wireguard, ISO-TP and usbip) this produces a double k_heap_free()/k_free() of the same block – heap-metadata corruption and a use-after-free on the heap-hardening poison pattern.

For the per-header refcount the buffer is returned to the pool free LIFO twice for any pool type (including fixed-data pools used by Bluetooth and networking), corrupting the free list so a later allocation hands the same buffer to two owners.

The fix converts both refcounts to atomic_inc/atomic_dec (overlaying buf->ref in an atomic_t-sized union and changing the data-block refcount from uint8_t to atomic_t).

Impact is gated on genuine concurrency and on an application architecture that shares one buffer among multiple independent unref'ers; the trigger is a refcount/timing race rather than packet content, so an external attacker has at most weak indirect influence over the race window. Affects all Zephyr releases through v4.4.0.

- Zephyr project bug tracker GHSA-284j-5jm9-55hh

This has been fixed in main for v4.5.0

- PR 108065 fix for main
- PR 110853 fix for v4.4
- PR 110852 fix for v4.3

CVE 2026-10654

RFCOMM session-disconnect race leaks session/L2CAP and denies further RFCOMM service in Zephyr Bluetooth Classic

A race condition in the Zephyr Bluetooth Classic RFCOMM host stack (subsys/bluetooth/host/classic/rfcomm.c) mishandles a simultaneous bidirectional session disconnect. When the local device has initiated a session teardown (state BT_RFCOMM_STATE_DISCONNECTING, DISC sent, RTX timer armed) and the connected peer concurrently sends its own DISC frame for dlci 0, rfcomm_handle_disc() invokes rfcomm_session_disconnected(), which unconditionally forced the session to BT_RFCOMM_STATE_DISCONNECTED without ever calling bt_l2cap_chan_disconnect().

Because the recovery timer was also cancelled and a later UA is ignored in the DISCONNECTED state, the session becomes permanently wedged: the underlying L2CAP channel is never released and the session slot in the fixed bt_rfcomm_pool[CONFIG_BT_MAX_CONN] array is never reclaimed (its conn pointer stays set).

Subsequent bt_rfcomm_dlc_connect() calls on that connection fail with -EINVAL due to the invalid session state, so RFCOMM service is denied for that peer, and repeated occurrences can exhaust the session pool. The DISC frame is peer-controlled over the air, but exploitation requires the peer's DISC to collide with a local-initiated disconnect (a high-complexity timing race).

Impact is availability/resource-leak only; there is no memory-safety, confidentiality, or integrity consequence. The defect shipped in released versions (present in v4.4.0 and earlier).

The fix only transitions to DISCONNECTED when the session is not already in DISCONNECTING, preserving the proper L2CAP teardown path.

- Zephyr project bug tracker GHSA-4m37-wp5x-hq4h

This has been fixed in main for v4.5.0

- PR 108089 fix for main
- PR 110865 fix for v4.4
- PR 110864 fix for v4.3
- PR 110863 fix for v3.7

CVE 2026-10655

Use-after-free race in SNTP async client when closing the socket while the socket service is still polling it

The asynchronous SNTP client in Zephyr (`subsys/net/lib/sntp/sntp.c`, `sntp_close_async`) closed the UDP socket file descriptor directly from the calling thread immediately after detaching it from the network socket service, without synchronizing with the socket-service poll thread.

The socket service thread polls each socket via `zvfs_poll`, which (in `zsock_poll_prepare_ctx`) registers a `k_poll_event` pointing into the socket's `net_context` (`&ctx->recv_q`) and then blocks in `k_poll` without holding a reference or lock. `net_context` objects are allocated from a fixed pool (`contexts[CONFIG_NET_MAX_CONTEXTS]`) and reused after close.

When `sntp_close_async` is invoked from a different thread than the poll thread (in the in-tree consumer `subsys/net/lib/config/init_clock_sntp.c`, the SNTP timeout handler runs on the system workqueue while the socket service thread is blocked in poll on the same fd), the close frees and may reuse the `net_context` while the poll thread still has a poller node linked into the freed object, resulting in a use-after-free / object confusion of kernel poll structures.

The SNTP timeout path is the normal no-response failure mode, so a network peer or off-path attacker who drops or delays the SNTP/NTP response can drive the racing close repeatedly (and periodically with `NET_CONFIG_SNTP_INIT_RESYNC`). The most likely consequence is a crash of the networking thread (denial of service), with potential memory corruption when the freed context slot is reallocated.

The fix defers the close to the socket service thread itself via `net_socket_service_close` (`NET_SOCKET_SERVICE_CLOSE_SOCKETS`), so the same thread that polls performs the close, eliminating the race. Affected releases: v4.2.0 through v4.4.0.

- Zephyr project bug tracker GHSA-34wr-cg29-c4mw

This has been fixed in main for v4.5.0

- PR 108180 fix for main
- PR 110860 fix for v4.4
- PR 110858 fix for v4.3

CVE 2026-10656

NULL-pointer dereference DoS in MAX32 USB device controller transfer-completion handlers

The MAX32xxx USB device controller driver (`drivers/usb/udc/udc_max32.c`, compatible `adi_max32_usbhs`) dereferenced an endpoint buffer in its OUT and IN transfer-completion handlers without checking it for `NULL`. `udc_event_xfer_out_done()` called `net_buf_add(buf, ep_request->actlen)` immediately after `buf = udc_buf_get(ep_cfg)`, where `udc_buf_get()` returns `NULL` when the endpoint FIFO is empty.

A transfer-completion event is queued from interrupt context and processed asynchronously by the driver thread; between queuing and processing, the endpoint FIFO can be drained by host-controlled control flow — in particular `udc_setup_received()` drains the EP0 OUT/IN FIFOs whenever a new SETUP packet arrives, and dequeue/disable/purge paths drain it likewise.

A USB host that aborts an in-flight EP0 control transfer with a new SETUP packet (legal USB behavior) can therefore cause a stale `XFER_OUT_DONE` event to be processed against an empty FIFO, producing `net_buf_add(NULL, ...)`, a near-NULL pointer dereference that faults and crashes the device. No authentication is required; the attacker is the USB host the device is connected to (physical bus access). Impact is denial of service (device crash).

The defect was introduced when the MAX32 UDC driver was added and shipped in Zephyr v4.4.0. The fix adds NULL-buffer checks that return early with `UDC_EVT_ERROR`/-ENOBUFS in both the OUT-done and IN-done handlers.

- Zephyr project bug tracker GHSA-58p9-6mjq-rf2m

This has been fixed in main for v4.5.0

- PR 108447 fix for main
- PR 109517 fix for v4.4

CVE 2026-10657

Out-of-bounds read in Zephyr DNS resolver mDNS suffix check (memcmp past string NUL)

Zephyr's DNS resolver detects mDNS (.local) queries in `dns_resolve_name_internal()` (`subsys/net/lib/dns/resolve.c`) with `memcmp(strrchr(query, '.'), ".local", 7)`, which always reads a fixed 7 bytes from the suffix pointer. When the resolved hostname's final label is shorter than 7 bytes (e.g. names ending in .org, .com, .net, .io, or a trailing dot), the comparison reads 1-2 bytes past the string's NUL terminator.

The hostname (`query`) is the caller-supplied name passed through the standard `getaddrinfo()`/`dns_get_addr_info()`/`dns_resolve_name()` path and is influenceable by operators or remote inputs (server names from configuration, parsed URLs, or app-facing interfaces).

On a tightly-sized buffer with no slack (for example a userspace `getaddrinfo` call where the hostname is copied with `k_usermode_string_alloc_copy` to exactly `strlen+1` bytes), the over-read crosses the allocation boundary; if that boundary is unmapped (guard page, memory-domain boundary under MPU, or an address sanitizer) the over-read faults, causing a denial of service. The over-read bytes are never returned, so there is no information disclosure.

The flaw is compiled only when `CONFIG_MDNS_RESOLVER` is enabled, exists since v1.10.0, and is fixed by replacing the fixed-length `memcmp` with a NUL-safe `strcmp(ptr, ".local")`.

- Zephyr project bug tracker GHSA-76jh-3j5f-9vq4

This has been fixed in main for v4.5.0

- PR 108372 fix for main
- PR 110870 fix for v4.4
- PR 110867 fix for v4.3
- PR 110868 fix for v3.7

CVE 2026-10658

Out-of-bounds access in Bluetooth ISO receive (`bt_iso_recv`) due to missing SDU-header length validation

`bt_iso_recv()` in `subsys/bluetooth/host/iso.c` pulled the ISO SDU header (4 bytes) or, when the timestamp flag is set, the timestamped SDU header (8 bytes) from the inbound HCI ISO Data buffer via `net_buf_pull_mem()` without first checking `buf->len`. The upstream

hci_iso() handler enforces buf->len == the controller-declared ISO Data_Load length, so a malicious or buggy controller / adjacent BLE peer on an established CIS/BIS can present a first-fragment (BT_ISO_START) or single (BT_ISO_SINGLE) PDU shorter than the SDU header. Because net_buf_simple_pull_mem only guards length with __ASSERT_NO_MSG (compiled out when CONFIG_ASSERT is disabled, the production default), the pull underflows buf->len (uint16_t, e.g. 0 - 8 = 0xFFF8) and advances buf->data past valid data: the subsequent reads of hdr->slen and hdr->sn are out-of-bounds reads of adjacent pool memory. For the multi-fragment (START) case the corrupted buffer is retained as iso->rx, and a following CONT/END fragment's net_buf_tailroom() guard underflows to a near-SIZE_MAX value, defeating the bounds check and causing net_buf_add_mem() to memcpy attacker-supplied fragment data far past the RX pool buffer (out-of-bounds write). The flaw affects ISO receive builds (CONFIG_BT_ISO_RX, selected by the default-off LE Audio options BT_ISO_PERIPHERAL/BT_ISO_CENTRAL/BT_ISO_SYNC_RECEIVER) and has existed since the ISO subsystem was introduced (v2.6.0) through v4.4.0. The fix adds explicit buf->len < sizeof(*ts_hdr) and buf->len < sizeof(*hdr) checks that drop the buffer before pulling.

- Zephyr project bug tracker GHSA-26g8-rmpf-j6cw

This has been fixed in main for v4.5.0

- PR 108603 fix for main
- PR 111024 fix for v4.4
- PR 110959 fix for v4.3
- PR 110958 fix for v3.7

CVE 2026-10659

NULL pointer dereference in Zephyr Dhara FTL disk driver on flash read error during journal resume

The Dhara flash translation layer disk driver (drivers/disk/ftl_dhara.c) implemented the dhara_nand_* callbacks so that, on a flash error, the error code was written unconditionally through the caller-supplied dhara_error_t *err pointer (e.g. *err = DHARA_E_ECC in dhara_nand_read, and similar in dhara_nand_erase/prog/copy).

The upstream Dhara library calls these callbacks with err == NULL along its journal-resume binary search: find_last_checkblock() invokes find_checkblock(j, mid, &found, NULL), which forwards the NULL pointer into dhara_nand_read(). This path runs during disk_ftl_access_init() -> dhara_map_resume() whenever the FTL disk is mounted/initialised.

If a flash read error (uncorrectable ECC, bad block, controller error) occurs on one of the probed checkpoint pages, the driver dereferences and writes to NULL, faulting the kernel (denial of service). The trigger is conditioned on the NAND medium content/health, which can be influenced by media wear, induced faults, or a corrupted/crafted on-flash image.

The fix routes all error assignments through the library's NULL-safe dhara_set_error() helper. Affects Zephyr v4.4.0, where the driver was introduced.

- Zephyr project bug tracker GHSA-q28v-3729-f82g

This has been fixed in main for v4.5.0

- PR 108594 fix for main
- PR 110955 fix for v4.4

CVE 2026-10660

Shared reassembly buffer in Bluetooth BAP Broadcast Assistant enables cross-connection memory corruption

The Bluetooth BAP Broadcast Assistant GATT client in subsys/bluetooth/audio/bap_broadcast_assistant.c reassembled remote Broadcast Receive State data into a single file-static net_buf_simple (att_buf, BT_ATT_MAX_ATTRIBUTE_LEN = 512 bytes) shared by all connection instances, while the BUSY flag, long-read handle, and reset/offset state were per-connection.

When the device acts as a Broadcast Assistant connected to multiple Scan Delegator peripherals, notification and long-read callbacks from different connections interleave on the shared buffer: the append in notify_handler (net_buf_simple_add_mem at the not-busy branch) performs no tailroom check, so receive-state notifications from two or more delegators accumulate on the same 512-byte buffer and, with a sufficiently large configured ATT MTU (BT_L2CAP_TX_MTU up to 2000) and two-to-three concurrent connections, write past the buffer into adjacent .bss (net_buf_simple_add only asserts in debug builds).

Even below the overflow threshold, one connection's net_buf_simple_reset zeroes the shared length while another connection's reassembly and GATT read offset are in flight, mixing one peer's data into another's parse. A malicious or compromised Scan Delegator (or two colluding peers) over BLE can trigger this, causing out-of-bounds writes (memory corruption / denial of service) and cross-connection data corruption.

The fix moves the buffer into the per-connection instance struct so each connection reassembles into its own buffer. Affects Zephyr releases shipping the Broadcast Assistant with the shared buffer, including v4.4.0 and earlier.

- Zephyr project bug tracker GHSA-73c7-3rh7-v5p9

This has been fixed in main for v4.5.0

- PR 107563 fix for main
- PR 111066 fix for v4.4
- PR 111065 fix for v4.3
- PR 111182 fix for v3.7

CVE 2026-10663

Use-after-free / double-free of the root USB device in the experimental USB host stack

In Zephyr's experimental USB host stack (CONFIG_USB_HOST_STACK), usbh_device_disconnect() (subsys/usb/host/usbh_device.c) freed the root usb_device slab object without clearing the cached pointer ctx->root. The bus removal handler dev_removed_handler() (subsys/usb/host/usbh_core.c) decides what to tear down solely from ctx->root, checking only that it is non-NULL.

Because UHC controller drivers (e.g. uhc_max3421e, uhc_mcux_common) synthesize UHC_EVT_DEV_REMOVED directly from physical bus line state with no debounce or state guard, an attacker with physical USB access (or a rogue device that bounces its connection) can deliver a second device-removed event after a root device disconnect. The handler then re-enters usbh_device_disconnect() with the dangling pointer, locking a mutex inside the freed object (use-after-free), removing the freed node from the device list, and calling k_mem_slab_free() on the already-freed block (double-free). If the slab block has been reissued to a newly attached device in between, this corrupts a live object.

Impact is denial of service (crash) and memory corruption; the attack vector is physical/local. The flaw was introduced in v4.4.0 by the connect/disconnect refactor and is fixed by clearing ctx->root in usbh_device_disconnect() before freeing.

- Zephyr project bug tracker GHSA-26q8-xjq3-f5p6

This has been fixed in main for v4.5.0

- PR 108796 fix for main
- PR 111021 fix for v4.4

CVE 2026-10664

Out-of-bounds write in nRF70 Wi-Fi driver power-save event handler (unbounded TWT flow count)

The nRF70 Wi-Fi driver's power-save event handler `nrf_wifi_event_proc_get_power_save_info()` in `drivers/wifi/nrf_wifi/src/wifi_mgmt.c` copied TWT (Target Wake Time) flow entries from an `nrf_wifi_umac_event_power_save_info` event into the fixed-size `twt_flows[WIFI_MAX_TWT_FLOWS]` (8-element) array of a caller-supplied `struct wifi_ps_config`, looping over event-provided `num_twt_flows` without validating it against `WIFI_MAX_TWT_FLOWS` or checking `event_len`. When `num_twt_flows` exceeds 8, the handler writes past the destination array (which is typically on the caller's stack, e.g. the `wifi ps` shell command) – an out-of-bounds write of ~40-byte TWT entries – and reads `twt_flow_info[i]` past the event buffer. The event is delivered by the nRF70 co-processor firmware in response to a host-initiated power-save GET, so reaching the overflow requires the firmware to emit a malformed or out-of-range event; the trust boundary is host-to-trusted-coprocessor rather than a direct remote-AP write, with over-the-air influence on the flow count being indirect and bounded by the 3-bit TWT flow-id space. Affected: builds with `CONFIG_NRF70_STA_MODE` on releases through v4.4.0. The fix rejects events with `num_twt_flows > WIFI_MAX_TWT_FLOWS` or with `event_len` shorter than the claimed entries, and adds a NULL check on the caller buffer.

- Zephyr project bug tracker GHSA-3r6j-pm38-r43m

This has been fixed in main for v4.5.0

- PR 108849 fix for main
- PR 109067 fix for v4.4
- PR 109068 fix for v4.3

CVE 2026-10665

Heap buffer overflow on WireGuard receive path via unbounded incoming packet length

In Zephyr's WireGuard subsystem (`subsys/net/lib/wireguard`), `wg_process_data_message()` in `wg_crypto.c` linearizes an inbound transport-data payload into a fixed pool buffer of `CONFIG_WIREGUARD_BUF_LEN` bytes before decryption. The call `net_buf_linearize(buf->data, data_len, pkt->buffer, ..., data_len)` passed the attacker-derived `data_len` as both the destination capacity and the copy length, defeating the function's internal `len = min(len, dst_len)` bound. `data_len` is derived from the received UDP datagram length and is only lower-bounded by `wg_ctrl_recv()` (no upper bound). When `data_len` exceeds `CONFIG_WIREGUARD_BUF_LEN` — e.g. when the buffer length is lowered below the link MTU, on links with MTU above the buffer size, or via reassembled IPv4/IPv6 fragments that exceed it — the underlying `memcpy` writes past the end of the pool buffer, an out-of-bounds write (CWE-787). The overflow occurs before the Poly1305 authentication check, so it requires only a valid receiver session index rather than a valid authenticator, and is reachable by a malicious or compromised peer (or an on-path attacker driving an established session) over the network, yielding remote memory corruption and at minimum a reliable denial of service. The defect was present in the WireGuard implementation shipped in Zephyr 4.4.0. The fix adds an explicit `data_len > CONFIG_WIREGUARD_BUF_LEN` rejection and corrects the linearize call to pass `net_buf_max_len(buf)` as the destination capacity.

- Zephyr project bug tracker GHSA-3wqm-wgx2-9367

This has been fixed in main for v4.5.0

- PR 108841 fix for main
- PR 111084 fix for v4.4

CVE 2026-10667

SMP use-after-free in Zephyr `CONFIG_USERSPACE` dynamic kernel-object tracking, reachable from unprivileged user threads

Zephyr's dynamic kernel-object tracking (`kernel/userspace/userspace.c`, formerly `kernel/userspace.c`) maintains a doubly-linked list (`obj_list`) of dynamically allocated kernel objects. Iteration over this list in `k_object_wordlist_foreach()` was performed under `lists_lock` using the SAFE iterator (which caches the next node), but list removal and freeing of nodes was performed under different, disjoint spinlocks: `objfree_lock` in `k_object_free()` and `obj_lock` in `unref_check()`. On an SMP system, while one CPU iterated `obj_list` under `lists_lock`, another CPU could unlink and `k_free()` the `dyn_obj` node that the iterator had cached as its next pointer, causing the iterator to dereference freed kernel memory (use-after-free / dangling list traversal). All of the racing operations are reachable from unprivileged user-mode threads via system calls: `k_object_alloc`/`k_object_alloc_size` and `k_object_release` drive removals through `unref_check()` (under `obj_lock`), while `k_thread_abort` and thread creation drive the iteration through `k_thread_perms_all_clear()`/`k_thread_perms_inherit()` (under `lists_lock`). A deprivileged user thread on a `CONFIG_SMP` + `CONFIG_USERSPACE` build can therefore corrupt the kernel's object-tracking structures across the userspace security boundary, yielding kernel memory corruption (potential privilege escalation) or a kernel crash (denial of service). The fix removes `objfree_lock` and serializes every `obj_list` modification under `lists_lock`, including holding it across find+remove in `k_object_free()` and around `unref_check()` in `k_thread_perms_clear()`. Affects `CONFIG_SMP``+``CONFIG_USERSPACE``+``CONFIG_DYNAMIC_OBJECTS` configurations; the defect dates to the 2019 spinlockification (commit 8a3d57b6cc6, first released in v1.14.0) and shipped through v4.4.0.

- Zephyr project bug tracker GHSA-9x5j-h3rh-x579

This has been fixed in main for v4.5.0

- PR 108721 fix for main
- PR 111067 fix for v4.4
- PR 111019 fix for v4.3
- PR 111018 fix for v3.7

CVE 2026-10668

Host-triggerable control-endpoint wedge (DoS) in Nuvoton NuMaker HSUSBD UDC driver

The Nuvoton NuMaker HSUSBD USB device-controller driver (`drivers/usb/udc/udc_numaker.c`) armed the control Data IN stage unconditionally (`base->CEPTXCNT = len` in `numaker_hsusbd_ep_trigger`). Because the HSUSBD hardware cannot disarm a control Data IN already armed for a previous transfer, a USB host that cancels an in-flight control transfer (timeout) and then issues a new SETUP packet can drive the driver out of sync: stale data may be transmitted in the new transfer and the control endpoint can become permanently stuck NAK'ing every subsequent control transfer.

A malicious or buggy host (physical/adjacent attacker driving the bus) can repeatedly cancel-and-re-SETUP to wedge the device's USB control endpoint, denying service to the device's USB function (the device stops enumerating/responding on the control pipe) until a USB reset or replug. The flaw is an availability-only denial of service; the FIFO copy loops (bounded by `net_buf` length and the hardware BUFFULL flag) and the `net_buf` lifecycle are independent of the arming desync, so there is no out-of-bounds access, use-after-free, or information leak.

The fix monitors the IN-token and new-SETUP events (`k_event`) and only arms control Data IN when an IN token is present and no new SETUP has arrived, cancelling the current transfer on a new SETUP. Affects boards using the Nuvoton NuMaker HSUSBD controller (`CONFIG_UDC_NUMAKER` with `DT_HAS_NUVOTON_NUMAKER_HSUSBD_ENABLED`); shipped in v4.4.0.

- Zephyr project bug tracker GHSA-rm28-x84j-4qrx

This has been fixed in main for v4.5.0

- PR 107010 fix for main
- PR 110646 fix for v4.4

CVE 2026-10669

Xtensa MPU `arch_buffer_validate()` integer-overflow lets a user thread bypass syscall pointer validation

On Xtensa SoCs built with `CONFIG_XTENSA_MPU` and `CONFIG_USERSPACE`, `arch_buffer_validate()` in `arch/xtensa/core/mpu.c` — the architecture hook that verifies a user-mode-supplied buffer is accessible to the calling user thread with the requested permission — defaulted its return value to 0 (access permitted) and only set a denial result inside its per-MPU-region probe loop. When the rounded extent of the buffer wraps the 32-bit address space (size + alignment offset near `SIZE_MAX`, or `ROUND_UP(size + offset)` overflowing to 0), the loop executes zero iterations and the function returns 0 = permitted without probing any MPU region.

The syscall-layer pre-checks (`K_SYSCALL_MEMORY_SIZE_CHECK` / `Z_DETECT_POINTER_OVERFLOW`) only catch a raw `addr+size` wrap and do not cover the `ROUND_UP`-induced wrap, and the string path (`arch_user_string_nlen` -> `arch_buffer_validate`) has no syscall-layer guard at all.

An unprivileged user-mode thread can therefore pass a crafted `(addr, size)` to any syscall that validates user buffers via `k_usermode_from_copy/to_copy` or `k_usermode_string_copy` and have validation succeed for memory it must not access; the kernel then reads from (disclosure) or, with `write=1`, writes to (corruption) attacker-chosen kernel or other-partition memory on the thread's behalf, enabling information disclosure, memory corruption, privilege escalation, and denial of service.

Affected from v3.7.0 (when Xtensa MPU userspace support was added) through v4.4.0. The fix changes the default to `-EINVAL` (deny by default), adds an explicit `size_add_overflow` check, and sets the success value only after the full range has been validated.

- Zephyr project bug tracker GHSA-4r4p-gh69-v6w4

This has been fixed in main for v4.5.0

- PR 109000 fix for main
- PR 109239 fix for v4.4
- PR 109238 fix for v4.3
- PR 109236 fix for v3.7

CVE 2026-10670

User-triggerable kernel NULL-pointer dereference (DoS) in `k_thread_name_copy()` syscall verifier

The `CONFIG_USERSPACE` verification handler for the `k_thread_name_copy()` system call (`z_vrfy_k_thread_name_copy()` in `kernel/thread.c`) calls `k_object_find()` on the caller-supplied thread pointer and then dereferences the returned `struct k_object` without checking it for `NULL`. `k_object_find()` returns `NULL` whenever the supplied pointer is not a registered (static or dynamic) kernel object.

The pre-fix guard tested `thread == NULL` instead of `ko == NULL`, so an unprivileged user-mode thread that invokes `k_thread_name_copy()` with any non-NULL but unregistered pointer (e.g. an arbitrary address) passes the NULL test, after which the verifier reads `ko->type` through a NULL pointer.

Because the syscall verifier runs in supervisor mode, this NULL dereference is a kernel-mode fault that halts or reboots the system, allowing untrusted user code to crash the kernel across the userspace security boundary (denial of service). The marshaller passes the thread argument to the verifier without any prior `K_SYSCALL_OBJ` validation, so the bad pointer reaches the defect directly.

The flaw affects builds with `CONFIG_USERSPACE` and `CONFIG_THREAD_NAME` enabled and has been present since the special-case lookup was introduced around v2.0.0; it is present in v4.4.0 and

earlier. The fix changes the guard to check the `k_object_find()` return value (`ko == NULL`) before dereferencing it.

- Zephyr project bug tracker GHSA-82h2-v4vm-q2g9

This has been fixed in main for v4.5.0

- PR 109076 fix for main
- PR 111088 fix for v4.4
- PR 111089 fix for v4.3
- PR 109364 fix for v3.7

CVE 2026-10671

User thread can re-initialize an in-use `k_pipe`, corrupting kernel wait queues (`CONFIG_USERSPACE`)

In Zephyr's kernel pipe implementation, the userspace syscall verifier `z_vrfy_k_pipe_init()` in `kernel/pipe.c` used `K_SYSCALL_OBJ()` (which requires the kernel object to already be initialized) instead of `K_SYSCALL_OBJ_NEVER_INIT()` (which rejects an already-initialized object). As a result, on `CONFIG_USERSPACE` builds an unprivileged user thread that has been granted access to a `k_pipe` object can invoke the `k_pipe_init` syscall to re-initialize a pipe that is already in use.

`z_impl_k_pipe_init()` unconditionally resets the ring buffer, sets `pipe->waiting` to 0, and re-initializes both wait queues (`z_waitq_init` on `pipe->data` and `pipe->space`) without waking or accounting for threads currently blocked on the pipe. Any thread already pended in `k_pipe_read()`/`k_pipe_write()` is left orphaned: still marked pending with `pended_on` pointing at the cleared wait queue and with stale `qnode_dlist` links into the (now re-initialized) embedded list head.

When such an orphaned waiter is later timed out or woken, the scheduler calls `sys_dlist_remove()` on its stale node, writing through dangling `prev`/`next` pointers into kernel wait-queue/scheduler structures, causing list corruption (an attacker-driven invalid kernel write), lost wakeups, indefinitely blocked threads, and silent data loss. The flaw lets a deprivileged user thread corrupt the state of a kernel object shared with other threads/partitions.

The fix switches the verifier to `K_SYSCALL_OBJ_NEVER_INIT()`, matching the existing `k_msgq_init` verifier, so a user thread can no longer re-initialize a live pipe. The vulnerable code shipped in v4.1.0 and remained through v4.4.0.

- Zephyr project bug tracker GHSA-p8w8-3x99-mg8f

This has been fixed in main for v4.5.0

- PR 109091 fix for main
- PR 111101 fix for v4.4
- PR 111102 fix for v4.3

CVE 2026-10672

Unterminated URI buffer causes out-of-bounds read in LwM2M firmware pull (Package URI)

`subsys/net/lib/lwm2m/lwm2m_pull_context.c` copied the firmware-update Package URI into a fixed static buffer (`context.uri`, size `CONFIG_LWM2M_SWMGMT_PACKAGE_URI_LEN`, default 128) with `memcpy(context.uri, uri, LWM2M_PACKAGE_URI_LEN)`, copying exactly the destination size with no length validation. The Firmware-Update object stores the server-supplied Package URI (/5/0/1) in a 255-byte buffer, so a LwM2M management server (or an on-path attacker on a session lacking strong DTLS) can WRITE a URI of 128-254 characters; only the first 128 bytes are then copied into `context.uri` with no NUL terminator. That buffer is subsequently consumed as a C string by `http_parser_parse_url(context.uri, strlen(context.uri), ...)`, `strlen`-based CoAP URI-path/PROXY-URI option appends, and `lwm2m_parse_peerinfo()`, causing

an out-of-bounds read of adjacent static memory. The over-read bytes are appended to outbound CoAP requests (information disclosure of adjacent device memory to the server/proxy) and can crash the device (denial of service). The vulnerable copy was introduced by the pull-context refactor (first released in v3.0.0) and is present through v4.4.0; the default-on `CONFIG_LWM2M_FIRMWARE_UPDATE_PULL_SUPPORT` path is affected. The fix adds a `strlen(uri) >= sizeof(context.uri)` check returning `-ENOMEM` and switches to `strcpy()`, guaranteeing a bounded, NUL-terminated buffer.

- Zephyr project bug tracker GHSA-rf6j-4mpp-j9mf

This has been fixed in main for v4.5.0

- PR 108964 fix for main
- PR 109235 fix for v4.4
- PR 109234 fix for v4.3
- PR 109233 fix for v3.7

CVE 2026-10674

DoS (hard fault) in NXP LPUART driver: unsupported runtime UART config leaves clocks disabled

The NXP LPUART serial driver (`drivers/serial/uart_mcux_lpuart.c`), when `CONFIG_UART_USE_RUNTIME_CONFIGURE` is enabled, called `LPUART_Deinit()` at the start of `mcux_lpuart_configure()`, which disables the LPUART peripheral clocks. The requested configuration is validated only afterwards (in `mcux_lpuart_configure_basic`), and unsupported parity/data-bit/stop-bit/flow-control values return `-ENOTSUP` before the clock is re-enabled.

As a result, a `uart_configure()` request with an unsupported configuration left the LPUART in a clock-disabled state; any subsequent access to LPUART registers (`poll_out`/`poll_in`, interrupt handling, or a later reconfigure) faults on the gated peripheral and escalates to a hard fault, crashing the system.

`uart_configure()` is a Zephyr syscall whose verifier (`z_vrfy_uart_configure`) only checks that `cfg` is readable user memory and forwards the caller-supplied configuration unchanged, so an unprivileged userspace thread with access to an LPUART device can deterministically trigger the fault, a persistent system-wide denial of service.

Introduced in v2.5.0 and present in all subsequent releases until this fix, which removes the `LPUART_Deinit()` call and instead only disables the transmitter/receiver, leaving the clock running.

- Zephyr project bug tracker GHSA-mw68-r353-m3vf

This has been fixed in main for v4.5.0

- PR 107186 fix for main
- PR 111106 fix for v4.4
- PR 111105 fix for v4.3
- PR 111104 fix for v3.7

CVE 2026-10675

Bluetooth Mesh PB-ADV: invalidated provisioning link kept alive indefinitely, blocking (re)provisioning (DoS)

In Zephyr's Bluetooth Mesh PB-ADV provisioning bearer (`subsys/bluetooth/mesh/pb_adv.c`), `prov_msg_recv()` rescheduled the provisioning protocol watchdog timer unconditionally at the top of the function, before the FCS check and before the `ADV_LINK_INVALID` check. Once a provisioning attempt fails, `prov_failed()` sets `ADV_LINK_INVALID` and the only recovery path is the

protocol timer firing (`protocol_timeout -> prov_link_close -> close_link -> reset_adv_link` and re-enabling of scanning and the unprovisioned device beacon).

A remote, unauthenticated attacker on the BLE advertising channel can first induce a provisioning failure (e.g. with a malformed generic-provisioning PDU) and then transmit any FCS-valid PB-ADV transaction PDU on the same link ID more often than once per protocol timeout (60 s, or 120 s for OOB input/output). Because each such packet reset the timer even on an invalidated link, `protocol_timeout` never fired, the dead link was never torn down, and the device remained pinned in an un-provisionable state with its unprovisioned beacon disabled and new Link Open requests rejected.

PB-ADV PDUs are processed without authentication and the FCS is a keyless CRC, so no pairing or prior trust is required and the attacker chooses the link ID itself. The impact is a persistent denial of provisioning/re-provisioning service; there is no memory-safety, confidentiality, or integrity impact.

The vulnerable code shipped in releases through v4.4.1. The fix moves the timer reschedule to after the `ADV_LINK_INVALID` check (and the FCS check before the reset) so an invalidated link can no longer be kept alive by incoming packets.

- Zephyr project bug tracker GHSA-4rwg-6mr4-55hc

This has been fixed in main for v4.5.0

- PR 109324 fix for main
- PR 110910 fix for v4.4
- PR 110909 fix for v4.3
- PR 110911 fix for v3.7

CVE 2026-10677

Kernel heap memory leak in `z_vrfy_k_poll()` lets an unprivileged user thread exhaust the kernel resource pool

The `CONFIG_USERSPACE` syscall verifier `z_vrfy_k_poll()` in `kernel/poll.c` allocates a kernel-side copy of the user-supplied `k_poll_event[]` via `z_thread_malloc()` and then validates each event's object handle. Before this fix, validation used `K_OOPS(K_SYSCALL_OBJ(...))` inline inside the loop, which kills the calling thread without freeing `events_copy`.

A user thread can pass `num_events >= 1` with a forged object handle to leak the allocation; because newly spawned user threads inherit the parent's `resource_pool` (`kernel/thread.c`), an attacker spawns sacrificial threads to repeat the leak until the shared kernel heap is exhausted. Once depleted, legitimate kernel allocations from that pool (`k_queue` alloc nodes, `k_msgq` buffers, future `k_poll` calls, etc.) fail, causing a system-level denial of service.

The fix replaces each inline `K_OOPS` with a conditional `goto oops_free` so the buffer is freed before the thread is killed. Affects Zephyr releases from v1.12.0 (when `k_poll` was first exposed to user mode) through v4.4.1.

- Zephyr project bug tracker GHSA-r3cc-8wcr-xfj9

This has been fixed in main for v4.5.0

- PR 109361 fix for main
- PR 111111 fix for v4.4
- PR 111112 fix for v4.3
- PR 109535 fix for v3.7

CVE 2026-10678

NULL-pointer / out-of-bounds write in Zephyr MCTP I2C+GPIO target binding driven by an unauthenticated I2C controller

The MCTP-over-I2C+GPIO target binding in Zephyr (`subsys/pmci/mctp/mctp_i2c_gpio_target.c`) processes pseudo-register writes from an I2C bus master byte-by-byte in `mctp_i2c_gpio_target_write_received()` without validating the order or the receive buffer. In the affected versions the `MCTP_I2C_GPIO_RX_MSG_ADDR` (data) handler dereferences and writes through `b->rx_pkt` without checking that the receive buffer was allocated: a controller that selects the data register and writes a byte without first sending the length register (which is what allocates the buffer) causes a write of an attacker-chosen byte through a NULL/unallocated `mctp_pktbuf` pointer (i.e. into a small attacker-advanceable offset above address 0), producing memory corruption or a hard fault.

The same handler also performs a write-then-check bounds test, allowing a one-byte heap overflow at `data[255]` when more than 255 data bytes are sent.

Because the I2C target callback is invoked with raw bytes supplied by whatever device is the bus master and the binding performs no authentication, a malicious or malfunctioning controller on the bus can trigger these without any prior protocol state, leading to memory corruption and/or denial of service on the target device.

The vulnerable code was introduced when the I2C+GPIO target binding was added and shipped in Zephyr v4.3.0 and v4.4.0. The fix defers allocation to the first data byte with a NULL check, treats a missing length as a zero-sized packet rejected by libmctp, and moves the bounds check before the store.

- Zephyr project bug tracker GHSA-pmwm-5rcm-39rr

This has been fixed in main for v4.5.0

- PR 109428 fix for main
- PR 111118 fix for v4.4
- PR 111117 fix for v4.3

CVE 2026-10679

Divide-by-zero in DesignWare SPI driver reachable from spi_transceive syscall (local DoS)

The DesignWare SPI driver (`drivers/spi/spi_dw.c`) computed the SPI BAUDR clock divider as `info->clock_frequency / config->frequency` without validating `config->frequency`.

`spi_transceive` is a Zephyr `__syscall` and its verify handler (`drivers/spi/spi_handlers.c`) copies the caller-supplied `spi_config` from userspace without checking the frequency field, so a userspace thread that has been granted access to a DesignWare SPI device kernel object can pass `frequency = 0` and trigger an unsigned integer divide-by-zero in `spi_dw_configure()`.

On Cortex-M Mainline (`SCB->CCR.DIV_0_TRP` is set in `z_arm_fault_init()`) and on ARC (a dedicated `__ev_div_zero` vector) this raises a CPU exception, resulting in a kernel fault and local denial of service.

The fix rejects zero frequency and frequencies above `clock_frequency / 2` (the DesignWare SSI databook minimum SCKDIV of 2) with `-EINVAL`. The defect affects all Zephyr releases up to and including v4.4.0; exploitation requires `CONFIG_USERSPACE=y` and an unprivileged thread already granted SPI driver permission. There is no memory-corruption or information-disclosure impact.

- Zephyr project bug tracker GHSA-3qcm-qwh2-v4hq

This has been fixed in main for v4.5.0

- PR 105452 fix for main
- PR 111121 fix for v4.4

- PR 111120 fix for v4.3

CVE 2026-10680

Out-of-bounds access in Zephyr BR/EDR L2CAP configuration request handling via `uint16_t` length underflow

The Classic (BR/EDR) L2CAP signaling handlers `l2cap_br_conf_req()` and `l2cap_br_conf_rsp()` in `subsys/bluetooth/host/classic/l2cap_br.c` validated the minimum command size against `buf->len` (the bytes remaining in the whole received PDU) instead of `len` (the per-command data length from the L2CAP signaling header). Because multiple signaling commands can be packed into one PDU, `buf->len` may exceed a command's `len`. An attacker can send a `CONF_REQ` command with a header length smaller than the configuration-request structure (e.g. 0), followed by another command so that `buf->len` still satisfies the check. The check then passes incorrectly and `opt_len = len - sizeof(*req)` underflows the `uint16_t` to a near-0xFFFF value. The configuration-option loop, which lacks an `opt_len`-versus-`buf->len` guard, then walks far past the end of the pooled ACL receive buffer using `net_buf` pull primitives that perform no runtime bounds check, producing an out-of-bounds read of host memory and, when the out-of-bounds option bytes encode an MTU or flush-timeout option, an out-of-bounds write. The BR/EDR signaling channel is processed before pairing/encryption and an L2CAP channel to an L0 service such as SDP can be opened without pairing, so an unauthenticated peer within radio range that can establish an ACL connection can trigger the flaw, leading to memory corruption and denial of service (host/device crash). The defect is present in released versions including v4.4.0. The fix validates against `len` instead of `buf->len` in both handlers.

- Zephyr project bug tracker GHSA-vrwx-p97q-8854

This has been fixed in main for v4.5.0

- PR 109308 fix for main
- PR 110661 fix for v4.4
- PR 110662 fix for v4.3
- PR 111405 fix for v3.7

CVE 2026-10681

SMP race in `thread_idx_alloc()` lets concurrent `k_object_alloc(K_OBJ_THREAD)` callers share a kernel-object permission slot

In Zephyr's userspace dynamic-objects subsystem, `thread_idx_alloc()` in `kernel/userspace/userspace.c` allocated a new thread permission index from the global `_thread_idx_map[]` bitmap without holding `lists_lock`.

On SMP systems, two user-mode threads invoking the `k_object_alloc(K_OBJ_THREAD)` syscall concurrently can both observe the same low free bit, perform the same non-atomic RMW to clear it, and return the identical `tidx`.

The two newly created `K_OBJ_THREAD` objects are then assigned the same `thread_id`, so the two user threads alias a single bit position in every kernel object's `perms[]` bitfield: any subsequent grant of access on a kernel object to one thread is implicitly a grant to the other, defeating userspace ACL isolation. A secondary lost-update window between the unlocked `&=~BIT()` in alloc and the locked `|= BIT()` in `thread_idx_free()` can also leak entries from the thread-index pool.

The defect is reachable from any user-mode thread via the unrestricted `__syscall k_object_alloc` and is gated on `CONFIG_USERSPACE`, `CONFIG_DYNAMIC_OBJECTS`, and `CONFIG_SMP`. The flaw was introduced when the per-thread permission index was added in 2018 and is present in every release up to and including v4.4.0. Fixed by holding `lists_lock` across the bitmap RMW and the permissions clear (and inlining the `obj_list` traversal that previously took the lock itself).

- Zephyr project bug tracker GHSA-j693-5rh5-8g8h

This has been fixed in main for v4.5.0

- PR 109616 fix for main
- PR 111409 fix for v4.4
- PR 111410 fix for v4.3
- PR 111408 fix for v3.7

CVE 2026-10682

Out-of-bounds write in Zephyr `log_filter_set` syscall verifier reachable from userspace

The userspace verifier `z_vrfy_log_filter_set()` for the `log_filter_set` syscall in `subsys/logging/log_mgmt.c` performed a signed comparison against the `int16_t src_id` parameter: `src_id < (int16_t)log_src_cnt_get(domain_id)`. Any negative value for `src_id` (e.g. -1) trivially satisfied this check and was forwarded into `z_impl_log_filter_set`, where it propagated to `filter_set()` and ultimately to `get_dynamic_filter()`, which uses `source_id` as an unsigned index into the linker-section array `&TYPE_SECTION_START(log_dynamic)[source_id].filters`.

After implicit conversion through `uint32_t`, an `int16_t` -1 becomes 0xFFFFFFFF, indexing `log_dynamic` far out of bounds and causing the kernel to perform an OOB read and an OOB read-modify-write (`LOG_FILTER_SLOT_GET/SET`) against memory adjacent to the `log_dynamic` section.

The written value is a constrained 3-bit log level slot within the targeted 32-bit word, but the target address is attacker-chosen (a small negative offset from `log_dynamic`) and the write occurs in supervisor mode following a syscall from an unprivileged user thread, providing a kernel memory-corruption / privilege-escalation primitive.

The defect is reachable on any build with `CONFIG_USERSPACE=y` and `CONFIG_LOG_RUNTIME_FILTERING=y`. Present from Zephyr v3.3.0 through v4.4.1. The fix replaces the signed bound check with an unsigned comparison: `(uint32_t)src_id < log_src_cnt_get(domain_id)`, which correctly rejects negative inputs.

- Zephyr project bug tracker GHSA-6vqh-mg7h-58qh

This has been fixed in main for v4.5.0

- PR 109690 fix for main
- PR 111418 fix for v4.4
- PR 111419 fix for v4.3
- PR 111417 fix for v3.7

CVE 2026-10683

DesignWare I2C target driver can be wedged into a permanent stuck state by an on-bus master (DoS)

In the Synopsys DesignWare I2C driver (`drivers/i2c/i2c_dw.c`) operating in target/slave mode, the `rx_full` interrupt handler gates the `write_requested()` callback on `dw->state != CMD_SEND`, and `dw->state` is only reset to READY on a STOP interrupt. The `START_DET` interrupt, whose handler in `i2c_dw_slave_read_clear_intr_bits()` would reset the state on every (re)START, was never added to the enabled interrupt mask in `i2c_dw_slave_register()`, so that recovery path was dead code.

As a result, if the STOP interrupt is lost (bus glitch/reset, or a concurrent master driving STOP) or the bus master issues a legal WRITE-repeated-START-WRITE sequence with the same direction, the driver remains in `CMD_SEND` permanently and never invokes `write_requested()` again for the life of the target.

An I2C master on the same physical bus can deliberately trigger this, causing the I2C target function to malfunction for all subsequent write transactions and desynchronizing consumer framing state (e.g. MCTP-over-I2C), a recoverable-by-reset denial of service of the target peripheral.

The fix unmasks `START_DET` so the state is reset at every bus (re)START. Impact is availability-only over a local board-level bus; no memory corruption results in the in-tree consumer, whose per-byte buffer write is independently bounds-checked.

- Zephyr project bug tracker GHSA-fj9c-r5qw-3639

This has been fixed in main for v4.5.0

- PR 107537 fix for main
- PR 111415 fix for v4.4
- PR 111414 fix for v4.3

CVE 2026-10684

Out-of-bounds read in coredump shell when printing stored-dump target code

In `subsys/debug/coredump/coredump_shell.c`, `print_coredump_hdr()` used the 16-bit `tgt_code` field of a stored Zephyr coredump header directly as an index into `coredump_target_code2str[]`, a fixed 7-element array of string pointers, with no bounds check.

A stored coredump whose `tgt_code` is >= 7 causes an out-of-bounds read of a `char*` up to ~64K entries past the array; that value is passed as the `%s` argument to `shell_print`, which dereferences and walks it as a string. The result is either disclosure of device memory contents to the shell user or a crash when the out-of-bounds pointer is unmapped.

The defect is reached via the `coredump print` shell command (`cmd_coredump_print_stored_dump` -> `pretty_print_coredump` -> `parse_and_print_coredump` -> `print_coredump_hdr`). The `tgt_code` field is device-generated and in-range during normal crash handling, so triggering requires local shell access plus the ability to stage or corrupt the stored coredump in the flash/in-memory backend.

Introduced in v4.2.0 (commit 13abd7fe730) and present through v4.4.0; fixed by clamping out-of-range codes to the 'unknown' (index 0) entry.

- Zephyr project bug tracker GHSA-9fw2-4429-49q8

This has been fixed in main for v4.5.0

- PR 109630 fix for main
- PR 111421 fix for v4.4
- PR 111422 fix for v4.3

CVE 2026-10685

Use-after-free of GATT subscribe params in Bluetooth host CCC-write response handler

The Zephyr Bluetooth GATT client CCC-write response handler `gatt_write_ccc_rsp()` in `subsys/bluetooth/host/gatt.c` invoked the application's `params->subscribe()` callback after it had already called `params->notify(conn, params, NULL, 0)`.

Per the public GATT API, a notify callback with `NULL` data is the documented signal that the subscription has terminated and the `bt_gatt_subscribe_params` struct may be freed or reused by the application; calling `subscribe()` on the struct afterwards is a use-after-free, including an indirect call through the freed `params->subscribe` function pointer.

The error branch is remotely (adjacent) reachable: a Zephyr device acting as a GATT client that calls `bt_gatt_subscribe()` can be driven into this ordering when a connected GATT server peer answers the CCC write with an ATT Error Response (the peer-supplied error code flows through `att_error_rsp` -> `att_handle_rsp` into `gatt_write_ccc_rsp`).

For applications that free or recycle subscription parameters in their notification-termination handler, this results in memory corruption, a crash (denial of service), or potentially attacker-influenced control flow. The fix reorders the handler so the `subscribe()` callback runs before the terminating `notify(NULL)` in both the error and unsubscribe paths.

- Zephyr project bug tracker GHSA-29xh-jm2m-4qvx

This has been fixed in main for v4.5.0

- PR 99920 fix for main
- PR 111430 fix for v4.4
- PR 111429 fix for v4.3
- PR 111428 fix for v3.7

CVE 2026-10686

Missing hop-limit decrement on IPv6 forwarding path allows unbounded packet looping (DoS) in Zephyr routers

Zephyr's IPv6 forwarding path re-sent routed unicast packets without ever decrementing the IPv6 hop limit. Both routing branches of `ipv6_route_packet()` (`subsys/net/ip`) were affected: the explicit-route path (`net_route_packet()`) and the on-link cross-interface path (`net_route_packet_if()`). Each set the packet forwarding flag and called `net_send_data()` with the hop limit untouched and no expiry check.

Per RFC 8200 the hop-limit decrement is the mechanism that bounds packet lifetime and terminates routing loops; without it, a device acting as an IPv6 router relays looping packets indefinitely. An on-path attacker who can induce or exploit a transient L3 loop turns it into a permanent forwarding storm, causing CPU/bandwidth resource exhaustion (availability DoS) on the forwarder and adjacent links; path-discovery and loop diagnostics that rely on hop-limit expiry are also defeated.

Affected configurations. In every affected release the forwarding path is reached via `CONFIG_NET_ROUTE` (enabled by default when `CONFIG_NET_IPV6_NBR_CACHE` is set), together with `CONFIG_NET_ROUTING` for cross-interface routing. Note that `CONFIG_NET_IPV6_FORWARDING` and `CONFIG_NET_IPV4_FORWARDING` — which appear in the fix and in this advisory's evidence notes — were introduced *after* v4.4.0, when the routing options were split and renamed; they do not exist in any affected release. When auditing a v4.4.1-or-earlier configuration, look for `CONFIG_NET_ROUTE` and `CONFIG_NET_ROUTING`.

IPv4 is not affected in any release. The IPv4 forwarding path (`net_route_ipv4_packet()` in `route_ipv4.c`) was added after v4.4.0 and has never shipped in a release. Its TTL decrement and IPv4 header-checksum recomputation landed on `main` as part of the same fix, so the evidence notes below discuss it, but no released version is reachable by way of IPv4.

Affected releases are v1.8.0 through v4.4.1: v1.8.0 introduced `net_route_packet()` and v2.2.0 added `net_route_packet_if()`, and neither decremented the hop limit. v4.3.1 carries the explicit-route fix but not the on-link one, so it is affected as well. Fixed on `main` by 7d8f1afa7345 (explicit-route path) and 589eadc74efa (on-link path).

- Zephyr project bug tracker GHSA-4cg6-6jc4-2r6h

This has been fixed in main for v4.5.0

- PR 109585 fix for main
- PR 111451 fix for v4.4
- PR 111450 fix for v4.3
- PR 111449 fix for v3.7

CVE 2026-10687

Under embargo until 2026-08-01

CVE 2026-10772

Under embargo until 2026-08-01

CVE 2026-10773

Out-of-bounds read in DHCPv4 client message-type name lookup (net_dhcpv4_msg_type_name)

The DHCPv4 client helper `net_dhcpv4_msg_type_name()` in `subsys/net/lib/dhcpv4/dhcpv4.c` indexes a static 8-element `const char *` name table after a faulty bounds check. The guard used `msg_type <= sizeof(name)` instead of `msg_type <= ARRAY_SIZE(name)`; `sizeof` returns the byte size of the pointer array (32 on 32-bit, 64 on 64-bit targets) rather than the element count of 8, so message-type values from 9 up to that byte size pass the check and cause `name[msg_type - 1]` to read past the end of the array.

The `msg_type` value originates from the DHCP MESSAGE TYPE option, which is read as an unchecked raw byte from a received packet (`net_pkt_read_u8`) and passed unmodified into the lookup. A DHCP server, or any host able to inject a spoofed DHCP reply onto the client's link, can therefore drive the index out of bounds. The out-of-range slot yields a garbage `const char *` that is then dereferenced by a %s log conversion.

The lookup is reached only from a debug log statement (`NET_DBG` / `LOG_DBG`), so the out-of-bounds read is triggerable only when the DHCPv4 log module is built at DEBUG level (`CONFIG_NET_DHCPV4_LOG_LEVEL_DBG`), which is not the default configuration. When that condition holds, the result is an out-of-bounds read and a wild-pointer dereference: most likely a crash of the DHCP client (denial of service) and potentially disclosure of an adjacent pointer's contents through the log output. The fix replaces `sizeof` with `ARRAY_SIZE`, restoring the correct 1..8 acceptance window.

- Zephyr project bug tracker GHSA-r5hq-xq42-wcfq

This has been fixed in main for v4.5.0

- PR 110135 fix for main
- PR 112423 fix for v4.4
- PR 115146 fix for v3.7

CVE 2026-10774

PSA key-slot leak in Bluetooth Mesh subnet deletion leading to resource-exhaustion DoS

Zephyr's Bluetooth Mesh subnet key management leaks one PSA Crypto key slot on every subnet-key teardown. In `subsys/bluetooth/mesh/subnet.c`, `net_keys_create()` imports the Private Beacon Key into a PSA key slot under `CONFIG_BT_MESH_PRIV_BEACONS` (enabled by default), but `subnet_keys_destroy()` guarded the matching `psa_destroy_key()` with `CONFIG_BT_MESH_V1d1`. That Kconfig symbol was removed when explicit Mesh 1.0.1 support was dropped, so the destroy branch became permanently dead code and the import is never balanced by a destroy.

The imbalanced teardown is reached every time subnet keys are destroyed: deleting a subnet (Config Server NetKey Delete), completing a Key Refresh Procedure (which retires the old key set), and resetting/re-provisioning the node. The over-the-air triggers are processed only under the node's device key, so they are exercisable by the provisioner or network administrator that owns the node, reachable over the Bluetooth Mesh network.

With the default `CONFIG_MBEDTLS_PSA_KEY_SLOT_COUNT` of 16, repeated add/delete or key-refresh cycles exhaust the shared PSA key-slot pool after roughly a dozen rounds. Once exhausted, `bt_mesh_private_beacon_key()` and thus subnet creation fail: the node can no longer add subnets or complete key refresh, and other PSA crypto consumers on the device may be

starved, until the device is rebooted. The fix aligns the destroy guard with the import guard (`CONFIG_BT_MESH_PRIV_BEACONS`) so each slot is freed.

- Zephyr project bug tracker GHSA-6q7g-798f-76p2

This has been fixed in main for v4.5.0

- PR 110235 fix for main
- PR 110438 fix for v4.4
- PR 110437 fix for v4.3
- PR 110436 fix for v3.7

CVE 2026-10848

Out-of-bounds read in Zephyr OCPP 1.6 RPC message parser (parse_rpc_msg)

The OCPP 1.6 client in `subsys/net/lib/ocpp` parsed inbound WAMP RPC frames in `parse_rpc_msg()` (`subsys/net/lib/ocpp/ocpp_j.c`) using a hand-rolled helper, `extract_string_field()`, that copied the message's `uid` and `action` fields with `strncpy(out_buf, token + 1, outlen - 1)` and then scanned the result with `strchr(out_buf, '"')`. Because `strncpy` does not NUL-terminate the destination when the source is at least `outlen - 1` (127) bytes long, the subsequent `strchr` reads past the 128-byte destination buffer into adjacent stack memory; if a `"` byte is found beyond the buffer, a one-byte out-of-bounds NUL write also occurs. A related defect in `extract_payload()` runs `strchr`/`strrchr` over the receive buffer, which may not be NUL-terminated when a maximal-length frame fills it.

The parsed bytes come directly from the OCPP central-system server over a websocket: the reader thread fills `recv_buf` via `websocket_recv_msg()` and calls `parse_rpc_msg()` on each inbound DATA frame (`subsys/net/lib/ocpp/ocpp.c`). A malicious or compromised central server, or an on-path attacker (OCPP is commonly deployed over plain `ws://`), can send an RPC frame whose `uid` or `action` field is 127+ bytes with no closing quote, triggering the out-of-bounds access.

The primary impact is a remotely triggerable denial of service: the unbounded scan can fault on an unmapped page, and the stray NUL write can corrupt adjacent stack state. The over-read data is not reflected to the peer, so disclosure is limited. The feature is EXPERIMENTAL and must be explicitly enabled (`CONFIG_OCPP`). The fix replaces the manual parser with the bounds-respecting `json_mixed_arr_parse()` and copies the extracted `uid` with an explicitly NUL-terminated buffer, eliminating both over-reads.

- Zephyr project bug tracker GHSA-jgqq-7mjj-w642

This has been fixed in main for v4.5.0

- PR 95399 fix for main
- PR 112426 fix for v4.4

CVE 2026-10849

Heap out-of-bounds write in Zephyr hawkBit OTA client when terminating server response body

The hawkBit device management client in `subsys/mgmt/hawkbit` accumulates the body of an HTTP response from the update server into a heap buffer in `response_json_cb()` (`subsys/mgmt/hawkbit/hawkbit.c`). The buffer is sized to hold the received body bytes but reserves no space for a terminating NUL. When the full response has arrived, the code writes `response_data[downloaded_size] = '\0'` — and whenever the accumulated body length equals the allocation, that terminator lands one byte past the end of the heap object (a heap-based out-of-bounds write, CWE-122 / CWE-787).

The body length and fragmentation are taken directly from the parsed HTTP response (`rsp->body_frag_start` / `rsp->body_frag_len`) and are fully controlled by the remote hawkBit server, which chooses its own response length. The precise trigger depends on how the

buffer grows, and both forms are remotely reachable. Since v4.0.0 the reallocation is sized to exactly `downloaded_size + body_len`, so **any** response body larger than the 1100-byte initial buffer makes the out-of-bounds write deterministic; such response sizes are normal for hawkBit deployment metadata. Before v4.0.0 the buffer grew by doubling and the growth check (`(downloaded_size + body_len) > response_buffer_size`) is false at equality, so a response body whose length is exactly the current allocation — 1100 bytes with the default initial buffer — skips the reallocation entirely and writes the terminator at `response_data[1100]` of an 1100-byte object. The HTTP length-mismatch check does not catch this, because the declared and received lengths genuinely agree. Either form is reachable by a malicious, compromised, or man-in-the-middle update server (TLS is optional and, when enabled, does not protect against a hostile server), with no authentication of response content and no client-side length cap protecting the write.

The out-of-bounds write is a fixed single NUL byte immediately following the allocation, corrupting adjacent allocator metadata or the next allocation. The practical impact is heap corruption leading to denial of service (fault on a subsequent allocation or free), with the bounded, allocator-dependent possibility of further corruption. The fix sizes the buffer to the body length plus one and copies with `memcpy`, ensuring the terminator always lands within the allocation.

- Zephyr project bug tracker GHSA-39h3-7phx-pwhv

This has been fixed in main for v4.5.0

- PR 109285 fix for main
- PR 112429 fix for v4.4
- PR 112428 fix for v4.3
- PR 115262 fix for v3.7

CVE 2026-11368

Use-after-free in Bluetooth host ATT TX completion on disconnect mid-transfer

The Bluetooth host ATT layer (`subsys/bluetooth/host/att.c`) associates each in-flight ATT TX buffer with its owning channel via the static `tx_meta_data_storage[]` array (`data->att_chan = chan`). When a buffer's last reference is dropped, its net-buf destroy callback defers the completion handling to the system workqueue (`att_tx_destroy -> att_tx_destroy_work_handler -> att_on_sent_cb -> bt_att_sent`), where `bt_att_sent` dereferences the channel and its ATT context (`sys_slist_get(&att->reqs)`).

When a peer disconnects while an ATT PDU (a server notification/indication or any response) is still in flight in the controller TX path, L2CAP tears the channel down in `l2cap_chan_del()`: it runs the disconnected callback and then the released callback (`bt_att_released`), which frees the channel slab slot. Because the in-flight buffer is held by the connection TX path rather than the channel's own queue, its deferred destroy work can run after the channel has been freed. The `att_on_sent_cb` guard intended to drop the stale callback itself dereferences `meta->att_chan`, which is now a dangling pointer into a freed (and possibly reused) slab slot.

A remote peer with an ATT connection can drive this by disconnecting during routine ATT traffic; no pairing or user interaction is required to reach the ATT bearer. The result is a use-after-free read/write of freed channel memory, reliably crashing the Bluetooth host (denial of service) and, because the channel slab slot may be reused, potentially corrupting live memory.

The fix makes `bt_att_released()` `NULL` the `att_chan` field of every `tx_meta_data_storage[]` entry still referencing the channel before freeing it, so the deferred guard observes a `NULL` pointer and drops the callback. Teardown and the destroy work both run on the cooperative system workqueue, so the array update is serialized and needs no lock.

- Zephyr project bug tracker GHSA-85vg-gwc4-77g7

This has been fixed in main for v4.5.0

- PR 110416 fix for main

- PR 112431 fix for v4.4

CVE 2026-11742

Use-after-free race in kernel `k_queue_peek_head/tail` due to missing spinlock

The kernel queue helper `z_queue_node_peek()` in `kernel/queue.c` dereferences a node taken from a queue's `data_q` list, reading the node's flag byte and, for items enqueued via `k_queue_alloc_append/alloc_prepend`, the data pointer of an internally allocated `alloc_node` struct. The implementations of `z_impl_k_queue_peek_head()` and `z_impl_k_queue_peek_tail()` performed this read-and-dereference without holding the queue's spinlock, while every other accessor of the same list — including `k_queue_get()`, which unlinks a node and `k_free()`s its backing `alloc_node` — operates under that lock.

Because peek was unsynchronized, a concurrent `k_queue_get()` on the same queue (on an SMP build, or under preemption/ISR concurrency) can free the node between the moment peek obtains the node pointer and the moment it dereferences it. The peek then reads flag bits and a data pointer out of freed, potentially re-allocated heap memory and returns a stale or dangling pointer to its caller. `k_fifo` and `k_lifo` are thin wrappers over `k_queue`, so this affects buffer queues used throughout the `net_buf`, Bluetooth, USB, and networking subsystems; the peek operations are also system calls reachable from `CONFIG_USERSPACE` threads.

The consequences are a use-after-free read that can leak stale heap contents (one pointer word) and, when the returned dangling pointer is subsequently consumed as a live buffer, a dereference that can crash the system or corrupt memory. Exploitation requires winning a small race window with local access (e.g. a userspace process racing `k_queue_peek_*` against `k_queue_get` on a shared queue, or two CPUs), so practical impact is bounded and of low severity.

The fix wraps both peek implementations with `k_spin_lock/k_spin_unlock` on the queue lock, making the read-and-dereference atomic with respect to the concurrent unlink-and-free and bringing peek into line with the rest of the queue's locking discipline.

- Zephyr project bug tracker GHSA-8xm3-4w69-29mm

This has been fixed in main for v4.5.0

- PR 110576 fix for main
- PR 112439 fix for v4.4
- PR 112438 fix for v4.3
- PR 112437 fix for v3.7

CVE 2026-11743

Missing negative-offset/overflow check in SF32LB MPI QSPI NOR flash driver allows out-of-bounds read and write

The SF32LB MPI QSPI NOR flash driver (drivers/flash/flash_sf32lb_mpi_qspi_nor.c) validated the flash offset and length on its read and write paths with the test `(offset + size) > data->size`. Because `offset` is a signed `off_t` while `size` is unsigned, a negative offset is converted to a large unsigned value and the addition can wrap to a small result that passes the check. The read path then performs `memcpy(dst, (void *)(data->base + offset), size)` and the write path programs flash at `offset` and cache-invalidates `data->base + offset`, in both cases accessing memory outside the mapped flash window. The driver's erase path already rejected negative offsets, but read and write did not.

In builds with `CONFIG_USERSPACE`, `flash_read` and `flash_write` are syscalls whose verifiers validate the device object and the caller's buffer but deliberately delegate offset bounds checking to the driver. An unprivileged thread that has been granted access to this flash device can therefore call the syscall with a crafted negative offset and a buffer valid in its own memory domain, and reach the unchecked access.

The most direct impact is on the read path: by choosing a negative offset and matching size, an attacker slides the `memcpy` source below the flash base and copies arbitrary CPU-addressable memory into its own buffer, disclosing memory it is not authorized to read. The write path additionally allows programming flash at an out-of-range address and invalidating an attacker-chosen cache range, affecting integrity and availability. Reachability requires userspace to be enabled and the raw flash device object to be granted to an untrusted thread.

The fix replaces the check with `qspi_nor_range_is_valid()`, which rejects negative offsets and performs the bound comparison in overflow-safe 64-bit arithmetic on both paths, and additionally adds an SRAM DMA bounce buffer plus source/destination overlap rejection to prevent a separate DMA bus-hang condition.

- Zephyr project bug tracker GHSA-c6wh-gwg4-fj5j

This has been fixed in main for v4.5.0

- PR 107793 fix for main
- PR 112433 fix for v4.4
- PR 112434 fix for v4.3

CVE 2026-11809

UpdateHub probe: uninitialized-heap out-of-bounds read of network-supplied metadata

The UpdateHub OTA client in `subsys/mgmt/updatehub/updatehub.c` contains an out-of-bounds / uninitialized-memory read in `z_impl_updatehub_probe()`. The probe response from the UpdateHub server is copied into a heap buffer (`metadata`) that is correctly NUL-terminated, but a second buffer (`metadata_copy`) is allocated with `k_malloc` (unzeroed) and filled with `memcpy(metadata_copy, metadata, strlen(metadata))`, which omits the terminating NUL. Everything after the copied content remains uninitialized heap.

When the first `json_obj_parse()` over the array descriptor fails, the code falls back to `json_obj_parse(metadata_copy, strlen(metadata_copy), ...)`. The `strlen()` call scans past the copied bytes through uninitialized heap and, if no zero byte is found before the end of the allocation, reads beyond the buffer; the resulting over-long length is then parsed as JSON. The probe payload is fully controlled by the (malicious, compromised, or — without the optional `CONFIG_UPDATEHUB_DTLS` — on-path) UpdateHub server, which can craft a large payload that fails the first parse to drive this path.

The consequence is a read of uninitialized heap, with a worst case of an out-of-bounds read past the `metadata_copy` allocation that can fault and crash the update thread/device, producing a network-triggerable denial of service. The over-read data is consumed only internally to evaluate the update and is not returned to the attacker, so there is no direct information disclosure and no out-of-bounds write.

The fix zeroes `metadata_copy` with `memset` before the copy, guaranteeing NUL termination and bounding `strlen()` within the allocation.

- Zephyr project bug tracker GHSA-6r86-hvv2-h6g4

This has been fixed in main for v4.5.0

- PR 104704 fix for main
- PR 112445 fix for v4.4
- PR 112444 fix for v4.3
- PR 112443 fix for v3.7

CVE 2026-11810

NULL-pointer dereference in UpdateHub OTA agent on empty inner metadata array (remote DoS)

The UpdateHub firmware-update agent's probe handler (`z_impl_updatehub_probe()` in `subsys/mgmt/updatehub/updatehub.c`) parses the JSON metadata returned by the update server into a fixed two-level nested-array struct. After parsing it validates only the outer array length (`objects_len != 2`) and then dereferences `objects[1].objects[0].objects.sha256sum` via `strlen()` without checking that the inner object array of element `[1]` is non-empty.

The metadata is attacker-influenceable network input: the agent fetches it over CoAP from the configured UpdateHub server during its routine OTA probe. A malicious or compromised update server (or, when DTLS is disabled, a network man-in-the-middle) can return a response whose second outer object array is empty. Because the parse target is zero-initialised, the corresponding `objects[1].objects[0].objects.sha256sum` pointer is NULL, and the subsequent `strlen()` dereferences address zero. The same defect exists in both the 'any boards' and 'some boards' metadata layouts.

The resulting CPU fault is fatal under Zephyr's default error handling, halting or resetting the device, so the flaw is a remotely triggerable denial of service. Impact is limited to availability; it is a read from NULL with no out-of-bounds write, memory corruption, or information disclosure. The fix rejects metadata whose inner object array is empty before any dereference, on both layouts.

- Zephyr project bug tracker GHSA-jfpc-324j-84ww

This has been fixed in main for v4.5.0

- PR 104704 fix for main
- PR 112445 fix for v4.4
- PR 112444 fix for v4.3
- PR 112443 fix for v3.7

CVE 2026-11811

Socket file-descriptor leak in UpdateHub OTA client start_coap_client() leading to resource-exhaustion DoS

The UpdateHub over-the-air update client's `start_coap_client()` in `subsys/mgmt/updatehub/updatehub.c` leaks the CoAP/DTLS socket descriptor on its connection-setup failure paths. The shared `error:` cleanup gated socket closing on a `ret > 0` flag, but `ret` was set to `-1` immediately after the socket was created, so when `zsock_setsockopt()` (DTLS) or `zsock_connect()` subsequently failed the gate was false and `cleanup_connection()` was never called. The open descriptor in the global `ctx.sock` was then overwritten by the next attempt, permanently leaking it from the socket / net_context pool until reboot.

The failing setup path is reached every time the OTA client tries to contact the UpdateHub server and the connection cannot be established — driven automatically by the periodic `autohandler()` poll (and on demand via the `updatehub_probe()`/`updatehub_update()` API or the `updatehub run` shell command). The DTLS handshake/connect outcome is influenceable by a network or on-path attacker who drops, resets, or otherwise disrupts traffic to the server, and also fails naturally whenever the server is unreachable.

Each failed attempt permanently leaks one descriptor; once the shared socket pool is exhausted, networking degrades device-wide until the device is rebooted, a denial-of-service condition. Severity is low because the leak rate is bounded by the configured OTA poll interval (default once per 24 hours), the effect is gradual and recovered by reboot, and only builds with the UpdateHub client enabled are affected. There is no memory-corruption, information-disclosure, or authentication impact.

- Zephyr project bug tracker GHSA-q3mh-4wj7-mq7f

This has been fixed in main for v4.5.0

- PR 104704 fix for main

- PR 112445 fix for v4.4
- PR 112444 fix for v4.3
- PR 112443 fix for v3.7

CVE 2026-11812

UpdateHub: race condition on shared context causes out-of-bounds write and DoS

The UpdateHub management subsystem (subsys/mgmt/updatehub/updatehub.c) drives every update operation through a single file-scope `ctx` structure that holds the CoAP block context, payload buffer, status code, socket, and a one-element poll-fd array `fds[1]`. Access to `ctx` was not serialized, and `prepare_fds()` wrote `ctx.fds[ctx.nfds]` and incremented `ctx.nfds` with no bounds check.

Two independent paths mutate `ctx` concurrently: the background autohandler running on the system workqueue, and user-triggered operations reached through the `updatehub run` shell command, direct API calls, or — since the operations are exposed as syscalls — userspace threads. When a second flow enters `prepare_fds()` while `ctx.nfds` is already 1, the write lands one element past the array; by struct layout it overlaps the adjacent `ctx.sock`/`ctx.nfds` members. More broadly, the unsynchronized sharing lets two flows interleave connection setup and teardown, double-closing a socket descriptor or scribbling the shared buffers.

The result is corruption of the update subsystem's internal state and denial of service of the firmware-update path; the out-of-bounds write is contained within the `ctx` structure and there is no demonstrated path to memory outside it or to code execution. Triggering requires a local actor able to invoke update operations (or, with CONFIG_USERSPACE, an unprivileged userspace thread) and to win a timing race against the background handler; remote peers cannot control the race timing. The fix serializes the entry points with a mutex and adds a bounds check to `prepare_fds()`.

- Zephyr project bug tracker GHSA-vprh-rff6-46xp

This has been fixed in main for v4.5.0

- PR 104704 fix for main
- PR 112445 fix for v4.4
- PR 112444 fix for v4.3
- PR 112443 fix for v3.7

CVE 2026-11893

Double free / use-after-free in Bouffalo Lab HCI driver send() error paths (hci_bflb)

The Bluetooth HCI driver for Bouffalo Lab on-chip BLE controllers (BL60x/BL70x/BL61x), `bt_bflb_send()` in `drivers/bluetooth/hci/hci_bflb.c`, violates the `bt_hci_driver_api.send()` buffer-ownership contract. That contract (documented at `include/zephyr/drivers/bluetooth.h`) requires the buffer reference to be consumed only on success; on error the caller still owns the reference and unrefs it. The driver instead routed all error paths through a shared label that unconditionally called `net_buf_unref(buf)` before returning the error code, consuming the buffer on failure as well.

When `send()` returns an error, the host TX path (`send_buf()` in `subsys/bluetooth/host/conn.c`) unrefs the same buffer again, believing it still owns it. This double-unref over-decrements the net_buf reference count. Because the buffer is a TX fragment whose destroy callback also decrements its still-queued parent buffer, the parent is freed prematurely while reachable on the connection TX queue, producing a use-after-free and corruption of the shared net_buf pool rather than a benign leak.

The error conditions are on the host-to-controller transmit path (controller send failure, or an unsupported H:4 packet type), so they are not driven directly by attacker-supplied radio bytes; a

remote/adjacent peer can influence them only indirectly, e.g. by inducing controller TX failures under heavy link load. The consequence when reached is BLE-stack denial of service (crash / pool corruption) with possible further memory corruption, bounded to devices using one of these Bouffalo Lab on-chip controllers.

- Zephyr project bug tracker GHSA-ph42-6rqx-728c

This has been fixed in main for v4.5.0

- PR 110711 fix for main
- PR 112558 fix for v4.4

CVE 2026-11894

Double-free / use-after-free in Realtek BEE Bluetooth HCI driver `send()` error paths

The Realtek BEE Bluetooth HCI driver's send callback, `bt_hci_bee_send()` in `drivers/bluetooth/hci/hci_bee.c`, violated the `bt_hci_driver_api` buffer-ownership contract. That contract requires the driver to consume (unref) the transmit `net_buf` only on success; on an error return the host caller retains ownership and unrefs the buffer itself. The pre-fix code routed all error paths through a shared cleanup label that unconditionally called `net_buf_unref(buf)` before returning the error code.

Because the host TX paths (in `subsys/bluetooth/host/hci_core.c`) unref the buffer again after `send()` returns an error, the buffer is freed twice: the driver returns it to its `net_buf` pool and the host then unrefs the already-freed buffer, corrupting the shared pool / underflowing the reference count (CWE-415). The same error branch additionally dereferenced `buf->len` inside a `LOG_ERR` call after the buffer had already been unref'd, a read of freed memory (CWE-416) that is compiled in at the default error log level.

The failing edges are reached when the controller's host-to-controller buffer allocation fails or the controller send fails (resource-exhaustion / IO conditions). A remote Bluetooth peer can push the device toward these conditions indirectly by driving heavy host transmit activity, at which point the double-free corrupts the host `net_buf` pool and most likely crashes the device, with residual potential for further memory corruption. The impact is confined to builds using this specific Realtek BEE HCI driver.

The fix returns early from each error path without unreffing and unrefs the buffer only on the success path, restoring the ownership contract and eliminating both the double-free and the use-after-free read.

- Zephyr project bug tracker GHSA-v9mj-h2m6-v9c6

This has been fixed in main for v4.5.0

- PR 110711 fix for main
- PR 112558 fix for v4.4

CVE 2026-11985

Cross-thread FPU register leak on ARM when FPU enabled without register sharing

On the Zephyr ARM port, enabling the hardware FPU (`CONFIG_FPU`) forces the "Floating point ABI" choice, which defaults to `CONFIG_FP_HARDABI`. Both `FP_HARDABI` and `FP_SOFTABI` permit the compiler to emit hardware FP instructions in any function, even code that never uses floating-point types. However, the callee-saved FP registers (s16-s31 / d8-d15) are only saved and restored across a context switch when `CONFIG_FPU_SHARING` is enabled (`arch/arm/core/cortex_m/swap_helper.S` and `arch/arm/core/cortex_a_r/swap_helper.S`), and prior to this fix selecting an ABI did not enable FPU register sharing, which defaults off.

In a build that enables the FPU with the default ABI but leaves `CONFIG_FPU_SHARING` disabled, the kernel preserves no callee-saved FP register state across thread switches. The documented

precondition for this "unshared" mode — that only a single thread ever executes FP instructions — is silently violated because the compiler may generate FP instructions in every thread.

Under CONFIG_USERSPACE, where threads are mutually isolated, this becomes an information-disclosure boundary crossing: a victim thread can leave secret-derived values in s16-s31, and a co-resident unprivileged thread can read those registers directly (FP register access is not privilege-gated), recovering data left behind by another thread. Without userspace the same defect causes cross-thread FP state corruption (a correctness fault). The leak is bounded to the 16 callee-saved single-precision registers and is opportunistic, so impact is low.

The fix makes FP_HARDABI and FP_SOFTABI select CONFIG_FPU_SHARING and tags every thread with K_FP_REGS at creation, so callee-saved FP state is always preserved across context switches whenever the compiler may emit FP instructions.

- Zephyr project bug tracker GHSA-qxr9-wh3c-hvgv

This has been fixed in main for v4.5.0

- PR 110300 fix for main
- PR 112547 fix for v4.4
- PR 112551 fix for v4.3
- PR 112550 fix for v3.7

CVE 2026-12051

NULL pointer dereference in USB DFU device_next download handler (handle_download)

The USB DFU class implementation in Zephyr's new (experimental) device_next USB device stack contains a NULL pointer dereference in handle_download() (subsys/usb/device_next/class/usbd_dfu.c). The handler computes MIN(setup->wLength, buf->len) and passes buf->data to the image write callback without checking that the buf net_buf pointer is non-NULL.

The handler is reached over the USB control endpoint, driven by the USB host. For a DFU_DNLOAD (download) request with no Data OUT stage — notably the zero-length terminating download that the DFU protocol uses to end a firmware transfer — the USB core invokes the class handler with a NULL buffer. After the device has been advanced to the DFU_DNLOAD_IDLE state (by sending one valid download block and a GET_STATUS), a zero-length DFU_DNLOAD reaches handle_download() with buf == NULL, dereferencing it.

The result is a NULL+offset read that triggers a fatal CPU fault, i.e. a denial of service (device crash/reset). The attacker is whatever controls the USB host the device is attached to; DFU download support must be enabled with a registered image. There is no memory corruption or information disclosure — impact is limited to availability. The fix adds an explicit if (buf != NULL) guard so the callback receives a zero-length, NULL-data transfer instead of crashing.

- Zephyr project bug tracker GHSA-vhvq-q6rw-jvm4

This has been fixed in main for v4.5.0

- PR 110830 fix for main
- PR 112560 fix for v4.4

CVE 2026-12052

Out-of-bounds write in USB CDC NCM control handler when host wLength is smaller than the response

The USB device-side CDC NCM class control-to-host handler usbd_cdc_ncm_cth in subsys/usb/device_next/class/usbd_cdc_ncm.c builds a fixed-size response for the GET_NTB_PARAMETERS (28-byte struct ntb_parameters) and GET_NTB_INPUT_SIZE (8-byte struct ntb_input_size)

class requests and copies the whole structure into the control DATA IN buffer with `net_buf_add_mem(buf, ..., sizeof(...))`, ignoring the host-supplied `wLength`.

The control DATA IN buffer is allocated by the USB stack with a capacity of exactly `wLength` bytes (`usbd_ep_ctrl_data_in_alloc -> udc_ctrl_data_alloc -> net_buf_alloc_len(&udc_ep_pool, wLength)`; no round-up is applied for the IN endpoint). Because `net_buf_add_mem`/`net_buf_simple_add` only bounds the copy with an `__ASSERT_NO_MSG`, which is compiled out in production builds, a host that issues one of these standard CDC NCM control requests with a `wLength` smaller than the response structure (e.g. `wLength = 1`) causes the handler to `memcpy` up to 27 bytes past the end of the allocated pool buffer.

The request fields come straight from the USB SETUP packet, so any host (or USB interposer) the Zephyr device enumerates against can trigger the overflow with no authentication once an image built with the device_next USB stack and the CDC NCM class is connected. The out-of-bounds write corrupts adjacent allocations and metadata in the shared `udc_ep_pool`, primarily causing memory corruption and denial of service of the USB stack; the overflow length is bounded (<= 27 bytes) and the written content is fixed device constants, and the bug reads nothing back so there is no information disclosure. The fix clamps the copy with `MIN(sizeof(...), setup->wLength)`, matching the existing CDC ACM handler.

- Zephyr project bug tracker GHSA-vr4p-6rg5-qgpx

This has been fixed in main for v4.5.0

- PR 110831 fix for main
- PR 112615 fix for v4.4
- PR 112614 fix for v4.3

CVE 2026-12232

Out-of-bounds read via unvalidated stream_id in Intel ALH DAI get_properties

The Intel ALH digital-audio-interface driver function `dai_alh_get_properties()` in `drivers/dai/intel/alh/alh.c` used a caller-supplied `int stream_id` with no range validation. The value indexes the fixed-size `static const uint8_t alh_handshake_map[64]` array and scales a FIFO register address, so an out-of-range `stream_id` produces an out-of-bounds read of one byte at an attacker-chosen signed offset from the array. That byte is written into `prop->dma_hs_id` and the resulting `struct dai_properties` is copied back to the caller, leaking it.

`dai_get_properties_copy()` is a Zephyr `__syscall`, and its verifier `z_vrfy_dai_get_properties_copy()` (`drivers/dai/dai_handlers.c`) validates only the device-object permission and the destination buffer, not `stream_id`. A user-mode thread that has been granted access to the ALH DAI device object can therefore call the syscall with an arbitrary `stream_id`, crossing the userspace/kernel sandbox boundary.

The impact is a one-byte-per-call arbitrary-offset kernel information disclosure (and leakage of a computed kernel address via `fifo_address`); a `stream_id` that resolves to an unmapped page faults in kernel context, giving a local denial of service. Exploitation requires `CONFIG_USERSPACE` and device access, making this a local, moderate-severity issue. The fix rejects negative and too-large `stream_id` values up front and returns NULL, which the copy wrapper maps to `-ENOENT`.

- Zephyr project bug tracker GHSA-3557-j848-pv24

This has been fixed in main for v4.5.0

- PR 110946 fix for main
- PR 112747 fix for v4.4
- PR 112745 fix for v3.7

CVE 2026-12233

Uninitialized mutex in TLS trusted-credential backend causes kernel NULL-deref DoS under contention

The PSA Protected Storage credential backend (subsys/net/lib/tls_credentials/tls_credentials_trusted.c) declared its credential-store mutex as a plain zero-filled `static struct k_mutex credential_lock;` and never called `k_mutex_init()` on it. A statically zero-filled `k_mutex` has an uninitialized wait queue (its dlist head/tail are NULL instead of the self-referential sentinels that `k_mutex_init`/`K_MUTEX_DEFINE` install). The uncontended lock path does not touch the wait queue, so the defect is latent and serialized use behaves correctly.

When two execution contexts contend on the lock, `k_mutex_lock()` pends the blocking thread on the wait queue via `z_pend_curr()`, which calls `sys_dlist_append()` on the zeroed list and dereferences a NULL tail pointer (`tail->next = node`), faulting the kernel. The lock is held during TLS handshake credential loading and by all credential add/get/delete operations, so a deployment performing concurrent TLS handshakes (for example a server handling multiple simultaneous connections from a remote peer) or a credential-management operation concurrent with a handshake can trigger the dereference.

The impact is a denial of service: a deterministic kernel panic / device reset on the first contention. There is no memory corruption beyond the NULL dereference and no confidentiality or integrity impact; mutual exclusion on the fast path remains correct. Exposure is limited to builds with `CONFIG_TLS_CREDENTIALS_BACKEND_PROTECTED_STORAGE` enabled (PSA Protected Storage / TF-M platforms); the default volatile RAM backend initializes its lock correctly and is unaffected.

The fix initializes the mutex statically with `K_MUTEX_DEFINE(credential_lock)`, providing a valid wait queue so the contended path no longer touches a NULL list.

- Zephyr project bug tracker GHSA-57c4-xcq2-fqj7

This has been fixed in main for v4.5.0

- PR 110943 fix for main
- PR 112741 fix for v4.4
- PR 112740 fix for v4.3
- PR 112739 fix for v3.7

CVE 2026-12234

TOCTOU double-fetch in `zsock_sendmsg`/`recvmsg` userspace verifiers allows kernel-heap out-of-bounds write

The userspace syscall verifiers `z_vrfy_zsock_sendmsg()` and `z_vrfy_zsock_recvmsg()` in `subsys/net/lib/sockets/sockets.c` snapshot the caller-supplied `struct net_msghdr` into a kernel-side copy with `k_usermode_from_copy()`, but then re-read the still-live user struct for subsequent decisions. The kernel `iovec` shadow buffer is sized from one read of `msg->msg_iovlen`, while the population loop is bounded by a second, live read of the same field.

Because `msg` points into ordinary user memory, a cooperating second thread in the same memory domain can inflate `msg->msg_iovlen` in the window between the sizing read and the loop test (a classic double-fetch / TOCTOU). The population loop then iterates past the number of `net_iovec` slots actually allocated, writing attacker-influenced `iov_base`/`iov_len` values beyond the end of the kernel-heap shadow buffer. The `recvmsg` verifier has the same defect on both its inbound and result write-back loops.

The code is reachable from an unprivileged user thread whenever `CONFIG_USERSPACE` is enabled and the `zsock_sendmsg`/`zsock_recvmsg` syscalls are available. A successful race corrupts kernel-managed heap memory across the user-to-kernel privilege boundary, yielding a local privilege-escalation primitive or, at minimum, a kernel-fault denial of service. The fix copies the header

once and derives every size, bound, and gate from the snapshot, copying each `iovec` entry atomically so its base and length can no longer be raced apart.

- Zephyr project bug tracker GHSA-fcp3-vrr2-xfjv

This has been fixed in main for v4.5.0

- PR 108079 fix for main
- PR 112619 fix for v4.4
- PR 112624 fix for v4.3
- PR 112625 fix for v3.7

CVE 2026-12235

Out-of-bounds write in Xtensa llext PLT relocation from malformed ELF (CWE-787)

The Linkable Loadable Extensions (llext) subsystem mis-handles PLT/RELA relocation entries when linking a relocatable (partially-linked) ELF extension. In `llext_link_plt()` (subsys/llext/llext_link.c), the relocatable branch (`tgt != NULL`, the path used for Xtensa relocatable objects) computed the patch address as `ext->mem[LLEXT_MEM_TEXT] - text.sh_offset + rela.r_offset + tgt->sh_offset` and then performed the relocation write there without validating `rela.r_offset`. Its sibling shared/dynamic branch already rejected out-of-range offsets via `llext_file_offset()`.

`rela.r_offset` is read directly from the ELF's RELA table, so a crafted entry with an offset larger than the target section makes the write land arbitrarily far outside the extension's text buffer. The result is an attacker-influenced out-of-bounds write (the location via `r_offset`, the written value being the resolved symbol address) performed in supervisor context at link time, before any extension code runs.

The path is reached from `llext_load()` whenever an application loads an attacker-influenced ELF extension on Xtensa with writable storage; llext is documented to accept extensions of untrusted origin. Impact is supervisor-context memory corruption (integrity and availability loss, and a sandbox-boundary escape for user-mode extensions). Exploitation is gated by the Xtensa relocatable PLT path and writable storage, and turning the out-of-range write into a useful primitive is non-trivial.

The fix adds a bound check rejecting any RELA entry whose `r_offset >= tgt->sh_size`, mirroring the existing validation in the shared branch.

- Zephyr project bug tracker GHSA-xv9q-6mrf-8j49

This has been fixed in main for v4.5.0

- PR 109875 fix for main
- PR 112635 fix for v4.4
- PR 112634 fix for v4.3
- PR 111541 fix for v3.7

CVE 2026-12236

Infinite loop (DoS) in Bluetooth GATT client parsing of Read-By-Type responses with zero data length

The Bluetooth host GATT client function `parse_read_std_char_desc()` in `subsys/bluetooth/host/gatt.c` parses an ATT Read By Type Response received from a remote GATT server during `BT_GATT_DISCOVER_STD_CHAR_DESC` discovery. The per-entry stride `rsp->len` is taken directly from the peer's PDU, and the parse loop both tests its exit condition (`length >= rsp->len`) and advances (`length -= rsp->len`, `pdu += rsp->len`) using that value. The minimum value of `rsp->len` was never validated before the loop.

A malicious or malfunctioning peer can reply with `rsp->len = 0`. Because `length` is unsigned and never decreases, the loop condition stays true forever and the read pointer never advances; as long as the body is at least a few bytes with a non-zero handle and a matching descriptor UUID, the host repeatedly re-parses the same bytes and invokes the discovery callback, never terminating. This hangs the Bluetooth host processing thread (CWE-835, loop with unreachable exit condition).

The condition is reachable by any connected peer once the local device initiates standard-descriptor-value discovery; GATT discovery does not require bonding or encryption, so an unauthenticated adjacent attacker that the device connects to can trigger it. The impact is denial of service of the Bluetooth subsystem (and likely a watchdog reset on constrained targets); there is no memory disclosure or corruption.

The fix adds a `rsp->len < sizeof(struct bt_att_data)` check before the loop, rejecting under-length responses so the stride is always non-zero and the loop terminates. The sibling parsers `parse_include()` and `parse_characteristic()` already validated `rsp->len` and are unaffected.

- Zephyr project bug tracker GHSA-483r-jq2x-5cp9

This has been fixed in main for v4.5.0

- PR 109066 fix for main
- PR 112840 fix for v4.4
- PR 112839 fix for v4.3
- PR 112841 fix for v3.7

CVE 2026-7007

Division by zero in Zephyr ext2 superblock parsing allows DoS via crafted filesystem image

The Zephyr ext2 file system validates the on-disk superblock in `ext2_verify_disk_superblock()` (`subsys/fs/ext2/ext2_impl.c`) before completing a mount. The validator checked the magic number, block size, revision and feature flags, but did not verify that the on-disk fields `s_blocks_per_group` and `s_inodes_per_group` are non-zero. Both fields are read directly from the image and are later used as divisors during mount-time initialization.

During mount, `get_ngroups()` divides and modulos `s_blocks_count` by `s_blocks_per_group` (reached via `ext2_fetch_block_group()` from `ext2_init_fs()`), and `get_itable_entry()` divides `(ino - 1)` by `s_inodes_per_group` when fetching the root inode (both in `subsys/fs/ext2/ext2_diskops.c`). A superblock with either field set to zero therefore causes an integer division by zero during the mount sequence.

An attacker who can present a crafted ext2 image to a device that mounts ext2 — removable media such as an SD card or a USB mass-storage device — can trigger this. On ARMv7-M / ARMv8-M-mainline Cortex-M targets, divide-by-zero trapping is enabled (`SCB_CCR_DIV_0_TRP`), so the division raises a UsageFault that Zephyr treats as a fatal error, producing a denial of service. The impact is limited to availability; the malformed value is consumed only as a divisor.

The fix rejects a zero `s_blocks_per_group` or `s_inodes_per_group` in the superblock validator, returning `-EINVAL` so the mount fails before any block-group or inode I/O occurs.

- Zephyr project bug tracker GHSA-wrf2-79mm-cvw5

This has been fixed in main for v4.5.0

- PR 107929 fix for main
- PR 113331 fix for v4.4
- PR 110884 fix for v4.3
- PR 110883 fix for v3.7

CVE 2026-8023

Path traversal in Zephyr HTTP server static-filesystem resource handler allows unauthenticated remote arbitrary file read

Zephyr's HTTP server (`subsys/net/lib/http`) provides a static-filesystem resource type (`HTTP_RESOURCE_TYPE_STATIC_FS`, available when `CONFIG_FILE_SYSTEM` is enabled) that serves files from a configured root directory. Before this fix, both the HTTP/1 and HTTP/2 front-ends placed the raw, attacker-controlled request path into `client->url_buffer` (assembled in `on_url()` for HTTP/1 and copied verbatim from the `:path` pseudo-header for HTTP/2) without resolving `./..` segments. The static-FS handler then built the on-disk filename by directly concatenating the configured root with that raw URL (`snprintk(fname, ..., "%s%s", static_fs_detail->fs_path, client->url_buffer)` at `http_server_http1.c:603` and `http_server_http2.c:490`) and opened it with `fs_open(fname, FS_O_READ)`. Because the handler is reached via wildcard/leading-dir (`fnmatch FNM_LEADING_DIR`) or fallback resource matching, a request such as `GET /<prefix>/../../<file>` is dispatched to the handler and, after the underlying filesystem (e.g. LittleFS/FAT) resolves the `..` segments, escapes the configured web root, letting an unauthenticated remote client read arbitrary readable files on the mounted volume (information disclosure). The HTTP server requires no TLS or authentication to reach this path. The fix adds `http_server_remove_dot_segments()`, which canonicalizes the path portion of the URL before resource lookup in both protocol handlers, neutralizing the traversal. Affects releases v4.0.0 through v4.4.0 for deployments that register a static-filesystem resource.

- Zephyr project bug tracker GHSA-hch3-53g6-jj3h

This has been fixed in main for v4.5.0

- PR 108531 fix for main
- PR 111347 fix for v4.4
- PR 111346 fix for v4.3

CVE 2026-9728

TOCTOU race in mbox_send syscall verifier allows userspace to leak kernel memory

The userspace syscall verifier `z_vrfy_mbox_send()` in `drivers/mbox/mbox_handlers.c` validated the nested `msg->data`/`msg->size` fields by reading them directly out of live userspace memory, and then forwarded the original, still-mutable userspace `struct mbox_msg *` pointer to `z_impl_mbox_send()` and the underlying driver. Between the access check and the driver's use of `msg->data`, the validated pointer could be replaced, leaving a time-of-check/time-of-use window.

On a system built with `CONFIG_USERSPACE`, any unprivileged userspace thread may invoke the `mbox_send()` system call. A second thread sharing the caller's address space can race to overwrite `msg->data` with a supervisor (kernel) address after the verifier's bounds check has passed but before the driver dereferences it. The driver then reads from the attacker-chosen address in supervisor context (for example `memcpy(&data32, msg->data, msg->size)` in the NXP mailbox driver, whose bytes are subsequently emitted to the peer mailbox endpoint).

The impact is a userspace-to-supervisor access-control bypass: disclosure of kernel memory contents (high confidentiality impact), or, for an invalid/unmapped target address, a faulting kernel read causing denial of service. The fix snapshots the entire `struct mbox_msg` into a kernel-stack copy with `k_usermode_from_copy()` and validates and forwards that immutable copy, closing the race.

- Zephyr project bug tracker GHSA-47q2-w832-7w67

This has been fixed in main for v4.5.0

- PR 109946 fix for main
- PR 110657 fix for v4.4

- PR 110656 fix for v4.3
- PR 113308 fix for v3.7

CVE 2026-9771

Missing device-pointer validation in flash_copy() syscall allows userspace privilege escalation

The `flash_copy()` system call is verified by `z_vrfy_flash_copy()` in `drivers/flash/flash_util.c`. On builds with `CONFIG_USERSPACE` enabled, this handler is the kernel-side trust boundary for a user-mode caller. Prior to the fix it validated only the output buffer (`K_SYSCALL_MEMORY_WRITE`) and passed the two `struct device *` arguments, `src_dev` and `dst_dev`, directly into the implementation without any object validation — unlike every sibling flash syscall, which guards its device pointer with `K_SYSCALL_DRIVER_FLASH`.

A user-mode thread fully controls the values of `src_dev`/`dst_dev` and the contents of its own address space. The implementation `z_impl_flash_copy()` dereferences these pointers and calls through their driver-API function tables (e.g. `api->get_parameters(dst_dev)`, `flash_read(src_dev, ...)`, `flash_write(dst_dev, ...)`). By supplying a pointer to a forged `struct device` whose `api` table contains attacker-chosen function pointers, an unprivileged thread can cause the kernel to call arbitrary code in supervisor mode; passing any arbitrary or invalid address otherwise yields a kernel crash or out-of-bounds read.

The result is a local privilege escalation out of the userspace sandbox (with kernel denial-of-service and information disclosure as lesser outcomes). The fix adds `K_SYSCALL_DRIVER_FLASH(src_dev, read)` and `K_SYSCALL_DRIVER_FLASH(dst_dev, write)` to `z_vrfy_flash_copy()`, which verify each device is a registered flash-driver kernel object the calling thread is permitted to use before any dereference, closing the path completely.

- Zephyr project bug tracker GHSA-68cj-3hg4-5vpm

This has been fixed in main for v4.5.0

- PR 109962 fix for main
- PR 110874 fix for v4.4
- PR 110873 fix for v4.3

CVE 2026-12363

Out-of-bounds write in LoRaWAN fragmented transport from a fragment index of 0

The LoRaWAN Fragmented Data Block Transport service (`subsys/lorawan/services/frag_transport.c`) does not validate the fragment counter in a received `DATA_FRAGMENT` command before forwarding it to the configured decoder. In `frag_transport_package_callback()` the value `frag_counter = hdr->frag_index_n & 0x3FFF` is taken directly from the downlink payload and passed to the decoder, which derives an array index and flash offset as `frag_counter - 1`. DataFragment fragments are 1-indexed, so a `frag_counter` of `0` underflows that arithmetic.

With the default Semtech/LoRaMAC-node decoder, this reaches `FragDecoder.FragNbMissingIndex[fragCounter - 1] = 0`; in `FragDecoderProcess()`, where `fragCounter - 1` evaluates to `-1` and writes a `uint16_t` zero out of bounds, just before the array and into the adjacent `MatrixM2B` recovery-matrix state of the static decoder object (`CWE-787`). A companion write derives a wild flash offset, but that path is rejected by the `flash_area_write()` bounds check. The in-tree low-memory decoder (`frag_dec()`) is not corrupted: its out-of-range bit-array and flash accesses are caught by `sys_bitarray_*` and `flash_area_*` bounds checks.

The handler is the registered downlink callback for the fragmentation transport port, reachable whenever an active fragmentation session exists, so the triggering byte is attacker-influenceable LoRaWAN/FUOTA network input. Triggering it requires authenticated downlinks (LoRaWAN MAC session keys or a malicious/compromised network or FUOTA server) and an active fragmentation session. The impact is contained: corruption of decoder state and denial of the firmware-update (FUOTA) session rather than controllable memory corruption or code execution. The fix

adds a transport-layer check that rejects `frag_counter == 0`, closing the defect for both decoder backends.

- Zephyr project bug tracker GHSA-fvm7-7whg-8gj6

This has been fixed in main for v4.5.0

- PR 111287 fix for main
- PR 112928 fix for v3.7

CVE 2026-12364

Missing user-space pointer validation in logging syscall z_log_msg_static_create allows kernel memory disclosure and denial of service

The user-space system-call verifier `z_vrfy_z_log_msg_static_create()` in `subsys/logging/log_msg.c` was a pure pass-through: it forwarded the caller-supplied `source`, `desc`, `package`, and `data` arguments directly to the kernel-mode implementation `z_impl_z_log_msg_static_create()` without performing any of the mandatory `K_SYSCALL_*` checks. Because `z_log_msg_static_create()` is declared `__syscall`, under `CONFIG_USERSPACE` any unprivileged user-mode thread can invoke it directly with fully attacker-controlled arguments.

The kernel-mode handler dereferences each of these untrusted values: `frontend_runtime_filtering()` reads through the `source` pointer as a `struct log_source_dynamic_data`, `cbprintf_package_copy()` reads `desc.package_len` bytes from the `package` pointer, and `z_log_msg_finalize()` performs a `memcpy()` of `desc.data_len` bytes from the `data` pointer. With no verification, a user thread can supply arbitrary kernel addresses and arbitrary lengths, and the kernel will read from them.

The impact is a kernel-mode denial of service (the kernel faults dereferencing an attacker-chosen pointer) and, where a log backend output is observable to the attacker, disclosure of arbitrary kernel memory copied into the emitted log message — a confidentiality breach across the user/kernel boundary that the userspace sandbox is meant to enforce. The reads do not corrupt kernel memory, so there is no out-of-bounds write primitive.

The fix adds the required validation to the verifier: it bounds `desc.package_len` against `Z_LOG_MSG_MAX_PACKAGE`, rejects non-NULL/length mismatches, and applies `K_SYSCALL_MEMORY_READ()` to `package`, `data`, and (when runtime filtering with a frontend is enabled) `source`, so any out-of-bounds or kernel pointer now raises `K_OOPS` instead of being honored.

- Zephyr project bug tracker GHSA-h7rf-g9mg-g23f

This has been fixed in main for v4.5.0

- PR 110506 fix for main
- PR 112853 fix for v4.4
- PR 112854 fix for v4.3
- PR 116338 fix for v3.7

CVE 2026-12365

Use-after-free in Zephyr delayable work-queue cancellation under SMP timing race

A use-after-free exists in the Zephyr second-generation work queue (`kernel/work.c`) in the handling of delayable work timeouts. When a delayable work item's timeout has been dequeued and its handler `work_timeout()` is in flight (blocked acquiring the work-queue spinlock), a concurrent cancellation does not wait for that handler to finish. In `unschedule_locked()` the pre-fix code called `z_abort_timeout()`, which for an already-announcing record returns `-EINVAL` without removing it; `cancel_async_locked()` then observes the work as idle, so even

k_work_cancel_delayable_sync() and k_work_flush_delayable() return without blocking on the in-flight handler.

Because those are the APIs the kernel header documents as the safe way to cancel before freeing a k_work_delayable, a caller that frees the object immediately after a successful sync cancel can race the still-pending handler. work_timeout() subsequently dereferences the freed record: it reads to->dticks via z_is_timeout_handler_canceled() and, if the freed slot has been reused so the bail check fails, performs a read-modify-write of wp->flags (K_WORK_DELAYED_BIT) and submits work against a stale dw->queue pointer — a use-after-free read and write.

The k_work API is kernel-mode only (no __syscall entry point), so this is a kernel-internal concurrency defect rather than a userspace privilege escalation. Triggering it requires an SMP build and a subsystem that schedules and then frees (or reschedules) a delayable work item in the narrow window while its timeout is announcing; an attacker able to influence the timing of such teardown (for example via connection churn driving subsystem timers) has a plausible but probabilistic path. The impact is kernel memory corruption or crash (denial of service).

The fix makes unschedule_locked() wait, by spinning on z_try_abort_timeout() returning -EAGAIN while releasing and re-acquiring the work spinlock, until any in-flight handler completes before returning, and switches work_timeout() to atomic K_WORK_DELAYED_BIT ownership. This closes both the free-then-handler use-after-free and the related reschedule early-fire race.

- Zephyr project bug tracker GHSA-rhmh-r93p-6g99

This has been fixed in main for v4.5.0

- PR 109977 fix for main
- PR 112961 fix for v4.4

CVE 2026-12366

Use-after-free freeing an armed dynamically-allocated k_timer in Zephyr userspace object disposal

Zephyr's dynamic kernel-object disposal path unref_check() in kernel/userspace/userspace.c frees an object's storage (k_free(dyn->data)) once its reference count reaches zero, after running a per-object-type cleanup. The cleanup switch handled only K_OBJ_MSGQ and K_OBJ_STACK; there was no K_OBJ_TIMER case. A dynamically-allocated, initialized, and armed k_timer keeps its embedded struct _timeout dnode linked in the global timeout queue (_timeout_q), so freeing the timer storage without cancelling the timeout leaves a dangling node in that queue.

When the timer next expires, the timeout machinery walks _timeout_q and invokes z_timer_expiration_handler() on the freed node, dereferencing and writing freed (and reusable) kernel heap in kernel/ISR context. This is a deterministic use-after-free that does not depend on SMP: the queued node is simply never unlinked at free time.

The disposal is reachable from an unprivileged user thread under CONFIG_USERSPACE + CONFIG_DYNAMIC_OBJECTS: a thread that holds the last permission on such a timer drops it via the k_object_release() syscall (or by exiting, through k_thread_perms_all_clear()), and can arm the timer itself via the k_timer_start() syscall. The free and the expiration handler run at kernel privilege while the actor is a user thread, so the bug is a sandbox-escape memory-corruption primitive usable for privilege escalation. The fix adds k_timer_cleanup() (cancel the timeout and wait for any in-flight handler) and calls it for K_OBJ_TIMER before freeing.

- Zephyr project bug tracker GHSA-x96g-542c-gccq

This has been fixed in main for v4.5.0

- PR 109977 fix for main
- PR 112961 fix for v4.4

CVE 2026-12519

Out-of-bounds stack read and write in Zephyr WNC-M14A2A modem socket-notify parsing

The WNC-M14A2A LTE-M modem driver mishandles unsolicited `%NOTIFYEV:` events in `on_cmd_socknotifyev()` (`drivers/modem/vendor_standalone/wncm14a2a.c`). The response line is linearized into a fixed 40-byte stack buffer via `net_buf_linearize()`, which caps the copy at 39 bytes and returns `out_len <= 39`. The two quote-delimiter scanning loops, however, were bounded by `len` — the full CR/LF-delimited frame length returned by `net_buf_findcrlf()` — rather than by `out_len`.

When a `%NOTIFYEV:` line longer than 39 bytes contains no `"` within the linearized region, the loop indices `p1`/`p2` walk past `value[39]` and read adjacent stack memory until a stray quote byte is found or the index reaches `len`. The over-read string is then passed to `strncmp()`/`atoi()`/`LOG_*`, and if a quote byte is found out of bounds the subsequent `value[p2] = '\0'` performs a single-NUL out-of-bounds stack write at an attacker-influenced offset.

The `%NOTIFYEV:` payload carries network-derived content (`LTIME` network time, `SIB1` base-station system information, `CSPS/RRCSTATE`), so a rogue cellular base station, a malicious or compromised modem module, or RF manipulation that induces an over-long notify line reaches the defect without any application interaction; the handler runs automatically on the unsolicited event in the modem RX thread.

The impact is out-of-bounds stack disclosure (into logs and parsing) and stack corruption that can crash the modem RX thread (denial of service). The write offset is only weakly controlled, so memory-safe code execution is not demonstrated. The fix bounds both scanning loops by `out_len`, keeping all accesses within the linearized buffer.

- Zephyr project bug tracker GHSA-8hrc-q8cp-6xhf

This has been fixed in main for v4.5.0

- PR 111243 fix for main
- PR 113040 fix for v4.4
- PR 113042 fix for v4.3
- PR 113041 fix for v3.7

CVE 2026-12520

Stack buffer overflow and off-by-one writes in Zephyr HL7800 modem AT response handlers

The Sierra Wireless HL7800 cellular modem driver (`drivers/modem/vendor_standalone/hl7800.c`, located at `drivers/modem/hl7800.c` in v4.4.0 and earlier) parses AT responses with roughly twenty handlers that call `net_buf_linearize(value, sizeof(value), *buf, 0, len)` into a 128-byte stack buffer and then write `value[out_len] = 0`. Because `net_buf_linearize()` (`lib/net_buf/buf.c`) can return a count equal to its destination-length argument, a field that exactly fills the buffer makes the terminating NUL land one byte past the end, a single-byte out-of-bounds write into adjacent stack memory.

The `+KCELLMEAS` cell-measurement handler `on_cmd_atcmdinfo_rssi()` is worse: it passed the wire length `len` as the destination size (`net_buf_linearize(value, len, *buf, 0, len)`), so a response line longer than 128 bytes overflows the `value` stack buffer with attacker-influenceable content. The line length comes from `net_buf_findcrlf()`, which accumulates bytes across the whole `net_buf` fragment chain and is not bounded to 128, so an over-long line reaches the defect.

The data originates from the cellular modem over UART, driven by the network: operator-scan results, `+CGCONTRDP` IP/DNS info, socket indications, and `+KCELLMEAS` neighbour-cell reports. An attacker able to shape what the modem emits — a rogue base station, a compromised modem baseband, or a remote peer feeding oversized response framing — can drive a line past 128 bytes. The handlers run in the driver's RX thread in kernel context, so the corruption is kernel-side.

The `+KCELLMEAS` path is a full stack buffer overflow whose worst case is code execution in kernel context and whose floor is a reliable crash; the remaining sites are single-byte NUL out-of-bounds writes. Exploitation requires the modem to emit an over-long AT response line, giving high attack complexity over an adjacent (cellular radio) vector. The fix passes `sizeof(dst) - 1` (and correct explicit bounds for the IMSI and `+KCELLMEAS` sites) so the terminator always stays in bounds.

- Zephyr project bug tracker GHSA-9xc4-j5x8-v6jx

This has been fixed in main for v4.5.0

- PR 111243 fix for main
- PR 113040 fix for v4.4
- PR 113042 fix for v4.3
- PR 113041 fix for v3.7

CVE 2026-12521

Zephyr HTTP server kernel timeout-list corruption on spurious zsock_poll() return

The HTTP server core loop `http_server_run()` in `subsys/net/lib/http/http_server_core.c` polls the listening, stop, and client sockets with an infinite timeout and treats a `zsock_poll()` return of `0` as impossible, executing `break` and returning `0` without reaching its `closing:` cleanup. However, `zsock_poll()` (`zvfs_poll_internal()` in `lib/os/zvfs/zvfs_poll.c`) can legitimately return `0` even with an infinite timeout: after `k_poll()` wakes, the `ZFD_IOCTL_POLL_UPDATE` pass may find every fd reporting no `revents` (a spurious wakeup), yielding `ret == 0`.

On that path `close_all_sockets()` is skipped, so the per-client `inactivity_timer` (`struct k_work_delayable`) cancellations in `close_client_connection()` never run. Control returns to `http_server_thread()` with `server_running` still true, which re-enters `http_server_init()`. `http_server_init()` then `memset()`s the `ctx->clients` array — including the `k_work_delayable` timeout nodes — while one or more of those timers are still armed and linked in the kernel sys-timeout list.

When the kernel later services such a timeout it operates on a reinitialized object and follows the now-zeroed list links, corrupting the kernel timeout list and producing a delayed fault. The HTTP server loop is driven by unauthenticated remote network peers (`CONFIG_HTTP_SERVER`, all of HTTP/1/2/3), and connection churn raises the probability of the spurious-wakeup race, so a remote peer can influence the trigger. The realistic impact is a denial of service (kernel crash/hang) from the corrupted timeout list.

The fix replaces `break` with `continue`, re-polling on a spurious `0` return, which leaves the sockets and their armed timers intact and never re-initializes the context over live timers.

- Zephyr project bug tracker GHSA-g5v9-xmfp-7gxm

This has been fixed in main for v4.5.0

- PR 111239 fix for main
- PR 112942 fix for v4.4
- PR 112941 fix for v4.3
- PR 112940 fix for v3.7

CVE 2026-12522

Stack buffer overflow in Zephyr hl7800 modem driver parsing network-supplied +CGCONTRDP address fields

The HL7800 cellular modem driver's `+CGCONTRDP:` response handler `on_cmd_atcmdinfo_ipaddr()` in `drivers/modem/vendor_standalone/hl7800.c` parses the PDP-context dynamic parameters (local address, subnet mask, gateway, and DNS servers) that

the cellular network assigns to the device. The response is linearized into a 256-byte stack buffer, after which each address field length is computed from comma/. delimiter positions in the network-supplied data and used directly as the length argument to `strncpy()` into the fixed 64-byte stack buffer `temp_addr_str` (and the 16-byte `iface_ctx.dns_v4_string`).

Because the field length is derived from attacker-controlled delimiter positions and was not bounded against the destination buffer, a single field can be far larger than 64 bytes. A malicious or impersonated cellular network (for example a rogue base station) can return a crafted `+CGCONTRDP` response with an overlong address field, causing `strncpy()` to write past `temp_addr_str` on the modem worker thread's stack, plus an out-of-bounds NUL write at `temp_addr_str[addr_len]`.

No device-side privileges or user interaction are required: the device itself issues the `AT+CGCONTRDP=1` query during normal network attach and parses whatever the network returns. The overflow corrupts adjacent stack memory in supervisor context, yielding at minimum a remotely triggerable crash and potentially control-flow hijacking on targets without stack protection.

The fix bounds every field length against its destination buffer (`temp_addr_str` and `dns_v4_string`) before each copy, rejecting overlong fields.

- Zephyr project bug tracker GHSA-hchc-6489-w66v

This has been fixed in main for v4.5.0

- PR 111243 fix for main
- PR 113040 fix for v4.4
- PR 113042 fix for v4.3
- PR 113041 fix for v3.7

CVE 2026-7656

Broken IPv6 Neighbor Discovery input validation allows spoofed RA/NS/NA acceptance in Zephyr net stack

The IPv6 Neighbor Discovery handlers in `subsys/net/ip/ipv6_nbr.c` (`handle_ra_input`, `handle_ns_input`, `handle_na_input`) used an incorrect boolean expression that combined the RFC 4861 validity checks with the ICMPv6 code check using the wrong operator precedence: the form was `((length/hop/source/target checks) && (icmp_hdr->code != 0))`. Because every legitimate ND message carries ICMPv6 code 0, an attacker setting `code == 0` (the normal value) caused the entire predicate to evaluate false, so the packet was never dropped and all of the other checks were silently skipped. The bypassed checks include the mandatory Hop Limit == 255 verification (which proves an ND packet originated on-link and was not forwarded) and, for Router Advertisements, the requirement that the source be a link-local address, as well as multicast-target sanity checks. As a result, an adjacent on-link attacker — and, because the Hop-Limit-255 guard is bypassed, potentially a remote/off-link attacker whose packets would otherwise be rejected — can have forged Router Advertisement, Neighbor Solicitation, and Neighbor Advertisement messages accepted. A forged RA lets the attacker reconfigure the victim's default router, on-link prefixes (SLAAC), MTU, reachable/retransmit timers, and (with `CONFIG_NET_IPV6_RA_RDNSS`) DNS servers, while forged NS/NA enable neighbor-cache poisoning, enabling man-in-the-middle, traffic redirection, and denial of service. The flaw is an input-validation/authentication weakness rather than a memory-safety issue: the underlying packet-parsing primitives (`net_pkt_get_data`, `net_pkt_read`, `net_pkt_skip`) are independently bounds-safe and the validated `length` is the true buffer length, so skipping the length check causes no out-of-bounds access. The defect has existed since the logic was introduced in 2018 and shipped in all releases through v4.4.0; it is fixed by splitting the condition so any failing check drops the packet.

- Zephyr project bug tracker GHSA-cpjw-rvwx-ph9f

This has been fixed in main for v4.5.0

- PR 107902 fix for main
- PR 108131 fix for v4.4
- PR 108192 fix for v4.3
- PR 108195 fix for v3.7

CVE 2026-10666

Stack buffer overflow in `net_ipaddr_parse()` IPv4 address-with-port parsing in `subsys/net/ip/utils.c`

`parse_ipv4()` in `subsys/net/ip/utils.c` (reached via `net_ipaddr_parse()` for strings of the form "a.b.c.d:port") copies the port substring into a fixed 17-byte stack buffer (`char ipaddr[NET_IPV4_ADDR_LEN + 1]`) using a length of `str_len - end - 1`, where `str_len` is the full, unbounded input length and end is only the (<=15-byte) offset of the ':' delimiter. Because the destination size is never consulted, a crafted address string with a long suffix after the colon (e.g. "1.2.3.4:" followed by hundreds of bytes) causes an out-of-bounds stack write whose length and contents are fully attacker-controlled (`memcpy` of the suffix plus a trailing NUL), enabling memory corruption and at minimum a denial of service, and potentially control-flow hijack. The parser is reached from the standard socket API (`zsock_getaddrinfo` / literal-address resolution), DNS server-string configuration, and the eswifi Wi-Fi co-processor DNS-response path, so an application that resolves a network-influenced address string is exposed. The bug was introduced when the parser was added (Zephyr v1.9.0) and shipped in all releases through v4.4.0. The fix removes the unbounded copy and validates the port length before copying into a small dedicated buffer. Note: the equivalent IPv6 "[addr]:port" path in `parse_ipv6()` retains the same unbounded copy at this commit and remains a separate, still-reachable instance of the defect.

- Zephyr project bug tracker GHSA-532c-7g7f-jhmh

This has been fixed in main for v4.5.0

- PR 108529 fix for main
- PR 109058 fix for v4.4
- PR 109072 fix for v4.3
- PR 109065 fix for v3.7

CVE 2026-10673

Out-of-bounds write in ADIN2111/ADIN1110 OA SPI Ethernet RX frame reassembly

The Zephyr ADIN2111/ADIN1110 10BASE-T1S/T1L Ethernet driver (`drivers/ethernet/eth_adin2111.c`) reassembles received Ethernet frames in OPEN Alliance (OA) SPI mode by copying device-supplied 64-byte data chunks into a fixed static buffer `ctx->buf` of size `CONFIG_ETH_ADIN2111_BUFFER_SIZE` (default 1524 bytes). In `eth_adin2111_oa_data_read()`, each valid chunk was `memcpy`'d into `ctx->buf[ctx->scur]` and the write cursor `scur` advanced, with no check that `scur` + len stayed within the buffer. The number of chunks (up to 255, from the BUFSTS RCA field) and the per-chunk length are taken entirely from the frame data received off the wire; the cursor is only reset on a start-of-frame chunk. An attacker on the single-pair Ethernet segment can therefore send a frame whose reassembled size exceeds the configured buffer, causing the driver's RX offload thread to write attacker-controlled frame bytes past the end of the static buffer into adjacent driver/kernel memory (up to roughly 14.8 KB in the worst case). This is a remotely/adjacently reachable out-of-bounds write (CWE-787) that can corrupt memory and cause denial of service or potentially code execution. The defect was introduced when OA SPI support was added (commit 0ca8b0756b1) and shipped in releases v3.7.0 through v4.4.0. The fix adds a bounds check that drops the oversized frame and resets the cursor before the copy.

- Zephyr project bug tracker GHSA-hm6v-4jh4-3qc4

This has been fixed in main for v4.5.0

- PR 108200 fix for main
- PR 108900 fix for v4.4
- PR 108901 fix for v4.3
- PR 108899 fix for v3.7

CVE 2026-12629

PL011 UART error interrupts never cleared, enabling an external-peer interrupt-storm denial of service

The ARM PL011 UART driver in `drivers/serial/uart_pl011.c` fails to acknowledge receive error interrupts. On the PL011, the framing, parity, break, and overrun error interrupts (`PL011_IMSC_ERROR_MASK`) are cleared only by writing the interrupt-clear register `UARTICR`; reading the data register clears the RX interrupt and the per-byte RSR status but not the error interrupt status in `MIS`. The interrupt service routine `pl011_isr()` acknowledged only the CTS modem-status interrupt and never wrote `icr` for the error bits, so an asserted error interrupt remains pending after the ISR returns.

When an application enables error-interrupt reporting via the public `uart_irq_err_enable()` API, an attacker who controls the serial peer can deterministically assert these error bits by injecting line errors on the RX line — a baud/stop-bit mismatch or mid-character break (framing/break error), a flipped parity bit (parity error), or FIFO flooding (overrun error). Because the error interrupt is never cleared, the interrupt line stays asserted and the CPU re-enters `pl011_isr()` immediately and indefinitely, producing an interrupt-storm livelock from which the core makes no forward progress.

The impact is an availability-only denial of service (permanent hang), reachable from an external or removable UART peer. Exploitation is gated by configuration: the error interrupt is off by default and no in-tree subsystem enables it, so only applications that explicitly call `uart_irq_err_enable()` on a PL011-based, interrupt-driven port are affected. The fix makes `pl011_isr()` acknowledge the pending error bits via `uart->icr`, breaking the loop, and additionally clears the latched RSR status in `pl011_err_check()`.

- Zephyr project bug tracker GHSA-36rp-2hcp-f5hv

This has been fixed in main for v4.5.0

- PR 111222 fix for main
- PR 112933 fix for v4.4
- PR 112932 fix for v4.3

CVE 2026-12630

6LoWPAN IPHC uncompression out-of-bounds read on reserved destination addressing mode

Zephyr's 6LoWPAN IP Header Compression (IPHC) uncompression code contains an out-of-bounds read in `get_ihpc_inlined_size()` (`subsys/net/ip/6lo.c`). The destination inline size is looked up in `da_inline_size_table`, which has 13 entries, using an index built from the `M`, `DAC` and `DAM` bits of the received IPHC dispatch word (`iphc & NET_6LO_IPHC_DA_MASK`, a 4-bit value of 0-15). The reserved combinations 13, 14 and 15 are not bounds-checked and read past the end of the table.

The `iphc` word is taken directly from the received frame, and `get_ihpc_inlined_size()` is reached on every inbound 6LoWPAN frame via `net_6lo_uncompress()` from the 802.15.4 receive path (`subsys/net/l2/ieee802154/ieee802154_6lo.c` and `ieee802154_6lo_fragment.c`). An unauthenticated attacker on the radio/adjacent link can therefore craft a frame whose destination addressing-mode nibble selects an out-of-range index, with no privileges or user interaction.

The out-of-bounds value becomes the computed `inline_size`, which then drives header reconstruction before the buffer-length check: it is used to dereference `*(pkt->buffer->data + sizeof(iphc) + inline_size)` and to compute a `size_t diff` that can underflow, leading to a further out-of-bounds read of the packet buffer and malformed uncompression. The practical impact is a radio-triggerable out-of-bounds read / denial-of-service on the receiver; the leaked byte is not returned to the attacker. The fix rejects any destination index beyond the table, aborting processing of the malformed frame.

- Zephyr project bug tracker GHSA-45c8-pmgj-6jrc

This has been fixed in main for v4.5.0

- PR 111272 fix for main
- PR 113046 fix for v4.4
- PR 113044 fix for v3.7

CVE 2026-12631

Broken access-control denial in k_thread_join/k_thread_abort syscall validation in Zephyr kernel

The Zephyr kernel validates the `k_thread_join()` and `k_thread_abort()` system calls (declared `__syscall` in `include/zephyr/kernel.h`) through `thread_obj_validate()` in `kernel/thread.c`. Its `default` switch branch is the access-denied path, taken when `k_object_validate()` returns `-EPERM` (the calling user thread was never granted access to the target thread object) or `-EBADF` (the supplied pointer is not a registered kernel object of the right type). That branch invoked `K_OOPS(K_SYSCALL_VERIFY_MSG(ret, "access denied"))`, but `K_SYSCALL_VERIFY_MSG` treats a true expression as success; the non-zero error code `ret` therefore read as "verified OK", the kernel oops was never raised, and control fell through to `CODE_UNREACHABLE`.

Because `k_thread_join()` and `k_thread_abort()` are system calls, an unprivileged user-mode thread (under `CONFIG_USERSPACE`) can reach this denial path directly by calling either syscall on a thread object it does not own. Instead of the offending thread being cleanly terminated, execution reaches `__builtin_unreachable()` while running in supervisor mode inside the syscall handler.

On Clang builds `CODE_UNREACHABLE` emits an illegal-instruction trap, so a user thread can deterministically crash the kernel — a locally triggerable denial of service that escapes the userspace sandbox. On GCC builds the path is undefined behavior: the compiler may drop the return-value handling for `thread_obj_validate()`, so it can return an undefined `bool`; if that is `false`, the caller proceeds into the real `k_thread_join()`/`k_thread_abort()` implementation for a thread the user was never authorized to access, an access-control bypass.

The fix changes the verification expression to `ret == 0`, so a denied (non-zero) result now correctly raises `K_OOPS` and terminates the offending caller.

- Zephyr project bug tracker GHSA-crfw-75jw-hjm3

This has been fixed in main for v4.5.0

- PR 111301 fix for main
- PR 113287 fix for v4.4
- PR 113286 fix for v4.3
- PR 113285 fix for v3.7

CVE 2026-12632

Out-of-bounds read in Zephyr PTP message parsing from unvalidated message type

Zephyr's Precision Time Protocol receive handler `ptp_msg_post_recv()` in `subsys/net/lib/ptp/msg.c` takes the 4-bit message type straight off the wire via `ptp_msg_type()` (`msg->header.type_major_sdo_id & 0xF`, range 0-15) and uses it to index the `msg_size[]` table. That table only

defines entries up to `PTP_MSG_MANAGEMENT` (0xD), giving it `ARRAY_SIZE == 14`. Before the fix there was no upper-bound check, so the undefined types `0xE` and `0xF` indexed one or two `int` slots past the end of the array — an out-of-bounds read of adjacent read-only data.

The out-of-bounds value is then reused as a length: it gates `msg_size[type] > cnt`, and when it is small or negative it makes `cnt - msg_size[type]` a large positive budget passed to `msg_tlv_post_recv()`, whose TLV loop then walks the message suffix past the received bytes, performing further out-of-bounds reads and in-place byte-swap writes on memory beyond the message slab.

The defect is reached directly from the network: `ptp_port_event_gen()` in `subsys/net/lib/ptp/port.c` reads a PTP frame with `ptp_transport_recv()` and calls `ptp_msg_post_recv()` with the attacker-chosen type. PTP uses UDP multicast or raw Ethernet (`0x88F7`) and is unauthenticated, so any host on the same link can trigger the indexing on a `CONFIG_PTP`-enabled node with no preconditions.

The reliably reproducible impact is a denial of service (fault/crash); a limited memory-corruption path exists but depends on the build-specific value adjacent to `msg_size[]`, which the attacker cannot tune. The fix rejects `type >= ARRAY_SIZE(msg_size)` with `-EBADMSG` before any indexing.

- Zephyr project bug tracker GHSA-frjr-h396-7wh4

This has been fixed in main for v4.5.0

- PR 111271 fix for main
- PR 111665 fix for v4.4
- PR 111667 fix for v4.3
- PR 111666 fix for v3.7

CVE 2026-12633

Out-of-bounds write in IPv6 6LoWPAN Context Option handling via unauthenticated Router Advertisement

The IPv6 neighbor-discovery code in `subsys/net/ip/ipv6_nbr.c` processes the 6LoWPAN Context Option (6CO, RFC 6775) carried inside ICMPv6 Router Advertisements. In `handle_ra_6co()` the 8-bit `context_len` field is taken directly from the packet and was never bounded to the RFC maximum of 128. The function computes `context->context_len / 8` and then performs `memset(context->prefix + context_len, 0, sizeof(context->prefix) - context_len)`, where `context->prefix` is a fixed 16-byte array.

With `context_len` between 136 and 255 (and the option length field set to 3, which the pre-fix validation accepts), `context_len / 8` evaluates to 17..31, so the `memset` length `16 - context_len/8` underflows the unsigned `size_t` argument to roughly `SIZE_MAX`. This produces an unbounded out-of-bounds `memset` that zeroes kernel memory well past the 6lo context structure.

The defect is reachable from unauthenticated, link-local input: any host on the same link can send a crafted Router Advertisement with a 6CO option. The RA handler validates only the option length field before calling `handle_ra_6co()`, so a single packet triggers the wild write. The code is compiled when `CONFIG_NET_6LO_CONTEXT` is enabled.

The impact is a reliable remote (adjacent) denial of service via memory corruption, with collateral integrity loss as the `memset` zeroes contiguous memory before the system faults. Router Advertisements are link-scoped and not forwarded, so the attacker must be on the same link (`AV:A`). The fix rejects any `context_len` greater than 128 before the length computation.

- Zephyr project bug tracker GHSA-h5m5-hm6j-cgpf

This has been fixed in main for v4.5.0

- PR 111275 fix for main
- PR 113049 fix for v4.4

- PR 113051 fix for v4.3
- PR 113050 fix for v3.7

CVE 2026-12634

Out-of-bounds stack write in the settings NVS backend from over-reported nvs_read length

The NVS backend of the Zephyr settings subsystem (`subsys/settings/src/settings_nvs.c`) reads stored setting-name entries into fixed 74-byte stack buffers and NUL-terminates them with `buf[rc] = '\0'`, where `rc` is the return value of `nvs_read()`. Per its contract, `nvs_read()` returns the *full stored entry length* (`wlk_ate.len`), which can exceed the supplied buffer length — only `MIN(len, stored_len)` bytes are actually copied, but the return value may be much larger, bounded only by the NVS sector size. Three sites (`settings_nvs_cache_match()`, `settings_nvs_load()`, and `settings_nvs_save()`) used this value directly as the NUL index without clamping, so an oversized stored name entry causes a single `\0` byte to be written past the end of the stack buffer at an attacker-influenced offset (CWE-787).

The oversized entry cannot arise through the normal settings API, where names are bounded by `SETTINGS_MAX_NAME_LEN`. It requires an actor able to write the flash that backs the settings partition — a co-resident or untrusted component sharing the flash device, a malicious settings image/restore, or offline/physical flash access (a shared-flash threat model). The malformed entry is parsed when `settings_load()` runs at boot or subsystem init, or during `settings_save()`.

The out-of-bounds write is a single NUL byte at an offset equal to the crafted entry length (up to the NVS sector size), so the practical impact is a crash or denial of service and limited stack corruption rather than reliable code execution. There is no confidentiality impact, and the path is not reachable from the network through the ordinary settings interface. The fix skips any entry whose `nvs_read()` length is greater than or equal to the buffer size before performing the NUL store.

- Zephyr project bug tracker GHSA-q7c8-m2qg-385c

This has been fixed in main for v4.5.0

- PR 111314 fix for main
- PR 113298 fix for v4.4
- PR 113296 fix for v4.3
- PR 113297 fix for v3.7

CVE 2026-12999

Infineon Airoc Wi-Fi driver leaks TX buffers on send failure, leading to permanent pool exhaustion

The Infineon Airoc Wi-Fi driver's transmit callback `airoc_mgmt_send()` in `drivers/wifi/infineon/airoc_wifi.c` allocates a `net_buf` from the fixed `airoc_pool` for every outbound packet. When `whd_network_send_ethernet_data()` returns a synchronous failure, the underlying WHD library does not take ownership of the buffer, but the pre-fix driver returned `-EIO` without releasing it. Each failed transmit therefore permanently leaks one buffer from the pool.

`airoc_pool` is small and fixed (`AIROC_WIFI_TX_PACKET_POOL_COUNT + AIROC_WIFI_RX_PACKET_POOL_COUNT`, default 20 buffers) and is shared by WHD's `whd_host_buffer_get` callback for both transmit and receive. Once enough send failures have leaked the pool dry, `airoc_wifi_host_buffer_get()` returns `WHD_BUFFER_ALLOC_FAIL` for all subsequent allocations, so both transmit and the WHD-driven receive path fail and Wi-Fi connectivity is lost until the device is rebooted.

The leak occurs only on the transmit error path. A Wi-Fi-adjacent attacker can influence the conditions that cause synchronous send failures (for example by deauthenticating/disassociating the station while the local stack continues to attempt transmits), and ordinary transient failures over

the device's lifetime accumulate toward the same state. Reliable on-demand triggering is of high complexity and the impact is availability-only, but the resulting denial of service is permanent and non-recoverable without a reboot.

The fix releases the buffer with `airoc_wifi_buffer_release()` on the failure branch, returning it to the pool. The commit also removes a redundant `k_sem_give()` in `airoc_mgmt_disconnect()`; because `data->sema_common` is a binary semaphore (limit 1) the duplicate give merely saturated at 1 and had no security impact.

- Zephyr project bug tracker GHSA-8w97-ghfm-5wjp

This has been fixed in main for v4.5.0

- PR 111163 fix for main
- PR 113304 fix for v4.4
- PR 113306 fix for v4.3
- PR 113305 fix for v3.7

CVE 2026-13212

Zephyr virtio driver calls an arbitrary function pointer from an out-of-range used-ring descriptor id

The Zephyr virtio driver does not validate the descriptor-chain head id that the virtio device writes into the used ring. In `virtio_isr()` (`drivers/virtio/virtio_common.c`), the device-written `vq->used->ring[idx].id` is used directly as an index into `vq->recv_cbs[]` and `vq->desc[]`, which are both allocated with exactly `vq->num` entries. `recv_cbs[]` holds `{cb, opaque}` callback entries, and the indexed callback pointer is then invoked as `cbe.cb(cbe.opaque, used_len)`.

Because the id is consumed as a 16-bit value with no bound check, a malicious or compromised virtio backend (an untrusted hypervisor, or an untrusted hardware/peer-processor virtio device on a PCI or MMIO transport) can supply an id far beyond `vq->num`. This causes an out-of-bounds read of a `{function pointer, argument}` pair from heap memory beyond `recv_cbs[]`, after which the driver calls that attacker-shaped pointer in the guest's interrupt context. No guest privileges or user interaction are required; the backend triggers it by writing the shared used ring and raising the queue interrupt.

The result is an arbitrary / attacker-influenced function-pointer call in the Zephyr guest, i.e. a control-flow-hijack primitive that can lead to code execution or, at minimum, a reliable crash. The fix rejects any used-ring id `>= vq->num` before indexing `recv_cbs[]`/`desc[]` or invoking the callback. This affects builds using `CONFIG_VIRTIO` with the PCI or MMIO transport.

- Zephyr project bug tracker GHSA-7884-373w-qqhx

This has been fixed in main for v4.5.0

- PR 111289 fix for main
- PR 113344 fix for v4.4
- PR 113345 fix for v4.3

CVE 2026-13213

Bluetooth HAS: NULL-pointer dereference DoS when a bonded peer reconnects before bt_has_register

The Hearing Access Service (HAS) GATT server in `subsys/bluetooth/audio/has.c` installs a connection-callback set unconditionally via `BT_CONN_CB_DEFINE`, so `security_changed()` runs for every connection that establishes security even before

the application has called bt_has_register(). The service attribute pointers hearing_aid_features_attr, preset_control_point_attr, and active_preset_index_attr remain NULL until bt_has_register() resolves them and sets has.registered.

With CONFIG_BT_SETTINGS, settings_set_cb() restores each bonded client's persisted context at boot and unconditionally sets context->flags to BONDED_CLIENT_INIT_FLAGS (non-zero). When a previously bonded peer reconnects and re-establishes security during the startup window before bt_has_register() has been called, security_changed() sees the non-zero flags and schedules notify_work_handler, which calls bt_gatt_is_subscribed() with a still-NULL attribute pointer. That triggers an assertion (__ASSERT(attr, ...) in bt_gatt_is_subscribed()), or a NULL dereference of attr->uuid when assertions are compiled out.

The result is a remotely triggerable (Bluetooth, adjacent) crash of the HAS peripheral. Exploitation requires the peer to have previously bonded with the device and to reconnect within the boot-time race window before the application registers the service; a peer that reconnects persistently can prolong the outage. Impact is denial of service only, with no memory corruption or information disclosure.

The fix adds an early if (!has.registered) { return; } guard in security_changed(), so no notification work is scheduled until the GATT service is registered and its attribute pointers are valid.

- Zephyr project bug tracker GHSA-9rj8-3fvm-cc9f

This has been fixed in main for v4.5.0

- PR 111767 fix for main
- PR 113349 fix for v4.4
- PR 113347 fix for v4.3
- PR 113348 fix for v3.7

CVE 2026-13214

Stack buffer overflow in OCPP GetConfiguration key parsing

The OCPP 1.6 client in subsys/net/lib/ocpp/ocpp_j.c contains a stack buffer overflow in parse_getconfig_msg(). When handling a GetConfiguration request from the central system, the handler copied the attacker-controlled JSON "key" string into the caller's fixed 50-byte stack buffer (skey[CISTR50], declared in subsys/net/lib/ocpp/ocpp.c) using an unbounded strcpy(). The parsed key value points directly into the receive buffer, so its length is bounded only by the message size (CONFIG_OCPP_RECV_BUFFER_SIZE, default 2048).

The GetConfiguration message is delivered over the WebSocket connection that the charge point opens to its configured central system. The reader thread ocpp_wsreader() reads the message into ui->recv_buf and dispatches it to parse_getconfig_msg() via the PDU function table. An attacker who controls the central system endpoint, or a man-in-the-middle on an unencrypted connection, can send a GetConfiguration request whose "key" field exceeds 50 bytes and overflow the reader thread's stack with attacker-chosen bytes.

The consequence is a remotely triggerable stack smash on the OCPP reader thread: at minimum a denial of service, and plausibly remote code execution depending on build-time hardening such as stack canaries and MPU configuration. The fix replaces the strcpy() with a bounded strncpy(key, payload.key[0], CISTR50 - 1) followed by explicit NUL termination, matching the bounded copies already used by the sibling handlers.

- Zephyr project bug tracker GHSA-fqhf-6v24-4px2

This has been fixed in main for v4.5.0

- PR 111242 fix for main
- PR 113330 fix for v4.4
- PR 113329 fix for v4.3

CVE 2026-13215

Zephyr ext2 mount: unvalidated superblock block size causes out-of-bounds write from a crafted filesystem image

The Zephyr ext2 filesystem driver fails to validate the `s_log_block_size` field of the on-disk superblock when mounting a filesystem. `ext2_verify_disk_superblock()` in `subsys/fs/ext2/ext2_impl.c` checks the magic number, revision, inode size and group counts, but never bounds `s_log_block_size`. On a successful verify, `subsys/fs/ext2/ext2_ops.c` computes `fs->block_size = 1024 << superblock.s_log_block_size` from this attacker-controlled `uint32_t`, so a crafted value either overflows the shift (undefined behaviour) or yields a block size far larger than `CONFIG_EXT2_MAX_BLOCK_SIZE`.

That block size is then passed to `k_mem_slab_init()` by `ext2_init_blocks_slab()` to carve `CONFIG_EXT2_MAX_BLOCK_COUNT` blocks out of the fixed static buffer `__ext2_block_memory_buffer`, whose size is `CONFIG_EXT2_MAX_BLOCK_COUNT * CONFIG_EXT2_MAX_BLOCK_SIZE`. `k_mem_slab_init()` does not verify that the requested blocks fit the buffer, and the ext2 wrapper discards its return value, so the slab is laid out past the end of the static buffer. The mount immediately reads block-group, bitmap and inode blocks of `fs->block_size` bytes each into these slab blocks, producing an out-of-bounds write into adjacent static memory on the first block read.

The entire path is gated only by data read from the mounted image, making this reachable by any attacker who can present a crafted ext2 image to a device that mounts it (for example a removable SD card or storage medium). Because the ext2 driver runs in kernel mode, supplying image bytes yields a supervisor-mode memory-corruption primitive, with impact ranging from denial of service to potential code execution.

The fix rejects `s_log_block_size` values that overflow the shift (greater than 11) or that produce a block size exceeding `CONFIG_EXT2_MAX_BLOCK_SIZE`, so the block slab can no longer be initialized larger than its backing buffer.

- Zephyr project bug tracker GHSA-j52j-gfj9-rwjm

This has been fixed in main for v4.5.0

- PR 111241 fix for main
- PR 113754 fix for v4.4
- PR 113327 fix for v4.3
- PR 113326 fix for v3.7

CVE 2026-13216

Out-of-bounds stack write in Zephyr virtio PCI driver from unvalidated device-supplied capability length

The virtio PCI driver (`drivers/virtio/virtio_pci.c`) parses a device's PCI capability list during driver initialization. In `virtio_pci_read_cap()` the device-supplied capability length byte `cap_len` (read from PCI config space via `pcie_conf_read()`) was only checked with `assert(tmp.cap_len == cap_struct_size)`. That `assert` resolves to `__ASSERT_NO_MSG()`, gated by `CONFIG_ASSERT`, which defaults off in production builds, so the value reached the copy logic completely unvalidated.

The length then drives a loop that copies extra capability dwords into a fixed-size stack buffer supplied by the caller. A `cap_len` below the 24-byte base `struct virtio_pci_cap` underflows the unsigned `extra_data_words` count to a near-`SIZE_MAX` value, producing an effectively unbounded stack write; a `cap_len` above the caller's buffer (up to 255) writes up to roughly 228 bytes of device-controlled data past the buffer. Both are out-of-bounds writes of attacker-controlled content executed in kernel mode during boot-time device probe.

The input originates from the virtio device. In the common deployment where Zephyr runs as a guest under a hypervisor, the device backend is the host, which already fully outranks the guest,

so the bug yields no privilege escalation. The exploitable case is a virtio device that is untrusted relative to the Zephyr kernel — an untrusted or physical/passthrough virtio PCIe device on a bare-metal system, or a confidential-computing posture where the guest must defend against the host — where a malicious device can corrupt the kernel stack and potentially achieve code execution or a crash.

The fix replaces the compiled-out assert with a runtime range check rejecting `cap_len` outside `[sizeof(struct virtio_pci_cap), cap_struct_size]` before any arithmetic or copy.

- Zephyr project bug tracker GHSA-qrh3-4mvv-w667

This has been fixed in main for v4.5.0

- PR 111289 fix for main
- PR 113344 fix for v4.4
- PR 113345 fix for v4.3

CVE 2026-13217

NULL-pointer dereference in Zephyr OCPP CALLRESULT parsing via unchecked strtok_r/atoi

The OCPP 1.6 client in `subsys/net/lib/ocpp/ocpp.c` reconstructs a session handle and PDU id from the `uid` field of a CALLRESULT message. In `ocpp_process_server_msg()` the code calls `atoi(strtok_r(uid, "-", &tmp))` without checking the `strtok_r` return value. When the server-supplied `uid` is empty or contains no `-` delimiter, `strtok_r()` returns `NULL` and `atoi(NULL)` dereferences a NULL pointer, which is undefined behaviour.

The `uid` originates from network data: `parse_rpc_msg()` in `subsys/net/lib/ocpp/ocpp_j.c` JSON-parses a frame received from the OCPP central system over TCP/WebSocket and copies the server-controlled string into the local buffer. A malicious or compromised central system, or a man-in-the-middle on a non-TLS `ws://` connection, can return a malformed `uid` to reach the defect. No authentication beyond the existing server connection (or MITM position) is required, and the reconstructed pointer is membership-validated by `ocpp_session_is_valid()`, so the impact is limited to the NULL dereference rather than arbitrary pointer use.

On Zephyr targets that trap access to address 0 (MMU/MPU platforms or `CONFIG_NULL_POINTER_EXCEPTION_DETECTION`), the dereference faults inside the OCPP reader thread and invokes the fatal handler, producing a remote denial of service of the charge point; on bare targets where address 0 is readable the call returns 0 and is benign, so the impact is availability-only and platform-conditional.

The applied fix guards only the first `atoi()`; the second `strtok_r(NULL, "-", &tmp)` followed by `pdu = atoi(buf)` in the same function remains unguarded and the identical NULL dereference is still reachable from the same network input when the `uid` has a first token but no second `-`-delimited token. A complete fix should validate the second token as well.

- Zephyr project bug tracker GHSA-w234-pcxp-4q8r

This has been fixed in main for v4.5.0

- PR 111242 fix for main
- PR 113330 fix for v4.4
- PR 113329 fix for v4.3

CVE 2026-13343

Uninitialised stack memory disclosure in the MIDI 2.0 UMP Stream responder

The UMP Stream responder library in `lib/midi2/ump_stream_responder.c` builds reply packets in a 16-byte `struct midi_ump` (`uint32_t data[4]`). The builders `make_endpoint_info()` and `make_function_block_info()` populate only the first two words (`res.data[0]` and `res.data[1]`)

and, before this fix, declared their result as an uninitialised local (`struct midi_ump res;`). The remaining two words (`res.data[2]`, `res.data[3]`) retain stale stack contents.

Endpoint Info and Function Block Info notifications are UMP Stream messages (`UMP_MT_UMP_STREAM`), which are 4 words long, so the full 16-byte packet — including the two uninitialised words — is transmitted verbatim by `cfg->send()`. The responder is driven by attacker-supplied UMP Stream Endpoint-Discovery / Function-Block-Discovery requests via `ump_stream_respond()`. In the in-tree Network MIDI 2.0 server (`subsys/net/lib/midi2/netmidi2.c`) these requests arrive as UDP datagrams and, with the default no-authentication endpoint, a remote peer can establish a session and trigger the responses; the same library also serves USB MIDI 2.0 hosts.

Each discovery request causes the device to disclose 8 bytes of its own uninitialised stack memory to the peer, and the request is freely repeatable. This is a confidentiality-only information leak (root cause is use of an uninitialised variable, CWE-457/CWE-908); the leaked words could include residual data or pointer values. There is no memory-corruption, integrity, or availability impact.

The fix zero-initialises both result structs (`struct midi_ump res = {0};`), so the trailing words are cleared before transmission. These are the only two responder builders that left trailing words unset (`send_string()` already zeroes its buffer), so the leak is fully closed.

- Zephyr project bug tracker GHSA-4w5x-w7j4-6xxc

This has been fixed in main for v4.5.0

- PR 111286 fix for main
- PR 113341 fix for v4.4

CVE 2026-13351

net: Maliciously fragmented IPv6 packets can prevent receiving/processing future incoming packets

The Zephyr network stack can be prevented from receiving or processing future incoming packets by sending a few maliciously fragmented IPv6 packets. When `net_ipv6_handle_fragment_hdr()` triggers an ICMPv6 error response for a malformed fragment, it returns `NET_OK` without unreferencing the packet, leaking the RX network packet buffer. Each call to `k_mem_slab_alloc()` lacks a counterpart `k_mem_slab_free()`, so replaying such a packet a few times exhausts the RX buffer slab, after which the driver repeatedly fails to obtain RX buffers, resulting in a denial of service.

- Zephyr project bug tracker GHSA-cv4q-2j56-4wqf

This has been fixed in main for v4.4.0

- PR 104044 fix for main
- PR 104205 fix for v4.3
- PR 104206 fix for v4.2
- PR 104209 fix for v3.7

CVE 2026-13478

Out-of-bounds read in Zephyr ext2 block-bitmap validation from a crafted s_blocks_count

The Zephyr ext2 filesystem driver validates the on-disk block bitmap in `ext2_init_fs()` (`subsys/fs/ext2/ext2_impl.c`) by passing `fs_blocks = s_blocks_count - s_first_data_block` to `ext2_bitmap_count_set()`. That helper (`subsys/fs/ext2/ext2_bitmap.c`) treats its argument as a number of bits and reads one bitmap byte per eight bits, but the bitmap

buffer (BGROUP_BLOCK_BITMAP) is a single fetched block of only fs->block_size bytes (capacity fs->block_size * 8 bits). s_blocks_count and s_first_data_block are taken verbatim from the superblock and were never bounded against this single-group capacity; ext2_verify_disk_superblock() checks the magic, revision, and block-size shift but not the block count.

A crafted ext2 image with an oversized s_blocks_count (up to ~4 billion, against a maximum 4096-byte block / 32768-bit bitmap) makes ext2_bitmap_count_set() scan roughly 512 MB of memory past the bitmap block — a large out-of-bounds read of the static block slab and adjacent memory.

The defect is reached during mount: ext2_init_fs() is invoked from ext2_mount() (subsys/fs/ext2/ext2_ops.c), the registered .mount operation. Any path that mounts an attacker-supplied ext2 image (removable media, a disk/flash partition, or a downloaded image) triggers it. The kernel-privileged parser operates on attacker-controlled data, so the bug is exploitable wherever untrusted ext2 media can be mounted.

Impact is an out-of-bounds read only: the resulting bit count is compared internally and the mount is rejected, so no attacker-controlled bytes are returned (not a useful information leak). The ~512 MB over-read will almost certainly cross an unmapped or MPU-protected boundary and fault, crashing the system — a denial of service triggered by mounting a single malformed image. The fix rejects any image whose fs_blocks exceeds fs->block_size * 8 before the scan.

- Zephyr project bug tracker GHSA-gj29-7f7m-4c29

This has been fixed in main for v4.5.0

- PR 111970 fix for main
- PR 112132 fix for v4.4
- PR 112131 fix for v4.3
- PR 113351 fix for v3.7

CVE 2026-13479

Out-of-bounds read in LoRaWAN clock-sync AppTimeAns downlink handler

The LoRaWAN application-layer clock-synchronization service parses downlinks in clock_sync_package_callback() (subsys/lorawan/services/clock_sync.c). Its command loop only guarantees that the one-byte command id is in bounds; for the CLOCK_SYNC_CMD_APP_TIME (AppTimeAns) command the handler then reads a 4-byte time correction via sys_get_le32() plus a 1-byte token without checking that 5 bytes remain in the receive buffer (len - rx_pos). A short or crafted AppTimeAns therefore reads up to 5 bytes past the end of the decrypted payload.

The payload (rx_buf/len) is the decrypted application frame delivered to the registered downlink callback (mcps_indication->Buffer/BufferSize). Reaching the handler requires a frame on the clock-sync port that passes LoRaWAN's MAC integrity check and FRMPayload decryption, so the practical attacker is a malicious or compromised network/application server (the designated sender of AppTimeAns) or a party holding the session keys, rather than an arbitrary radio listener.

The over-read is bounded: the backing store is a fixed 255-byte static buffer, so the few stray bytes do not fault, and the read values (time_correction, token) are used only internally and never transmitted, so there is no disclosure to the attacker and no crash. The sole effect is that a stale token matching ctx.req_token can apply a garbage time_correction to the device's own clock offset (ctx.time_offset), a minor integrity impact confined to the victim's time estimate. The fix adds an explicit length check that drops a too-short AppTimeAns. Note the sibling one-byte reads in the periodicity and force-resync handlers remain unguarded with the same negligible impact.

- Zephyr project bug tracker GHSA-2m6g-p3vx-p2fh

This has been fixed in main for v4.5.0

- PR 111983 fix for main
- PR 113433 fix for v4.4
- PR 113432 fix for v4.3
- PR 113431 fix for v3.7

CVE 2026-13480

Out-of-bounds read in LoRaWAN fragmented data block transport (FUOTA) downlink handler

The LoRaWAN TS004 Fragmented Data Block Transport handler `frag_transport_package_callback()` in `subsys/lorawan/services/frag_transport.c` parses downlink command bytes without validating that enough payload bytes remain before each access. The loop's only bound is `rx_pos < len`; after consuming the one-byte command id the handler cast `rx_buf + rx_pos` to a 10-byte `struct frag_transport_setup_req`, and for a `DATA_FRAGMENT` command passed `&rx_buf[rx_pos]` to the fragment decoder, which reads exactly `ctx.frag_size` bytes — with no remaining-length check in either case.

The fragment size is attacker-chosen in a preceding `FRAG_SESSION_SETUP` command (`ctx.frag_size = req->frag_size`, capped at `CONFIG_LORAWAN_FRAG_TRANSPORT_MAX_FRAG_SIZE`, default 232). `rx_buf` aliases the 255-byte static `MacCtx.RxPayload` buffer in the loramac-node MAC layer, while `len` is the actual decrypted payload length. By padding a downlink with mismatched-index `DATA_FRAGMENT` filler commands (each advancing `rx_pos` by three bytes without producing an answer) and appending one matching-index fragment near the end of the payload, an attacker can make the decoder read up to roughly `frag_size` bytes past the end of `RxPayload`, copying adjacent static memory into the decoder buffers and the FUOTA flash image.

The handler runs only on downlinks that have already passed the LoRaWAN frame MIC and FRMPayload decryption, so the defect is reachable only by a party holding the device's session keys (the FUOTA server or an attacker who has compromised those keys). The out-of-bounds bytes are never returned to the sender — the only uplink emitted is a status answer carrying fragment counts — so there is no direct disclosure channel, and on typical flat-memory LoRaWAN MCUs the over-read stays within mapped memory, making a crash unlikely. The impact is therefore a bounded out-of-bounds read with limited confidentiality consequence and no write or control-flow primitive. The fix adds remaining-length guards before each access.

- Zephyr project bug tracker GHSA-845m-2m84-g5h2

This has been fixed in main for v4.5.0

- PR 111983 fix for main
- PR 113433 fix for v4.4
- PR 113432 fix for v4.3
- PR 113431 fix for v3.7

CVE 2026-13481

Out-of-bounds read in PTP management TLV TIME parsing in Zephyr net PTP

The IEEE 1588 PTP management-message parser in `subsys/net/lib/ptp/tlv.c` mishandles the `PTP_MGMT_TIME` management id. In `tlv_mgmt_post_recv()`, the `PTP_MGMT_TIME` case casts `mgmt_tlv->data` to a 10-byte `struct ptp_timestamp` and reads it (then byte-swaps and writes it back) without first checking that the TLV data field is at least `sizeof(struct ptp_timestamp)`. Every sibling management id in the same switch validates its length first; `PTP_MGMT_TIME` was the only case lacking that check.

The length passed in is the management data size (`tlv->length - 2`), and the upstream guard in `ptp_tlv_post_recv()` only requires `tlv->length > 2`, while `msg_tlv_post_recv()` validates only that the TLV fits within the received byte count, not a per-id minimum. A peer on the local PTP segment can therefore send a `PTP_MSG_MANAGEMENT` message carrying a short `PTP_MGMT_TIME`

TLV (data as small as 2 bytes), causing the parser to read and write 8 bytes beyond the validated data. The message type and TLV contents are taken straight off the wire, so the path is reachable by any adjacent attacker when `CONFIG_PTP` is enabled.

The over-read and write-back stay within the `struct ptp_msg` allocation (`mgmt_tlv->data` lives in the leading `mtu[NET_ETH_MTU]` union member, so `data + 10` lands at most a few bytes past `mtu[]`, inside the same object), so this is an out-of-bounds read of adjacent in-object memory plus a bounded in-place corruption of the message's parsed timestamp, not past-allocation memory corruption. Impact is limited to minor information exposure of adjacent bytes and corruption of the device's parsed management TIME value; there is no crash on the access and no reachable reference-count corruption.

The fix adds `if (length < sizeof(struct ptp_timestamp)) { return -EBADMSG; }` before the cast, matching the other management-id cases and fully closing the receive-path defect.

- Zephyr project bug tracker GHSA-mh5r-jxh8-hxwx

This has been fixed in main for v4.5.0

- PR 111969 fix for main
- PR 113353 fix for v4.4
- PR 117480 fix for v4.3
- PR 117479 fix for v3.7

CVE 2026-13734

Zephyr WireGuard mutates peer state before anti-replay check, enabling capture-replay endpoint hijack

Zephyr's WireGuard VPN data-plane receive handler `wg_process_data_message()` in `subsys/net/lib/wireguard/wg_crypto.c` validated the anti-replay counter too late. After AEAD decryption of a `MESSAGE_TRANSPORT_DATA` packet succeeded, the code committed several peer-state changes — `update_peer_addr()` (endpoint roaming update), the `keypair->last_rx`/`peer->last_rx` liveness timers, and `keypair_update()` (promote `next`→`current` and destroy the previous keypair) — and only afterward called `wg_check_replay()`. On a replayed packet the replay check returned `-EINVAL`, but none of the preceding mutations were rolled back.

The AEAD tag authenticates content but not freshness, so a replayed-but-authentic transport packet decrypts correctly. An attacker who captures one valid ciphertext off the wire (an on-path or shared-medium observer) can re-inject it from an arbitrary spoofed source address. Reaching the handler requires no credentials: it is driven directly from inbound UDP datagrams via the dispatch in `subsys/net/lib/wireguard/wg.c`.

Because the state mutations committed before the replay check, the replay repoints the peer endpoint to the attacker-chosen source address (roaming hijack), redirecting the victim's subsequent outbound tunnel traffic until the legitimate peer's next packet re-corrects it; it also prematurely destroys the previous keypair and refreshes the RX liveness timer. The tunnel payload stays encrypted under the session keypair, so this is an integrity/availability impact (traffic redirection and session disruption), not payload disclosure. The fix moves `wg_check_replay()` to immediately after a successful decrypt, before any peer-state mutation, matching the WireGuard specification and the Linux reference implementation.

- Zephyr project bug tracker GHSA-x7q7-fjx9-4vj2

This has been fixed in main for v4.5.0

- PR 111043 fix for main
- PR 112250 fix for v4.4

CVE 2026-13735

WireGuard keepalive transport-data messages accepted without Poly1305 authentication

Zephyr's WireGuard implementation in `subsys/net/lib/wireguard/wg_crypto.c` mishandled keepalive packets. In `wg_process_data_message()`, any type-4 transport-data message whose payload was exactly 16 bytes (an empty plaintext plus a bare Poly1305 tag, i.e. a keepalive) was accepted and returned immediately, before `wg_decrypt_packet()` was ever called. The Poly1305 authentication tag was therefore never verified; the only preceding gates were a cleartext receiver-index lookup (`get_peer_keypair_for_index()` on the attacker-supplied `data_hdr->receiver`) and a non-cryptographic keypair validity/expiry check.

The path is reachable entirely from the network: inbound UDP on the WireGuard port is dispatched by `wg_input()` to `handle_transport_data()` and then `wg_process_data_message()`. The 32-bit receiver index is transmitted in cleartext in WireGuard handshake and data messages, so an on-path observer learns it directly and an off-path attacker can brute-force it against the UDP port. Given an active receiving-valid session for that index, an attacker could send a 16-byte garbage payload and have it accepted without possessing the session key.

On acceptance the unauthenticated message caused the management layer to observe a spoofed `NET_EVENT_VPN_CONNECTED` signal (setting `peer->first_valid` and notifying any `net_mgmt` listener) and incremented the keepalive-RX statistic. The impact is limited to integrity of this status signal: no plaintext is decrypted or injected, no key is disclosed, and the early-return path did not update the peer endpoint or liveness timers, so there is no traffic-injection, session-takeover, or availability consequence.

The fix removes the pre-decrypt early return so a 16-byte payload flows through `wg_decrypt_packet()`, which verifies the Poly1305 tag over the empty plaintext, followed by the existing anti-replay check; only an authenticated, non-replayed message is then recognised as a keepalive. Forged keepalives now fail the tag check and are counted as decrypt failures.

- Zephyr project bug tracker GHSA-xxrw-r78f-f6mx

This has been fixed in main for v4.5.0

- PR 111043 fix for main
- PR 112250 fix for v4.4

CVE 2026-14366

SiWx91x WiFi driver double-unref / use-after-free of caller-owned TX net_pkt

The Silicon Labs SiWx917 WiFi driver's transmit callback `siwx91x_send()` in `drivers/wifi/siwx91x/siwx91x_wifi.c` frees a network packet it does not own. In the Zephyr TX path the `net_pkt` is owned by the L2/networking stack; the driver only borrows it to copy the frame bytes into a local `net_buf`. Before the fix, after transmitting, `siwx91x_send()` additionally called `net_pkt_unref(pkt)` on the caller-owned packet, dropping its last reference and returning it to the shared packet pool prematurely. This code path is compiled in by default (`CONFIG_WIFI_SILABS_SIWX91X_NET_STACK_NATIVE`).

The caller, `ethernet_send()` in `subsys/net/l2/ethernet/ethernet.c`, keeps using the packet after the driver returns: it reads `net_pkt_get_len(pkt)`, updates TX statistics, and then performs its own `net_pkt_unref(pkt)`. Because the driver already released the packet, these are use-after-free reads followed by a second unref (a double free). When concurrent network activity recycles the freed slab slot between the two unrefs, the trailing unref decrements a different, live packet's reference count and frees it, corrupting the `net_pkt` pool shared by both the receive and transmit paths.

The defect is exercised by ordinary transmission over the native-stack SiWx917 WiFi interface, and an adjacent attacker on the same WiFi network can induce transmissions (for example ARP or ICMP echo replies, or TCP handshakes) to drive the path. The primary observable impact is loss of availability (transmit hangs and crashes from pool corruption), with race-

dependent memory corruption of the kernel networking buffer pool. The fix removes the erroneous `net_pkt_unref(pkt)` from `siwx91x_send()`; the driver's receive-path unref, which correctly frees a packet the driver itself allocated, is unaffected.

- Zephyr project bug tracker GHSA-f9qq-jv4w-pqxg

This has been fixed in main for v4.5.0

- PR 112180 fix for main
- PR 113523 fix for v4.4
- PR 113522 fix for v4.3

CVE 2026-14367

I3C IBI work-node free-list data race between ISR and workqueue thread

The I3C IBI subsystem in `drivers/i3c/i3c_ibi_workq.c` hands out statically-allocated work nodes through a free-list `i3c_ibi_work_nodes_free` implemented as a plain `sys_slist_t`, which provides no synchronization. The allocation helpers (`i3c_ibi_work_enqueue`, `i3c_ibi_work_enqueue_target_irq`, `i3c_ibi_work_enqueue_hotjoin`, `i3c_ibi_work_enqueue_controller_request`, `i3c_ibi_work_enqueue_cb`) called `sys_slist_get()` directly from **ISR context**, while the workqueue handler `i3c_ibi_work_handler()` returned nodes with `sys_slist_append()` from the **workqueue thread**, with no lock on either side.

Because `sys_slist_get()` and `sys_slist_append()` are neither atomic nor interrupt-safe, an IBI interrupt that fires while the workqueue thread is mid-append (or a truly parallel access under `CONFIG_SMP`) races on the shared list. This corrupts the list linkage: a node may be handed to two consumers, a node may be lost, or the head/tail pointers may be left inconsistent so `sys_slist_get()` returns a stale or garbage pointer. In the double-hand-out case the subsequent `memcpy(ibi_node, ibi_work, sizeof(*ibi_node))` overwrites a node still in flight; a garbage pointer turns the same `memcpy` into an out-of-bounds write.

The race is driven by I3C bus traffic — IBIs, hot-joins, and controller-role requests originate from target devices on the bus, and I3C supports hot-joining devices. An attacker controlling an I3C peripheral on the board's chip-to-chip bus can generate high-frequency interrupts timed to collide with the free operation. Exploitation requires physical access to the bus and winning a narrow timing window; the most realistic impact is a crash or hang (denial of service), with memory corruption possible but hard to control.

The fix wraps all free-list `sys_slist_get()`/`sys_slist_append()` operations in the new `ibi_work_alloc()`/`ibi_work_free()` helpers, each guarded by a `k_spinlock` (`ibi_work_lock`), closing the race across ISR and thread contexts.

- Zephyr project bug tracker GHSA-gfj5-gcxv-9jqm

This has been fixed in main for v4.5.0

- PR 110786 fix for main
- PR 113436 fix for v4.4
- PR 113437 fix for v4.3
- PR 117902 fix for v3.7

CVE 2026-14368

Off-by-one out-of-bounds NUL write in Zephyr LwM2M JSON string parser

The LwM2M JSON content formatter's `get_string()` in `subsys/net/lib/lwm2m/lwm2m_rw_json.c` copies a parsed JSON string into a caller-supplied buffer and NUL-terminates it. The length guard used `if (string_length > buflen)`, which accepts a string whose length is exactly `buflen`.

After `memcpy()` fills the whole buffer, `buf[string_length] = '\0'` then writes one byte past the end of the buffer (CWE-787).

The string value and its length are taken directly from the incoming CoAP payload during a LwM2M WRITE: `do_write_op_json()` parses the payload obtained from `coap_packet_get_payload()`, and `get_string()` is invoked from `lwm2m_write_handler()` (`engine_get_string()` in `subsys/net/lib/lwm2m/lwm2m_message_handling.c`) for a `LWM2M_RES_TYPE_STRING` resource. The destination `buf`/`buflen` is either the resource instance's fixed data buffer (`res_inst->data_ptr`/`max_data_len`) or the engine validation buffer (`msg->ctx->validate_buf`). A LwM2M server (the client's DTLS peer) can therefore write a string resource with a value whose length equals the target buffer size and force a one-byte overflow.

The overflow is a single out-of-bounds write of the constant byte `0x00` immediately past the resource or validation buffer, corrupting the adjacent byte in memory. It is not an information leak and the written value is fixed, so it is not a direct code-execution primitive, but it can corrupt adjacent state (an adjacent resource value, a length/flag field, or a struct field) and cause data corruption or a crash. Triggering the write is deterministic; the resulting impact depends on memory layout.

The fix changes the guard to `string_length >= buflen`, rejecting the exact-length case and aligning the JSON formatter with the other content formatters (`lwm2m_rw_plain_text.c`, `lwm2m_rw_oma_tlv.c`, `lwm2m_rw_senml_json.c`, `lwm2m_rw_cbor.c`, `lwm2m_rw_senml_cbor.c`), which already used the correct boundary check.

- Zephyr project bug tracker GHSA-vg53-h6qq-xx7h

This has been fixed in main for v4.5.0

- PR 112021 fix for main
- PR 113446 fix for v4.4
- PR 113444 fix for v4.3
- PR 113519 fix for v3.7

CVE 2026-14696

Ethernet bridge RX packet leak enables denial of service via RX buffer-pool exhaustion

When Ethernet bridging is enabled (`CONFIG_NET_ETHERNET_BRIDGE`), `eth_bridge_input_process()` in `subsys/net/l2/ethernet/bridge/bridge_input.c` decides how each frame received on a bridge member interface is handled. For frames that must also be delivered to the local stack, the code called `eth_bridge_handle_locally()` and returned `NET_OK`. That helper does not consume the packet — it only calls `bridge_iface_recv()` (via `virtual_recv()`), which returns `NET_CONTINUE` without taking ownership of `pkt`.

The `NET_OK` verdict then propagates through `ethernet_recv()` up to `processing_data()` in `subsys/net/ip/net_core.c`, where `NET_OK` is interpreted as "the packet was consumed, do not free it." Because no consumer actually took ownership, the RX `net_pkt` is never returned to the pool and is leaked. The concretely reproducible leak occurs for frames whose EtherType has no registered L3 handler when `CONFIG_NET_ETHERNET_FORWARD_UNRECOGNISED_ETHERTYPE` is set (default `y` when `CONFIG_NET_SOCKETS_PACKET` is enabled): the fall-through L3 dispatch does not overwrite the `NET_OK` verdict, so `ethernet_recv()` returns `NET_OK` and the buffer is never released.

Any device on a bridged L2 segment can emit broadcast/multicast frames carrying an arbitrary EtherType with no authentication. Each such frame permanently consumes one buffer from the finite RX pool (`CONFIG_NET_PKT_RX_COUNT`), so a brief broadcast flood exhausts the pool and the device can no longer receive traffic until it is rebooted — a persistent denial of service. There is no confidentiality or integrity impact.

The fix makes `eth_bridge_handle_locally()` propagate the real `net_verdict` and return `NET_CONTINUE` for locally-kept frames, writing the bridge interface back through a new `dst_iface` out-parameter so the packet follows the normal receive path and is unreferenced exactly once.

- Zephyr project bug tracker GHSA-3m4w-wc4v-766q

This has been fixed in main for v4.5.0

- PR 111931 fix for main
- PR 113524 fix for v4.4

CVE 2026-14697

IPv6 Neighbor Solicitation packet leak causes TX pool exhaustion denial of service

`net_ipv6_send_ns()` in `subsys/net/ip/ipv6_nbr.c` allocates a transmit `net_pkt` for a Neighbor Solicitation. When it is called with a data packet pending on an unresolved neighbor and that neighbor's `pending_queue` is already non-empty (an NS is already outstanding), the function appends the data packet and returns early without ever sending the NS via `net_send_data()` or releasing it with `net_pkt_unref()`. The freshly allocated NS `net_pkt` and its attached TX buffers are held only by a local variable and are leaked permanently, never returning to `CONFIG_NET_PKT_TX_COUNT` / `CONFIG_NET_BUF_TX_COUNT`.

The leaking branch sits on the normal IPv6 transmit path: `net_ipv6_prepare_for_send()` (called from `net_if.c`) invokes `net_ipv6_send_ns()` for any outbound or forwarded IPv6 packet whose next hop is not yet in the neighbor cache. An on-link (adjacent) attacker can drive it deterministically by sending a burst of request packets (for example ICMPv6 echo requests or UDP datagrams) that all spoof a single non-existent on-link source address: the node generates a reply to each, the first reply queues an NS, and every subsequent reply during the roughly three-second `INCOMPLETE` resolution window takes the leaking branch and loses one TX packet. Router-configured nodes forwarding attacker traffic toward a non-existent on-link host leak identically.

Because the leaked packets are never reclaimed and `CONFIG_NET_PKT_TX_COUNT` defaults to only 4 (14 for Ethernet), a brief low-rate burst exhausts the TX pool. Once exhausted the node can no longer allocate any transmit packet and cannot send TCP/UDP, ARP/ND, or any reply at all, producing a complete and persistent network denial of service that does not self-heal until reboot. The fix releases the unsent NS packet with `net_pkt_unref(pkt)` before the early return.

- Zephyr project bug tracker GHSA-x956-p489-8mf5

This has been fixed in main for v4.5.0

- PR 112372 fix for main
- PR 113655 fix for v4.4
- PR 113654 fix for v4.3

CVE 2026-14986

Under embargo until 2026-09-04

CVE 2026-15460

Missing channel-state validation in Zephyr Bluetooth Classic L2CAP receive path

The Bluetooth Classic (BR/EDR) L2CAP receive handler `bt_l2cap_br_recv()` in `subsys/bluetooth/host/classic/l2cap_br.c` dispatched inbound data PDUs based only on the destination channel ID, without checking that the target channel had reached the `BT_L2CAP_CONNECTED` state. A dynamic channel is assigned its RX CID and added to the connection's channel list while still in `BT_L2CAP_CONNECTING` (and later `BT_L2CAP_CONFIG`) — before configuration completes and, for PSMs that require security, before the peer is authenticated (`l2cap_br_conn_req()`).

Because the channel is already findable by `bt_l2cap_br_lookup_rx_cid()` during this window, a remote peer within radio range can send a data PDU addressed to that CID and have it processed on a not-yet-established channel. The dispatch keys off channel fields (`BR_CHAN(chan)->rx.mode`, `rx.mps`) that are only initialized during configuration by `l2cap_br_conf()`; since channel objects are pooled and `bt_l2cap_br_chan_del()` does not reset `rx.mode` or the reassembly buffer `_sdu`, a reused channel can carry stale state into the `CONNECTING` window and route the frame into the retransmission/flow-control path (`bt_l2cap_br_ret_fc_recv()`) with stale parameters and a possibly stale `_sdu` pointer.

The impact is delivery of attacker data to upper-layer protocol handlers on a half-open (and possibly unauthenticated) channel, plus operation on stale or partially initialized channel state on reused channel objects — leading to channel/link teardown (denial of service) and, in the stale-`_sdu` case, a dangling-pointer condition. The fix adds an explicit `BR_CHAN(chan)->state < BT_L2CAP_CONNECTED` guard that drops any data received before the channel is fully connected.

- Zephyr project bug tracker GHSA-hx89-rm6c-hjrh

This has been fixed in main for v4.5.0

- PR 112394 fix for main
- PR 113700 fix for v4.4
- PR 113701 fix for v4.3
- PR 113710 fix for v3.7

CVE 2026-15461

Type confusion in Zephyr HL78xx GNSS NMEA driver causes wild-pointer write from GNSS input

The Sierra Wireless HL78xx modem GNSS driver (`drivers/modem/hl78xx/`, later `drivers/modem/vendor_standalone/hl78xx/`) embeds a generic `struct gnss_nmea0183_match_data match_data` inside `struct hl78xx_gnss_data`. The generic NMEA0183 match helper (`drivers/gnss/gnss_nmea0183_match.c`) requires that context to be the *first* member because its callbacks cast `user_data` directly to `struct gnss_nmea0183_match_data *`. In the affected releases `match_data` was the second member (after `const struct device *dev`), so it sat at a non-zero offset while `gnss_nmea0183_match_init()` initialized it at the correct address. The registered NMEA handlers instead pass the whole device data object (`data->devices.gnss->data`, offset 0), producing an offset-shifted type confusion between where state is initialized and where the parse callbacks read and write it.

When NMEA sentences from the GNSS receiver are parsed, the GGA/RMC callbacks write parsed fix data into the wrong location within the struct, and the GSV callback (`gnss_nmea0183_match_gsv_callback`, active under `CONFIG_GNSS_SATELLITES`) reads its `satellites` pointer and bound from the wrong offsets — non-pointer bytes of `struct hl78xx_gnss_data` — and then writes parsed `struct gnss_satellite` entries through that bogus pointer. This is a write through an uninitialized/wild pointer with a garbage bound.

The NMEA handlers are registered by default (`CONFIG_HL78XX_GNSS_SOURCE_NMEA` is the default GNSS source) on devices using the HL78xx GNSS. The driver runs in kernel context and the NMEA data originates from the GNSS radio front-end, so a party able to influence the GNSS signal (for example GNSS/GPS spoofing at radio proximity) can drive the kernel-side parser into the faulty write. The most likely impact is a crash (denial of service) because the bogus pointer resolves to a fixed near-NULL value, with adjacent-memory corruption possible on MMU-less targets. Confidentiality is not affected. Exploitation requires the satellites feature to be enabled and active, so attack complexity is high.

- Zephyr project bug tracker GHSA-vvjg-6rg4-7235

This has been fixed in main for v4.5.0

- PR 112937 fix for main
- PR 113705 fix for v4.4

CVE 2026-6682

Integer overflow in FatFs FAT32 volume mount (mount_volume) yields an attacker-controlled file size and out-of-bounds access in Zephyr

Zephyr bundles ChaN's FatFs (via the `zephyrproject-rtos/fatfs` west module) as the FAT/exFAT filesystem backing `subsys/fs/fat_fs.c`. In `mount_volume()` (`modules/fs/fatfs/ff.c`), the FAT area size is computed as `fasize = ld_32(BPB_FATSz32); ... fasize *= fs->n_fats;` — a 32-bit multiply with no overflow check.

A crafted FAT32 volume that sets `BPB_FATSz32 = 0x80000001` with two FATs makes the product wrap (to `0x00000002`), so the computed data region overlaps the FAT region. Because the pre-multiply value is kept in `fs->fsize`, the later plausibility check does not catch the wrap.

An attacker who can get the device to mount such a volume (a malicious SD card or USB medium) can place forged directory entries in the overlapped region, causing `f_stat()`/directory reads to return attacker-controlled file sizes; application code that reads a file using that size as a length then overflows its buffers, giving heap- or stack-based memory corruption during ordinary file operations.

This is a core FAT32 code path with no compile-time gate (FAT12/16/32 is always built), so a default Zephyr FatFs configuration is affected. Upstream FatFs is unmaintained for security purposes (the maintainer did not respond to runZero or JPCERT/CC), so Zephyr carries the fix in its vendored copy. Tracked upstream as CVE-2026-6682.

- Zephyr project bug tracker GHSA-m537-wqw2-2wrj

This has been fixed in main for v4.5.0

CVE 2026-6683

Divide-by-zero in FatFs exFAT sync (sync_fs) crashes Zephyr on a crafted exFAT volume

Zephyr's bundled FatFs (`zephyrproject-rtos/fatfs`) supports exFAT when `CONFIG_FS_FATFS_EXFAT` is enabled. In `sync_fs()` (`modules/fs/fatfs/ff.c`) the free-cluster bookkeeping divides by `(fs->n_fatent - 2)`.

The cluster count is read from the exFAT boot region as `ncl = ld_32(BPB_NumClusEx)` and is validated only against an upper bound (`> MAX_EXFAT`), never a lower bound, so a crafted exFAT volume with `BPB_NumClusEx = 0` yields `n_fatent = 2` and a divisor of zero.

Mounting such a volume and performing any write/sync triggers a divide-by-zero (SIGFPE / CPU fault), a denial of service.

The defect is present only when exFAT is compiled in, which is not the default Zephyr configuration; devices that enable exFAT and mount untrusted removable media are exposed. Upstream FatFs is unmaintained for security, so Zephyr carries the fix in its vendored copy. Tracked upstream as CVE-2026-6683.

- Zephyr project bug tracker GHSA-c5j5-mrhx-hjg4

This has been fixed in main for v4.5.0

CVE 2026-6685

Integer underflow in FatFs dirty-sector cache (f_read/f_write) causes wrong-sector I/O on crafted fragmented media in Zephyr

In FatFs's read/write path (`f_read`/`f_write` in `modules/fs/fatfs/ff.c`), the dirty-sector cache-refill decision compares `fp->sect - sect` against the run length `cc` using unsigned arithmetic. On a fragmented FAT layout where a later cluster maps to a lower absolute sector than the currently cached one, the subtraction underflows (wraps to a large unsigned value), so the guard that decides whether the cached window overlaps the requested range is evaluated incorrectly.

The result is that FatFs flushes or reuses the wrong cached sector, reading or writing file data to/from an incorrect on-disk location — cross-file data corruption and potential disclosure of unrelated file contents, and a path to out-of-bounds behaviour during ordinary file operations.

A crafted volume with a deliberately fragmented cluster chain (attacker-controlled removable media) triggers the condition. This is on the default read/write path (no exFAT or LFN gating). Upstream FatFs is unmaintained for security, so Zephyr carries the fix in its vendored copy. Tracked upstream as CVE-2026-6685.

- Zephyr project bug tracker GHSA-rg3p-32gq-hqw3

This has been fixed in main for v4.5.0

CVE 2026-6686

FatFs f_lseek past end-of-file exposes uninitialized/stale cluster contents (deleted file data) in Zephyr

In FatFs (`f_lseek` in `modules/fs/fatfs/ff.c`), seeking a file opened for write to an offset beyond its current end extends the cluster chain via `create_chain()` but does not zero the newly allocated clusters.

FatFs marks the file as larger without initializing the backing sectors, so a subsequent read of the grown region returns whatever was previously on the medium in those clusters — typically the residual contents of deleted files. On a device where a lower-privileged or later actor can read a file that was extended this way, previously deleted or unrelated file data is disclosed (CWE-908, use of uninitialized resource). No memory-safety corruption occurs; the impact is confidentiality of on-media data.

The bug is on the default write path (no exFAT/LFN gating). Upstream FatFs is unmaintained for security, so Zephyr carries the fix in its vendored copy. Tracked upstream as CVE-2026-6686.

- Zephyr project bug tracker GHSA-rjhg-f2h7-rffm

This has been fixed in main for v4.5.0

CVE 2026-6687

Stack buffer overflow in FatFs exFAT volume-label read (f_getlabel) via an unvalidated on-disk length in Zephyr

In FatFs's `f_getlabel()` (`modules/fs/fatfs/ff.c`), the exFAT volume-label copy loop is bounded by the raw on-disk byte `dj.dir[XDIR_NumLabel]` (0–255) rather than the spec maximum of 11 characters.

A crafted exFAT volume that sets `XDIR_NumLabel` to a large value (e.g. 128) makes the loop read label characters past the 32-byte directory entry and write up to that many UTF-decoded characters into the caller-supplied `label[]` buffer, overflowing a typical fixed-size label array (e.g. `char label[12]`/`label[24]`) on the stack — memory corruption with potential control-flow impact.

In Zephyr this is reachable only in downstream applications: `f_getlabel` is compiled solely with `CONFIG_FS_FATFS_EXTRA_NATIVE_API=y`, exFAT must be enabled, and no in-tree Zephyr code calls it (application code supplies the buffer). It is nonetheless a genuine defect in the vendored library, and because upstream FatFs is unmaintained for security Zephyr carries the fix (clamp the label length) in its vendored copy so opted-in applications are protected. Tracked upstream as CVE-2026-6687.

- Zephyr project bug tracker GHSA-fxw6-w668-cgfh

This has been fixed in main for v4.5.0

CVE 2026-15890

Under embargo until 2026-09-21

CVE 2026-15891

NULL pointer dereference in Zephyr MQTT-SN client when removing a non-responsive gateway

The MQTT-SN client keepalive handler `process_ping()` in `subsys/net/lib/mqtt_sn/mqtt_sn.c` removes the gateway record after `PINGREQ` retries are exhausted. It invoked `SYS_SLIST_PEEK_HEAD_CONTAINER(&client->gateways, gw, next)` but discarded the result. That macro is a pure expression that does not assign to `gw`, so `gw` retained its `NULL` initializer regardless of the list contents.

The code then dereferences the NULL `gw` (`gw->gw_id`) and passes it to `mqtt_sn_gw_destroy()`, reaching `k_mem_slab_free(&gateways, NULL)`. With `CONFIG_MEM_SLAB_POINTER_VALIDATE` enabled this triggers `k_panic()`; in the default configuration it performs a write through the NULL pointer (`*(char **)mem = slab->free_list;`) and corrupts the slab free list. The outcome is a crash/kernel panic or, on targets where address 0 is writable, silent memory-allocator corruption.

The vulnerable branch runs whenever the connected MQTT-SN gateway fails to answer keepalive `PINGREQ`s for the configured number of retries. This condition is controlled by the remote peer: a malicious or compromised gateway, or an on-path/adjacent attacker that advertises itself as a gateway and then stops responding (or blackholes the real gateway's `PINGRESP`s), forces the client into the defect. MQTT-SN runs over UDP and no authentication is required.

The impact is a remotely triggerable denial of service (availability) of the affected MQTT-SN client; there is no attacker-controlled data written. The sibling remover `process_advertise()` uses `SYS_SLIST_FOR_EACH_CONTAINER_SAFE` and is not affected. The fix assigns the macro's return value to `gw`.

- Zephyr project bug tracker GHSA-c4g8-4f9p-4746

This has been fixed in main for v4.5.0

- PR 113142 fix for main
- PR 113718 fix for v4.4
- PR 113723 fix for v4.3

CVE 2026-15892

Heap memory leak in mcumgr settings-management handlers on access-hook rejection leads to denial of service

The mcumgr SMP settings-management group handlers `settings_mgmt_read()`, `settings_mgmt_write()`, and `settings_mgmt_delete()` in `subsys/mgmt/mcumgr/grp/settings_mgmt/src/settings_mgmt.c` allocate a `key_name` buffer (and, for read, a `data` buffer) via `k_malloc()` when `CONFIG_MCUMGR_GRP_SETTINGS_BUFFER_TYPE_HEAP` is enabled, relying on the `end:` label to `k_free()` them. When `CONFIG_MCUMGR_GRP_SETTINGS_ACCESS_HOOK` is also enabled and the application access hook rejects a request by returning status `MGMT_CB_ERROR_RC`, the handler executed `return ret_rc;` directly, bypassing `end:` and leaking the heap allocation on every rejected request.

The settings handlers are reachable over the unauthenticated SMP transport (Bluetooth LE, UART, or UDP, depending on product configuration). The access hook is the mechanism applications use to deny unauthorized settings access, and `MGMT_CB_ERROR_RC` is a common rejection style, so an attacker who can send `settings read/write/delete` commands that the hook rejects triggers a heap leak on each attempt.

Because the leaked memory is never reclaimed until reboot, a sustained stream of rejected requests monotonically exhausts the kernel heap until `k_malloc()` fails, denying mcumgr service and impacting any other heap consumer on the device — a denial of service. The impact is availability-only; there is no memory corruption or information disclosure. Only configurations that select the heap buffer type, enable the access hook, and register a hook that returns `MGMT_CB_ERROR_RC` are affected (the default stack buffer type cannot leak).

- Zephyr project bug tracker GHSA-rq68-wgv4-hcq3

This has been fixed in main for v4.5.0

- PR 113178 fix for main
- PR 113507 fix for v4.4
- PR 113506 fix for v4.3
- PR 113505 fix for v3.7

CVE 2026-15893

Zephyr IPv6 Neighbor Discovery zero reachable time from crafted Router Advertisement causes assertion/DoS

`net_if_ipv6_calc_reachable_time()` in `subsys/net/ip/net_if.c` derives a randomized ND reachable time from `ipv6->base_reachable_time` as `min_reachable + sys_rand32_get() % (max_reachable - min_reachable)`, where `min_reachable = base/2` and `max_reachable = 3*base/2` using integer division. When `base_reachable_time` is `1`, both `min_reachable` and the modulus collapse so the function returns `0`, and `net_if_ipv6_set_reachable_time()` stores that `0` into `ipv6->reachable_time`.

The `base_reachable_time` is attacker-controlled: `handle_ra_input()` in `subsys/net/ip/ipv6_nbr.c` accepts the Reachable Time field of an incoming Router Advertisement whenever it is nonzero and `<= MAX_REACHABLE_TIME`, so a single unauthenticated, link-local RA carrying a Reachable Time of `1` drives the computed reachable time to `0`. Router Advertisements are unauthenticated by default and require only adjacency to the target link.

When a neighbor is subsequently confirmed reachable, `net_ipv6_nbr_set_reachable_timer()` reads the value and executes `NET_ASSERT(time, "Zero reachable timeout!")`. On builds with `CONFIG_ASSERT` enabled this triggers a fatal kernel assertion — a remote denial of service; on builds without assertions the reachable timer is armed with `K_MSEC(0)` and fires immediately, forcing reachable neighbors into perpetual re-solicitation (`STALE`), degrading Neighbor Discovery. The impact is limited to availability; there is no memory-safety, confidentiality, or integrity consequence.

- Zephyr project bug tracker GHSA-8v32-9xf8-r765

This has been fixed in main for v4.5.0

- PR 113226 fix for main
- PR 113605 fix for v4.4
- PR 113686 fix for v4.3
- PR 113687 fix for v3.7

CVE 2026-15894

Under embargo until 2026-10-07

CVE 2026-15923

Infinite loop denial of service in Zephyr SDIO byte-I/O from a card-supplied zero max_blk_size

The Zephyr SDIO subsystem function `sdio_io_rw_extended_helper()` in `subsys/sd/sdio.c` finishes transfers with a byte-I/O loop that uses `size = MIN(remaining, func->cis.max_blk_size)` as the per-iteration step. The value `func->cis.max_blk_size` is decoded directly from the SDIO card's CIS FUNCE tuple in `sdio_decode_cis()` and is not validated. When a card reports a maximum block size of zero, `size` is always `0`, `remaining` never decreases, and the loop spins forever.

The loop is reached from the public SDIO client API used by drivers, including `sdio_read_fifo()`, `sdio_write_fifo()`, and the incrementing register read/write helpers, each of which enters the

loop while holding the per-card mutex `func->card->lock`. A card advertising `max_blk_size == 0` therefore hangs the calling thread permanently on its first non-block-aligned transfer and never releases the mutex, denying service to the SDIO peripheral (and any subsystem such as Wi-Fi that depends on it) until the device is reset.

The malicious value must come from the SDIO card itself, so the defect is exploitable where a removable SDIO/combo card slot lets an attacker insert a crafted or malfunctioning card (a physical attack vector); on boards with a soldered SDIO peripheral it is not attacker-influenceable. There is no memory-safety, confidentiality, or integrity impact — only a permanent availability loss. The fix returns `-EIO` when `func->cis.max_blk_size` is zero, before the loop is entered.

- Zephyr project bug tracker GHSA-4pvm-wrcp-jjf5

This has been fixed in main for v4.5.0

- PR 112628 fix for main
- PR 113730 fix for v4.4
- PR 113731 fix for v4.3
- PR 113732 fix for v3.7

CVE 2026-15924

Use-after-free / double-free from unsynchronized concurrent access to the TLS client session cache in Zephyr sockets

Zephyr's TLS socket layer in `subsys/net/lib/sockets/sockets_tls.c` keeps a single process-global array, `client_cache`, of cached client sessions that is shared by every TLS socket context. The functions that mutate and read it — `tls_session_save()`, `tls_session_get()`, `tls_session_cache_reset()`, and the settings restore handler — allocate, free, and dereference each entry's heap buffer (`entry->session`). Before the fix these accesses were serialized only by the *per-socket* context mutex `ctx->lock` (assigned per socket in `ctx_set_lock()`), which provides no mutual exclusion between different sockets touching the shared cache.

Because `CONFIG_NET_SOCKETS_TLS_MAX_CLIENT_SESSION_COUNT` defaults to `1`, any two concurrent client sockets contend for the same slot. A thread in `tls_session_get()` reading `entry->session` inside `mbedtls_ssl_session_load()` can run concurrently with another thread in `tls_session_save()` that selects the same entry for reuse and executes `mbedtls_free(entry->session)` before reallocating — a use-after-free read, and a double-free when two saves evict the same entry. Both corrupt the mbedTLS heap. The cache is reached on ordinary client paths: at connect time via `tls_session_store()`/`tls_session_restore()`, and (on `main`) whenever a TLS 1.3 session ticket arrives during `recv()`/`poll()` via `tls_session_store_current()`.

Exploitation requires an application that opts into per-socket client session caching (the `TLS_SESSION_CACHE` socket option, off by default) and runs concurrent TLS client connections on multiple threads; the timing that opens the window is influenced by the remote peer(s), so a malicious or compromised server can raise session-ticket frequency to widen it. The reliably-demonstrable impact is memory corruption leading to a crash or heap corruption (denial of service). The fix adds a dedicated `session_cache_lock` mutex taken across every accessor of `client_cache`, serializing all reads and frees and closing the race.

- Zephyr project bug tracker GHSA-wcgm-pq6x-v2gf

This has been fixed in main for v4.5.0

- PR 113405 fix for main
- PR 113736 fix for v4.4
- PR 113737 fix for v4.3
- PR 113738 fix for v3.7

CVE 2026-16147

it82xx2 USB device controller submits incomplete OUT transfer buffers, causing use-after-free and event-list corruption

The ITE IT82xx2 USB device-controller driver (`drivers/usb/udc/udc_it82xx2.c`) mishandles multi-packet OUT transfers on non-control endpoints. In `work_handler_out()` the active transfer buffer is obtained with `udc_buf_peek()` (which does not dequeue it); when a full max-packet-size packet arrives but the buffer still has tailroom (the transfer is not yet complete), the pre-fix code both re-arms the endpoint to keep filling that same `buf` via `work_handler_xfer_continue()` and simultaneously hands the same, still-being-filled buffer to the upper stack with `udc_submit_ep_event()`.

Because `udc_submit_ep_event()` transfers ownership of the buffer to the USB device stack (`usbd_event_carrier()` appends `&buf->node` to `uds_ctx->ep_events`, after which the class handler processes and `net_buf_unref()`s it), the driver continues to DMA subsequent host-controlled OUT packets into a buffer the upper stack may already have freed and recycled — a use-after-free write. In addition, since the buffer was never dequeued, the completing packet runs `udc_buf_get()` on the same object and submits it a second time, appending `&buf->node` to the event slist twice (singly-linked-list corruption) and causing a double `net_buf_unref()`.

The IT82xx2 is a USB peripheral controller, so the untrusted USB host controls OUT-transfer packetization and can force this path against any non-control OUT endpoint whose queued buffer exceeds one packet — an ordinary bulk/interrupt pattern. The driver and USB device stack run in kernel context above the external host, giving the host a device-side kernel heap-corruption primitive: a reliable denial of service and, because the written bytes are attacker-controlled, plausible corruption of adjacent `net_buf` pool memory. The vector is physical (USB attach). The fix defers submission until the buffer is completely filled and lets `xfer_work_handler()` drive continuation, so each OUT buffer is submitted to the upper stack exactly once.

- Zephyr project bug tracker GHSA-3q4g-7w6j-8qfp

This has been fixed in main for v4.5.0

- PR 113463 fix for main
- PR 113847 fix for v4.4
- PR 113850 fix for v4.3
- PR 113849 fix for v3.7

CVE 2026-16148

Kernel panic in the it82xx2 USB device controller driver via re-initialization of a busy delayable work item

The ITE it82xx2 USB device-controller driver initialized its bus-suspend detection work with `k_work_init_delayable(&priv->suspended_work, suspended_handler)` inside `it82xx2_enable()` (the driver's `.enable` op) in `drivers/usb/udc/udc_it82xx2.c`. This work item is scheduled essentially continuously while the USB bus is active: the interrupt handler reschedules it on every SOF frame and `suspended_handler()` reschedules itself, so its timeout node is normally linked in the kernel timeout list / a workqueue pending queue.

`k_work_init_delayable()` (`kernel/work.c`) unconditionally overwrites the entire `k_work_delayable` structure, including its timeout and queue linkage, with no busy check. Because `it82xx2_disable()` does not cancel the work, a normal disable-then-enable cycle re-runs `api->enable()` (`udc_enable()` only rejects a redundant enable, not a re-enable after disable) and re-initializes the still-pending work in place, corrupting the kernel timeout/workqueue linked lists and causing a kernel panic.

An external USB host — for example a host performing USB DFU detach (`dfu-util --detach`) or forcing repeated attach/reset/re-enumeration — drives the `udc_disable()`/`udc_enable()` transitions and controls suspend/resume timing, so it can arrange for the suspend work to be pending

across a re-enable. This yields an unauthenticated denial of service (kernel panic) reachable across the USB boundary from a removable, physically-connected host, with no confidentiality or integrity impact demonstrated.

The fix moves the `k_work_init_delayable()` call into the one-time preinit function so the work is initialized exactly once, eliminating the re-initialization of an in-use item.

- Zephyr project bug tracker GHSA-fvp9-j2pq-477x

This has been fixed in main for v4.5.0

- PR 113463 fix for main
- PR 113847 fix for v4.4
- PR 113850 fix for v4.3
- PR 113849 fix for v3.7

CVE 2026-16511

Under embargo until 2026-11-02

CVE 2026-16512

Under embargo until 2026-09-18

CVE 2026-16513

Under embargo until 2026-09-25

CVE 2026-16514

Under embargo until 2026-09-18

CVE 2026-16515

Under embargo until 2026-09-18

CVE 2026-17050

Under embargo until 2026-09-19

CVE 2026-17051

Under embargo until 2026-09-20

CVE 2026-17052

Under embargo until 2026-09-20

CVE 2026-17053

Under embargo until 2026-09-20

CVE 2026-17054

Under embargo until 2026-09-21

CVE 2026-2411

Bluetooth GATT notify/indicate enforces the wrong attribute's permissions, bypassing encryption/authentication requirements on characteristic values

Zephyr's Bluetooth host declares a GATT characteristic as two consecutive attributes: a Characteristic Declaration whose permission is hard-coded to `BT_GATT_PERM_READ`, and a Characteristic Value attribute that carries the application-specified security permissions (e.g. `BT_GATT_PERM_READ_ENCRYPT` / `READ_AUTHEN` / `READ_LESC`). The public notify and indicate APIs explicitly accept either attribute, and passing the declaration is the documented, common idiom. Before sending each notification or indication, the host re-checks link security with `bt_gatt_check_perm()` against `params->attr` in `gatt_notify()`, `gatt_indicate()`, and `gatt_notify_multiple_verify_params()` (`subsys/bluetooth/host/gatt.c`).

When the application passed the Characteristic Declaration attribute, the host correctly redirected the value handle but left `params->attr` pointing at the declaration, so the security check evaluated the declaration's permissions (no security required) instead of the value's. As a result the encryption/authentication/LESC requirement configured on the characteristic value was skipped. The Notify-Multiple path additionally used a mask that omitted the LE Secure Connections requirement.

A remote peer triggers the disclosure by connecting (optionally without pairing or encryption) and writing the Client Characteristic Configuration descriptor to enable notifications or indications, causing the server to emit the protected value over a link that has not reached the required security level. The impact is information disclosure / access-control bypass for characteristic values the application intended to expose only over a secured link; exposure depends on the application declaring encrypt/authen-required notify/indicate characteristics and on the CCC being writable at a lower security tier. There is no memory-safety or availability impact.

The fix adds `bt_gatt_attr_resolve_value()`, which maps a declaration attribute to the following value attribute before the permission check, and switches the Notify-Multiple path to the full `BT_GATT_PERM_READ_ENCRYPT_MASK` so the LESC requirement is also enforced.

- Zephyr project bug tracker GHSA-4w3r-v9q9-4462

This has been fixed in main for v4.5.0

- PR 108371 fix for main
- PR 111535 fix for v4.4
- PR 111536 fix for v4.3
- PR 111620 fix for v3.7

CVE 2026-18413

Under embargo until 2026-09-26

CVE 2026-18414

Under embargo until 2026-09-26

CVE 2026-18415

Under embargo until 2026-09-26

CVE 2026-18416

Under embargo until 2026-09-26

CVE 2026-18417

Under embargo until 2026-09-27

CVE 2026-18418

Under embargo until 2026-10-11

CVE 2026-18746

Under embargo until 2026-09-28

CVE 2026-18747

Under embargo until 2026-09-28

CVE 2026-18748

Under embargo until 2026-10-20

CVE 2026-19184

Under embargo until 2026-10-04

CVE 2026-19185

Under embargo until 2026-10-04

CVE 2026-19186

Under embargo until 2026-10-07

CVE 2026-19569

Under embargo until 2026-10-09

CVE 2026-19570

Under embargo until 2026-10-09

CVE 2026-19571

Under embargo until 2026-10-09

CVE 2026-19574

Under embargo until 2026-10-09

CVE 2026-19575

Under embargo until 2026-10-09

CVE 2026-19576

Under embargo until 2026-10-10

CVE 2026-19577

Under embargo until 2026-10-10

CVE 2026-19595

Under embargo until 2026-10-18

CVE 2026-19669

Under embargo until 2026-10-10

CVE 2026-19673

Under embargo until 2026-10-14

CVE 2026-19676

Under embargo until 2026-10-17

CVE 2026-19735

Under embargo until 2026-10-11

CVE 2026-19736

Under embargo until 2026-10-11

CVE 2026-19737

Under embargo until 2026-10-11

CVE 2026-19738

Under embargo until 2026-10-11

CVE 2026-19739

Under embargo until 2026-10-11

CVE 2026-19740

Under embargo until 2026-10-11

CVE 2026-19741

Under embargo until 2026-10-22

CVE 2026-19742

Under embargo until 2026-10-25

CVE 2026-19809

Under embargo until 2026-10-19

CVE 2026-19935

Under embargo until 2026-10-11

CVE 2026-19936

Under embargo until 2026-10-12

CVE 2026-19937

Under embargo until 2026-10-12

CVE 2026-19938

Under embargo until 2026-10-12

CVE 2026-19939

Under embargo until 2026-10-13

CVE 2026-19940

Under embargo until 2026-10-13

CVE 2026-19947

Under embargo until 2026-10-21

CVE 2026-75083

Under embargo until 2026-10-14

CVE 2026-75084

Under embargo until 2026-10-14

CVE 2026-75085

Under embargo until 2026-10-14

CVE 2026-76787

Under embargo until 2026-10-16

CVE 2026-76788

Under embargo until 2026-10-16

CVE 2026-77684

Under embargo until 2026-10-18

CVE 2026-77685

Under embargo until 2026-10-18

CVE 2026-78116

Under embargo until 2026-10-20

CVE 2026-78117

Under embargo until 2026-10-20

CVE 2026-78118

Under embargo until 2026-10-20

CVE 2026-78119

Under embargo until 2026-10-20

CVE 2026-79976

Under embargo until 2026-10-21

CVE 2026-79977

Under embargo until 2026-10-22

CVE 2026-79978

Under embargo until 2026-10-23

CVE 2026-79979

Under embargo until 2026-10-23

CVE 2026-79980

Under embargo until 2026-10-23

CVE 2026-79981

Under embargo until 2026-10-23

CVE 2026-79982

Under embargo until 2026-10-23

CVE 2026-81038

Under embargo until 2026-10-24

CVE 2026-81039

Under embargo until 2026-10-24

CVE 2026-82388

Under embargo until 2026-10-26

CVE 2026-82389

Under embargo until 2026-10-26

CVE 2026-82390

Under embargo until 2026-10-26

CVE 2026-82391

Under embargo until 2026-10-26

CVE 2026-82961

Under embargo until 2026-10-27

CVE 2026-82962

Under embargo until 2026-10-27

CVE 2026-85032

Under embargo until 2026-10-30

CVE 2026-85033

Under embargo until 2026-10-31

CVE 2026-85034

Under embargo until 2026-10-31

CVE 2026-85035

Under embargo until 2026-10-31

CVE 2026-85036

Under embargo until 2026-10-31

CVE 2026-86086

Under embargo until 2026-12-01

CVE 2026-86092

Under embargo until 2026-12-03

CVE 2026-87038

Under embargo until 2026-11-02

CVE 2026-87039

Under embargo until 2026-11-02

CVE 2026-87040

Under embargo until 2026-11-02

CVE 2026-87041

Under embargo until 2026-11-02

CVE 2026-87042

Under embargo until 2026-11-03

CVE 2026-87043

Under embargo until 2026-11-03

CVE 2026-87044

Under embargo until 2026-11-03

CVE 2026-87045

Under embargo until 2026-11-06

CVE 2026-90585

Under embargo until 2026-11-07

CVE 2026-90586

Under embargo until 2026-11-08

CVE 2026-90587

Under embargo until 2026-11-08

CVE 2026-90588

Under embargo until 2026-11-08

CVE 2026-90589

Under embargo until 2026-11-08

CVE 2026-90590

Under embargo until 2026-11-09

CVE 2026-90591

Under embargo until 2026-11-09

CVE 2026-90832

Under embargo until 2026-11-10

CVE 2026-90833

Under embargo until 2026-11-10

CVE 2026-90834

Under embargo until 2026-11-10

CVE 2026-91007

Under embargo until 2026-11-12

9.8 Security standards and Zephyr

Historically, organizations managed cybersecurity independently, defining their own assessment and response protocols. Today, governments are increasingly regulating this area through new mandates and security standards. These standards define specific guidelines and compliance requirements for connected products.

This section evaluates the implications of these security standards for the Zephyr project itself, as well as downstream product developers. The goal is to provide developers with the information needed to build certifiable, compliant products using Zephyr.

9.8.1 EU Cyber Resilience Act (CRA)

Warning

This document is for informational purposes only and does not constitute legal advice. Consult with your legal counsel for compliance guidance specific to your situation. You can consult European Union accredited laboratories (see List of Notified Bodies for CRA) that can help you.

Overview

The Cyber Resilience Act ([?]) is an EU regulation that establishes cybersecurity requirements for products with digital elements (PDEs) placed on the EU market. It entered into force on December 10, 2024.

Key Dates

- **June 11, 2026**: Assessment bodies operational; market surveillance authorities defined
- **September 11, 2026: Manufacturers must report actively exploited vulnerabilities and severe** incidents
- **December 11, 2027**: Full regulation applies to products placed on the market from that date, or placed on the market before that date and substantially modified afterwards.

This page explains how the CRA relates both to manufacturers using Zephyr in commercial products, and to the Zephyr Project itself in its role as an open-source software steward.

For manufacturers, the CRA imposes essential cybersecurity requirements (Annex I Part I) along with vulnerability handling and reporting obligations (Annex I Part II). For the Zephyr Project as an open-source software steward, the CRA introduces a tailored set of obligations, including maintaining a cybersecurity policy, reporting actively exploited vulnerabilities and severe incidents, and cooperating with market surveillance authorities.

For Manufacturers Using Zephyr

Does the CRA apply to my product? The CRA applies if you place a product with digital elements (PDE) on the EU market for commercial purposes. This includes hardware devices with embedded software, and standalone software products.

A product with a digital element is hardware and/or software that is connected to another device or network. The scope of the product includes the remote data processing solution (RDPS): software not part of the product but required for it to function (for example, a mobile phone app or a cloud service).

The CRA applies to end products or components that are placed on the market in the European Union. According to definition in Blue Guide and EU CRA Guidelines: A product is placed on the market when it is made available for the first time on the Union market. A product is made available on the Union market when it is supplied for distribution, consumption or use on the Union market in the course of a commercial activity, whether in return for payment or free of charge. Placing on the market is applicable to every individual unit of a product series; development or design dates are not relevant. More information about "Placing on the market" can be found in the EU CRA Guidelines. It details specific cases like standalone software, source code releases and complex products.

Several market exceptions are defined in the law (medical, automotive, avionics, military, spare parts). These exceptions have strict conditions; refer to the CRA text (Article 2).

Which category does my product belong to? The CRA classifies products into categories based on risk: **Important Products** (Annex III) and **Critical Products** (Annex IV). Products not listed in either category are considered **Default** products and have lower requirements.

For example, default products can typically rely on self-assessment (see *Which compliance path must I choose?*) with fewer documentation and assurance requirements.

Category	Short description	Example Zephyr Use Cases
Default	Products with digital elements that are not listed as "important" or "critical".	• Wi-Fi smart bulb or switch (e.g., running Matter over Thread/Wi-Fi). • Wearable activity tracker or smartwatch for personal wellness. • Bluetooth LE audio accessory or wireless sensor tag. • Temperature sensor connected to a CAN bus.
Important (Class I)	Higher-risk products, often consumer-facing, performing security- or access-relevant functions.	• Smart door lock or access control reader for residential use. • Router managing network traffic. • Connected alarm system or security sensor. • Micro-controllers with AVA_VAN.1 protection level.
Important (Class II)	Higher-risk products used in enterprise/industrial/infra contexts or with privileged network roles.	• Micro-controllers with AVA_VAN.2 or AVA_VAN.3 protection level.
Critical	Products whose compromise could severely impact critical infrastructure or essential services.	• Smart meter gateways. • Hardware Security Module (HSM) or smartcard firmware.

Core Functionality vs. Integration

Classification is determined by the **core functionality** of the final product, not by the individual components it integrates (Article 7).

- Integrating an important or critical component (e.g., a secure element, embedded browser) into another product does **not** automatically render that product important or critical.
- A product that *can* perform the functions of an important/critical category but whose core functionality is different is **not** considered to have that core functionality.

Examples:

- If you build a network firewall using Zephyr, the core function is security, making the product an Important Product (Class II).
- If you build a coffee machine using Zephyr, and you utilize Zephyr's networking stack features to protect the device, the core function remains making coffee. This is a "default" product.

In a nutshell, using security-critical Zephyr features (like cryptography or secure boot) in a device does **not** elevate that device to a higher risk class.

For detailed classification, see Annex III and Annex IV. The full list of product categories with technical descriptions is provided in Implementing Regulation (EU) 2025/2392.

Which compliance path must I choose? The CRA defines different conformity assessment procedures based on your product category. You must choose the path that corresponds to your classification and reliance on harmonized standards.

Table 1: CRA Product Categories & Assessment Routes

Category	Conformity Assessment Procedure	Notified Body involvement?
Default	Module A (Self-assessment by the manufacturer) or Module B + Module C, or Module H.	Not Mandatory for Module A.
Important Class I	Module A **ONLY IF** harmonized standards cited in the Official Journal of the European Union are fully applied. Otherwise: Module B + Module C, or Module H.	Not Mandatory for Module A.
Important Class II	Module B + Module C, or Module H. (Self-assessment is **NOT** allowed).	Yes (Mandatory)
Critical	**European Cybersecurity Certification** (e.g., EUCC) or Module B + Module C + Module H (pending delegate acts).	Yes (Mandatory)

The "Modules" refer to specific assessment procedures defined in Decision No 768/2008/EC ("New Legislative Framework") and adapted by the CRA in Annex VIII:

- Module A (Internal production control): You create the technical documentation, perform the risk assessment, and declare conformity yourself. No external auditor is required.
- Module B (EC-type examination) + Module C (Conformity to type): A Notified Body[1] examines the technical design (Module B) and issues a certificate. You then ensure production conforms to that type (Module C).
- Module H (Full Quality Assurance): A Notified Body[1] audits your Quality Management System (QMS) governing design, production, and testing.

[1] Notified Bodies will become operational by **June 11, 2026**.

What are my main obligations as a manufacturer? The CRA defines manufacturer obligations primarily in Article 13 (product requirements and due diligence) and Article 14 (vulnerability handling and reporting). Regardless of a product's classification, the following core obligations apply to all manufacturers of products with digital elements.

Risk assessment
Assess and document cybersecurity risks throughout the product lifecycle.

Due diligence
Exercise due diligence when integrating third-party components (including open-source software like Zephyr).

Vulnerability handling
Handle vulnerabilities when placing a product on the market. Monitor vulnerabilities during the support period (lifetime or 5 years minimum). Vulnerability handling means identifying hardware and software vulnerabilities (for example, via an SBOM), performing a risk assessment for your product, and providing a software update when the risk cannot be accepted.

Incident reporting
Report actively exploited vulnerabilities affecting your product, as well as severe incidents affecting the project's infrastructure. This obligation continues after the end of the support period.

Technical documentation
Create documentation per Article 31 and Annex VII.

Conformity assessment
Assess conformity per Article 32 and Annex VIII.

CE marking
Affix CE mark and draw up EU declaration of conformity.

What are the penalties for non-compliance?

- Up to EUR 15,000,000 or 2.5% of worldwide annual turnover (Article 13 & Article 14 violations).
- Up to EUR 10,000,000 or 2% of turnover (violations of other obligations).

What are the vulnerability reporting obligations? The CRA distinguishes between "ordinary" vulnerabilities (handled through your normal vulnerability management process) and cases that trigger strict notification timelines under Article 14: **actively exploited vulnerabilities** and **severe incidents**.

The tables below summarize the minimum reporting steps.

Table 2: Actively exploited vulnerabilities (Article 14 (1) and (2))

Step	Deadline	Content (minimum)
Early warning	Within 24 hours of becoming aware	Notify via the single reporting platform to your CSIRT coordinator and ENISA that an actively exploited vulnerability affects your product. Indicate, where known, in which Member States the product is available.
Vulnerability notification	Within 72 hours of becoming aware	Provide general information on the affected product, the general nature of the exploit and the vulnerability, corrective or mitigating measures already taken, and measures users can take. Indicate, where applicable, how sensitive you consider the notified information to be.
Final report	No later than 14 days after a corrective or mitigating measure is available	Describe the vulnerability, its severity and impact, any information on malicious actors exploiting it (if available), and details on the security update or other corrective measures made available.

Table 3: Severe incidents (Article 14 (3) through (6))

Step	Deadline	Content (minimum)
Early warning	Within 24 hours of becoming aware	Notify via the single reporting platform to your CSIRT coordinator and ENISA that a severe incident impacts the security of your product. Indicate whether it is suspected to be caused by unlawful or malicious acts and, where known, in which Member States the product is available.
Incident notification	Within 72 hours of becoming aware	Provide general information on the nature of the incident, an initial assessment (including impact/severity as known at the time), corrective or mitigating measures already taken, and measures users can take. Indicate, where applicable, how sensitive you consider the notified information to be.
Final report	Within 1 month after the incident notification	Provide a detailed description of the incident, including severity and impact, the type of threat or likely root cause, and applied and ongoing mitigation measures.

How can I obtain an SBOM (Software Bill of Materials)? Zephyr can automatically generate SBOMs for your application using the `west spdx` command.

See *Software bill of materials: west spdx* for details on how to configure and use this tool.

How should I handle Zephyr vulnerabilities? As a manufacturer integrating Zephyr into a product, you remain responsible for vulnerability management and, where applicable, CRA reporting. Zephyr provides vulnerability information, but you must assess and act on it for your own product.

A practical workflow is:

1. **Stay informed**. Register to the Zephyr Vulnerability Alert Registry to receive notifications when vulnerabilities are disclosed.
2. **Assess impact**. For each advisory, use your SBOM and configuration to determine whether the affected Zephyr component is present, reachable, and security-relevant in your product.

3. **Plan remediation**. Decide on the appropriate response (e.g., apply a patch, adjust configuration, ...).
4. **Deploy fixes**. Integrate, test, and roll out the chosen fix, and update your SBOM and product documentation as needed.
5. **Meet reporting obligations**. If the vulnerability affects your product and is actively exploited, or leads to a severe incident, report it in line with the Article 14 timelines and as per the previous section, *What are the vulnerability reporting obligations?*.

What timelines does Zephyr follow for vulnerability handling? Zephyr operates its own PSIRT process with target timelines for triage, notification, and disclosure. These are *project* timelines, not legal deadlines for manufacturers.

Your CRA reporting obligations as per Article 14 are triggered by when *you* become aware that your product is affected by an actively exploited vulnerability or a severe incident. This can be **earlier than** some of the Zephyr milestones below, meaning you might have to send an early warning or incident report even before a Zephyr fix is available or before public disclosure.

Zephyr uses private GitHub security advisories and an embargo period (at most 90 days) to coordinate fixes and disclosure. While the full process is described in *Security Vulnerability Reporting*, the key milestones are:

- **Within 7 days**: PSIRT feedback to initial reporter.
- **Within 30 days**: Manufacturers notified via alert registry and fix made available from the project.
- **Up to 90 days total**: Security-sensitive vulnerabilities are made public after embargo period.

Do I need to report vulnerabilities I find in Zephyr? **Yes**, under Article 13(6), if you discover a vulnerability in a component (including Zephyr) integrated into your product, you **must** report it. What's more, if you develop a fix for that vulnerability, you must also share the relevant code or documentation. See *Security Vulnerability Reporting*.

Additionally, consider voluntary reporting to CSIRT or ENISA per Article 15.

For Zephyr as an Open Source Steward

What is Zephyr's role under the CRA? Zephyr is an **"open-source software steward"** under Article 3 (14): a legal person that systematically provides sustained support for developing PDEs intended for commercial activities.

Zephyr's obligations under Article 24:

Cybersecurity policy
: Document security policies and vulnerability handling.

Cooperation
: Work with market surveillance authorities to mitigate risks.

Incident reporting
: Report actively exploited vulnerabilities for the project and severe incidents affecting Zephyr's infrastructure (to the extent Zephyr is involved).

Does the CRA apply to Zephyr contributors? **No.** The CRA does not apply to individual contributors to Zephyr (Recital 18).

Contributors developing features or fixing bugs are not subject to CRA obligations.

How is Zephyr meeting its steward obligations?

Article 24(1): Security policy (Complete)

- Documented at *Zephyr Security Overview*
- Vulnerability reporting process: *Security Vulnerability Reporting*
- Secure coding guidelines: *Secure Coding*

Article 24(2): Cooperation with authorities (In Progress)

- Registered CVE Numbering Authority (CNA) since 2017
- Active Zephyr Project Security Incident Response Team (PSIRT)
- **In Progress**: Identifying CSIRT coordinator for EU

Article 14(1) & Article 14(3): Incident reporting (In Progress)

- **In Progress**: Determining if NVD processes work for CSIRT/ENISA reporting
- Plan to align with EU reporting requirements

Article 14(8): User notification (Complete)

- Vulnerability Alert Registry for manufacturers and integrators
- CVE publication and security advisories

Article 52(3): Corrective actions (Complete)

- Established processes in place with CVE authorities
- Timely response through PSIRT

External Resources

Official CRA documentation

- EU CRA Regulation 2024/2847
- Implementing Regulation (EU) 2025/2392 (Technical descriptions of important and critical product categories)
- ENISA CRA Requirements-Standards Mapping
- European Commission CRA FAQ
- European Commission CRA Guidelines (answer a lot of questions about interpration)

Standards and Technical Specifications Relevant existing standards:

ETSI and CEN/CENELC are developing harmonized standards in response to the CRA Standardisation Request (M/606).

CRA horizontal standards developed by CEN/CENELEC:

- EN40000-1-1 : Vocabulary definition
- EN40000-1-2 : Quality process for the development of Secure by Design product. It includes risk assessment, sourcing due diligence, development, test, validation, manufacturing, decommissioning…
- EN40000-1-3 : Vulnerability handling process
- EN40000-1-4 : Security controls : technical solutions to reduce cybersecurity risk. This is based on the hEN18031-x standard developed for the CE RED Delegated Act.

Presentation of early draft of EN40000-1-4 is available on CEN/CENELEC website: Security Controls.

ETSI Public draft standards include product-specific requirements for:

- Operating Systems (prEN 304 626)
- Browsers (prEN 304 617)
- Password Managers (prEN 304 618)
- Firewalls (prEN 304 636)

For the complete list of draft standards and participation in public consultations, see the ETSI Cyber Resilience Act Portal.

- Once harmonized, these standards could be freely accessible to EU, EFTA, and UK citizens via Harmonized standards.

Educational resources

- Linux Foundation: Understanding the EU CRA
- Linux Foundation CRA Readiness Report
- Linux Foundation CRA Compliance Best Practices
- OpenSSF CRA Guidance

Zephyr-specific resources

- *Zephyr Security Overview*
- *Security Vulnerability Reporting*
- Zephyr Vulnerability Alert Registry
- *Zephyr Vulnerabilities*

9.8.2 ETSI 303-645

ETSI EN 303 645, also known as "Cyber Security for Consumer Internet of Things: Baseline Requirements," is a standard developed by the European Telecommunications Standards Institute (ETSI).

The standard includes provisions for secure software updates, data protection, secure communication, and the minimization of exposed attack surfaces, among other things. It is part of a broader effort to address the challenges and risks associated with IoT devices.

Full version of the standard can be found here.

Terminology

Table 4: ETSI 303645 terminology

administrator	user who has the highest-privilege level possible for a user of the device, which can mean they are able to change any configuration related to the intended functionality.
associated services	digital services that, together with the device, are part of the overall consumer IoT product and that are typically required to provide the product's intended functionality.
authentication mechanism	method used to prove the authenticity of an entity.
authentication value	individual value of an attribute used by an authentication mechanism.
best practice cryptography	cryptography that is suitable for the corresponding use case and has no indications of a feasible attack with current readily available techniques.

continues on next page

Table 4 – continued from previous page

constrained device	device which has physical limitations in either the ability to process data, the ability to communicate data, the ability to store data or the ability to interact with the user, due to restrictions that arise from its intended use.
consumer	natural person who is acting for purposes that are outside her/his trade, business, craft or profession.
consumer IoT device	network-connected (and network-connectable) device that has relationships to associated services and are used by the consumer typically in the home or as electronic wearables.
critical security parameter	security-related secret information whose disclosure or modification can compromise the security of a security module.
debug interface	physical interface used by the manufacturer to communicate with the device during development or to perform triage of issues with the device and that is not used as part of the consumer-facing functionality
defined support period	minimum length of time, expressed as a period or by an end-date, for which a manufacturer will provide security updates.
device manufacturer	entity that creates an assembled final consumer IoT product, which is likely to contain the products and components of many other suppliers.
factory default	state of the device after factory reset or after final production/assembly.
initialization	process that activates the network connectivity of the device for operation and optionally sets authentication features for a user or for network access.
initialized state	state of the device after initialization.
IoT product	consumer IoT device and its associated services.
isolable	able to be removed from the network it is connected to, where any functionality loss caused is related only to that connectivity and not to its main function; alternatively, able to be placed in a self-contained environment with other devices if and only if the integrity of devices within that environment can be ensured
logical interface	software implementation that utilizes a network interface to communicate over the network via channels or ports.
manufacturer	relevant economic operator in the supply chain (including the device manufacturer).
network interface	physical interface that can be used to access the functionality of consumer IoT via a network.
owner	user who owns or who purchased the device.
personal data	Any information relating to an identified or identifiable natural person.
physical interface	physical port or air interface (such as radio, audio or optical) used to communicate with the device at the physical layer.
public security parameter	security related public information whose modification can compromise the security of a security module.
remotely accessible	intended to be accessible from outside the local network.
security module	set of hardware, software, and/or firmware that implements security functions.
security update	software update that addresses security vulnerabilities either discovered by or reported to the manufacturer.
sensitive security parameters	critical security parameters and public security parameters.
software service	software component of a device that is used to support functionality.
telemetry	data from a device that can provide information to help the manufacturer identify issues or information related to device usage.

continues on next page

Table 4 – continued from previous page

unique per device	unique for each individual device of a given product class or type.
user	natural person or organization.

Provisions Assessment

The following table is a self-assessment using Table B.1 from ETSI EN 303 645, specifically focusing on the Zephyr RTOS as a component within IoT products.

According with ETSI 303 645, table B.1 provides a mechanism to give information about the implementation of the provisions presented in the standard. Zephyr has adopted the following notations used in the standard:

Table 5: ETSI 303645 Table B.1 notations

M	the provision is a mandatory requirement
R	the provision is a recommendation
M C	the provision is a mandatory requirement and conditional
R C	the provision is a recommendation and conditional
Y	The provision is supported by Zephyr
N	The provision is not supported by Zephyr
N/A	The provision is not applicable to Zephyr or it is product makers responsibility

Table 6: ETSI 303645 provisions assessment using table B.1

Provision	Description	Status	Support	Detail
Provision 5.1-1	Where passwords are used and in any state other than the factory default, all consumer IoT device passwords shall be unique per device or defined by the user.	M C	N/A	
Provision 5.1-2	Where pre-installed unique per device passwords are used, these shall be generated with a mechanism that reduces the risk of automated attacks against a class or type of device.	M C	N/A	
Provision 5.1-3	Authentication mechanisms used to authenticate users against a device shall use best practice cryptography, appropriate to the properties of the technology, risk and usage.	M	N/A	
Provision 5.1-4	Where a user can authenticate against a device, the device shall provide to the user or an administrator a simple mechanism to change the authentication value used.	M C	N/A	
Provision 5.1-5	When the device is not a constrained device, it shall have a mechanism available which makes brute-force attacks on authentication mechanisms via network interfaces impracticable.	M C	N	**TODO**
Provision 5.2-1	The manufacture shall make a vulnerability disclosure policy publicly available.	M	Y	*Vulnerability Management*
Provision 5.2-2	Disclosed vulnerabilities should be acted on in a timely manner.	R	Y	*Vulnerability Timeline*
Provision 5.2-3	Manufacturers should continually monitor for, identify and rectify security vulnerabilities within products and services they sell, produce, have produced and services they operate during the defined support period.	R	Y	Modules are covered
Provision 5.3-1	All software components in consumer IoT devices should be securely updatable.	R	Y	*Device firwmware upgrade*
Provision 5.3-2	When the device is not a constrained device, it shall have an update mechanism for the secure installation of updates.	M C	Y	*Device firwmware upgrade*
Provision 5.3-3	An update shall be simple for the user to apply.	M C	N/A	

continues on next page

Table 6 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.3-4	Automatic mechanisms should be used for software updates.	R C	N/A	
Provision 5.3-5	The device should check after initialization, and then periodically, whether security updates are available.	R C	N/A	
Provision 5.3-6	If the device supports automatic updates and/or update notifications, these should be enabled in the initialized state and configurable so that the user can enable, disable, or postpone installation of security updates and/or update notifications.	R C	N/A	
Provision 5.3-7	The device shall use best practice cryptography to facilitate secure update mechanisms.	M C	Y	*West Sign*
Provision 5.3-8	Security updates shall be timely.	M C	N/A	
Provision 5.3-9	The device should verify the authenticity and integrity of software updates.	R C	Y	Functionality provided by MCUboot. Also see *Device Firwmware Upgrade*.
Provision 5.3-10	Where updates are delivered over a network interface, the device shall verify the authenticity and integrity of each update via a trust relationship.	M	N/A	
Provision 5.3-11	The manufacturer should inform the user in a recognizable and apparent manner that a security update is required together with information on the risks mitigated by that update.	R C	N/A	
Provision 5.3-12	The device should notify the user when the application of a software update will disrupt the basic functioning of the device.	R C	N/A	Zephyr provides this information for its updates. Anyone using Zephyr in their products must check if they are affected
Provision 5.3-13	The manufacturer shall publish, in an accessible way that is clear and transparent to the user, the defined support period.	M	Y	Release Life Cycle and Maintenance
Provision 5.3-14	For constrained devices that cannot have their software updated, the rationale for the absence of software updates, the period and method of hardware replacement support and a defined support period should be published by the manufacturer in an accessible way that is clear and transparent to the user.	R C	N/A	

continues on next page

Table 6 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.3-15	For constrained devices that cannot have their software updated, the product should be isolable and the hardware replaceable.	R C	N/A	
Provision 5.3-16	The model designation of the consumer IoT device shall be clearly recognizable, either by labelling on the device or via a physical interface.	M	N/A	
Provision 5.4-1	Sensitive security parameters in persistent storage shall be stored securely by the device.	M	N	There is not secure storage within Zephyr
Provision 5.4-2	Where a hard-coded unique per device identity is used in a device for security purposes, it shall be implemented in such a way that it resists tampering by means such as physical, electrical or software.	M C	N/A	
Provision 5.4-3	Hard-coded critical security parameters in device software source code shall not be used.	M	Y	*Hardening Tool*
Provision 5.4-4	Any critical security parameters used for integrity and authenticity checks of software updates and for protection of communication with associated services in device software shall be unique per device and shall be produced with a mechanism that reduces the risk of automated attacks against classes of devices.	M	N/A	
Provision 5.5-1	The consumer IoT device shall use best practice cryptography to communicate securely.	M	Y	
Provision 5.5-2	The consumer IoT device should use reviewed or evaluated implementations to deliver network and security functionalities, particularly in the field of cryptography.	R	Y	
Provision 5.5-3	Cryptographic algorithms and primitives should be updatable.	R	N	The whole image must be updated
Provision 5.5-4	Access to device functionality via a network interface in the initialized state should only be possible after authentication on that interface.	R	N/A	

continues on next page

Table 6 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.5-5	Device functionality that allows security-relevant changes in configuration via a network interface shall only be accessible after authentication. The exception is for network service protocols that are relied upon by the device and where the manufacturer cannot guarantee what configuration will be required for the device to operate.	M	N/A	
Provision 5.5-6	Critical security parameters should be encrypted in transit, with such encryption appropriate to the properties of the technology, risk and usage.	R	Y	
Provision 5.5-7	The consumer IoT device shall protect the confidentiality of critical security parameters that are communicated via remotely accessible network interfaces.	M	Y	
Provision 5.5-8	The manufacturer shall follow secure management processes for critical security parameters that relate to the device.	M	N/A	
Provision 5.6-1	All unused network and logical interfaces shall be disabled.	M	Y	*Kconfig*
Provision 5.6-2	In the initialized state, the network interfaces of the device shall minimize the unauthenticated disclosure of security-relevant information.	M	Y	
Provision 5.6-3	Device hardware should not unnecessarily expose physical interfaces to attack.	R	Y	*Kconfig* and *Hardening Tool*
Provision 5.6-4	Where a debug interface is physically accessible, it shall be disabled in software.	M C	Y	*Hardening Tool*
Provision 5.6-5	The manufacturer should only enable software services that are used or required for the intended use or operation of the device.	R	Y	*Kconfig* and *Hardening Tool*
Provision 5.6-6	Code should be minimized to the functionality necessary for the service/device to operate.	R	Y	*Kconfig*
Provision 5.6-7	Software should run with least necessary privileges, taking account of both security and functionality.	R	Y	*Security Overview*
Provision 5.6-8	The device should include a hardware-level access control mechanism for memory.	R	Y	*Memory protection*

continues on next page

Table 6 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.6-9	The manufacturer should follow secure development processes for software deployed on the device.	R	Y	*Security Overview* and *Coding guidelines*
Provision 5.7-1	The consumer IoT device should verify its software using secure boot mechanisms.	R	Y	Functionality provided by MCUboot. Also see *Security Overview*.
Provision 5.7-2	If an unauthorized change is detected to the software, the device should alert the user and/or administrator to the issue and should not connect to wider networks than those necessary to perform the alerting function.	R	N	Zephyr does not provide runtime detection / notification.
Provision 5.8-1	The confidentiality of personal data transiting between a device and a service, especially associated services, should be protected, with best practice cryptography.	R	Y	
Provision 5.8-2	The confidentiality of sensitive personal data communicated between the device and associated services shall be protected, with cryptography appropriate to the properties of the technology and usage.	M	Y	
Provision 5.8-3	All external sensing capabilities of the device shall be documented in an accessible way that is clear and transparent for the user.	M	Y	*Sensing Subsystem*
Provision 5.9-1	Resilience should be built in to consumer IoT devices and services, taking into account the possibility of outages of data networks and power.	R	Y	
Provision 5.9-2	Consumer IoT devices should remain operating and locally functional in the case of a loss of network access and should recover cleanly in the case of restoration of a loss of power.	R	Y	
Provision 5.9-3	The consumer IoT device should connect to networks in an expected, operational and stable state and in an orderly fashion, taking the capability of the infrastructure into consideration.	R	Y	
Provision 5.10-1	If telemetry data is collected from consumer IoT devices and services, such as usage and measurement data, it should be examined for security anomalies.	R C	N/A	

continues on next page

Table 6 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 5.11-1	The user shall be provided with functionality such that user data can be erased from the device in a simple manner.	M	N/A	
Provision 5.11-2	The consumer should be provided with functionality on the device such that personal data can be removed from associated services in a simple manner.	R	N/A	
Provision 5.11-3	Users should be given clear instructions on how to delete their personal data.	R	N/A	
Provision 5.11-4	Users should be provided with clear confirmation that personal data has been deleted from services, devices and applications.	R	N/A	
Provision 5.12-1	Installation and maintenance of consumer IoT should involve minimal decisions by the user and should follow security best practice on usability.	R	N/A	
Provision 5.12-2	The manufacturer should provide users with guidance on how to securely set up their device.	R	N/A	
Provision 5.12-3	The manufacturer should provide users with guidance on how to check whether their device is securely set up.	R	N/A	
Provision 5.13-1	The consumer IoT device software shall validate data input via user interfaces or transferred via Application Programming Interfaces (APIs) or between networks in services and devices.	M	Y	*Syscall verification* and *Coding guidelines*
Provision 6.1-1	The manufacturer shall provide consumers with clear and transparent information about what personal data is processed, how it is being used, by whom, and for what purposes, for each device and service. This also applies to third parties that can be involved, including advertisers.	M	N/A	
Provision 6.1-2	Where personal data is processed on the basis of consumers' consent, this consent shall be obtained in a valid way.	M C	N/A	
Provision 6.1-3	Consumers who gave consent for the processing of their personal data shall have the capability to withdraw it at any time.	M	N/A	

continues on next page

Table 6 – continued from previous page

Provision	Description	Status	Support	Detail
Provision 6.1-4	If telemetry data is collected from consumer IoT devices and services, the processing of personal data should be kept to the minimum necessary for the intended functionality.	R C	N/A	
Provision 6.1-5	If telemetry data is collected from consumer IoT devices and services, consumers shall be provided with information on what telemetry data is collected, how it is being used, by whom, and for what purposes.	M C	N/A	

Chapter 10

Safety

These documents describe the processes, developer guidelines and requirements for ensuring safety is addressed within the Zephyr project.

10.1 Zephyr Safety Overview

10.1.1 Introduction

This document is the safety documentation providing an overview over the safety-relevant activities and what the Zephyr Project and the Zephyr Safety Working Group / Committee try to achieve.

This overview is provided for people who are interested in the functional safety development part of the Zephyr RTOS and project members who want to contribute to the safety aspects of the project.

10.1.2 Overview

In this section we give the reader an overview of what the general goal of the safety certification is, what standard we aim to achieve and what quality standards and processes need to be implemented to reach such a safety certification.

10.1.3 Safety Document update

This document is a living document and may evolve over time as new requirements, guidelines, or processes are introduced.

1. Changes will be submitted from the interested party(ies) via pull requests to the Zephyr documentation repository.
2. The Zephyr Safety Committee will review these changes and provide feedback or acceptance of the changes.
3. Once accepted, these changes will become part of the document.

10.1.4 General safety scope

The general scope of the Safety Committee is to achieve a certification for the IEC 61508 standard and the Safety Integrity Level (SIL) 3 / Systematic Capability (SC) 3 for a limited source scope (see certification scope TBD). Since the code base is pre-existing, we use the route 3s/1s approach defined by the IEC 61508 standard.

Route 3s
Assessment of non-compliant development. Which is basically the route 1s with existing sources.

Route 1s
Compliant development. Compliance with the requirements of this standard for the avoidance and control of systematic faults in software.

Summarization IEC 61508 standard

The IEC 61508 standard is a widely recognized international standard for functional safety of electrical, electronic, and programmable electronic safety-related systems. Here's an overview of some of the key safety aspects of the standard:

1. **Hazard and Risk Analysis**: The IEC 61508 standard requires a thorough analysis of potential hazards and risks associated with a system in order to determine the appropriate level of safety measures needed to reduce those risks to acceptable levels.
2. **Safety Integrity Level (SIL)**: The standard introduces the concept of Safety Integrity Level (SIL) to classify the level of risk reduction required for each safety function. The higher the SIL, the greater the level of risk reduction required.
3. **System Design**: The IEC 61508 standard requires a systematic approach to system design that includes the identification of safety requirements, the development of a safety plan, and the use of appropriate safety techniques and measures to ensure that the system meets the required SIL.
4. **Verification and Validation**: The standard requires rigorous testing and evaluation of the safety-related system to ensure that it meets the specified SIL and other safety requirements. This includes verification of the system design, validation of the system's functionality, and ongoing monitoring and maintenance of the system.
5. **Documentation and Traceability**: The IEC 61508 standard requires a comprehensive documentation process to ensure that all aspects of the safety-related system are fully documented and that there is full traceability from the safety requirements to the final system design and implementation.

Overall, the IEC 61508 standard provides a framework for the design, development, and implementation of safety-related systems that aims to reduce the risk of accidents and improve overall safety. By following the standard, organizations can ensure that their safety-related systems are designed and implemented to the highest level of safety integrity.

Why IEC 61508?

The IEC 61508 standard was selected because it serves as a foundational functional safety standard applicable across various industry sectors. It provides a robust framework that can be used as base for specific standards for different industries. This makes IEC 61508 particularly relevant for Zephyr, as the operating system's versatility allows it to be effectively utilized across a wide range of industry sectors.

The following diagram illustrates the relationship between the IEC 61508 standard and other related standards:

10.1.5 Quality

Quality is a mandatory expectation for software across the industry. The code base of the project must achieve various software quality goals in order to be considered an auditable code base from a safety perspective and to be usable for certification purposes. But software quality is not an additional requirement caused by functional safety standards. Functional safety considers quality as an existing pre-condition and therefore the "quality managed" status should be pursued for any project regardless of the functional safety goals. The following list describes the quality goals which need to be reached to achieve an auditable code base:

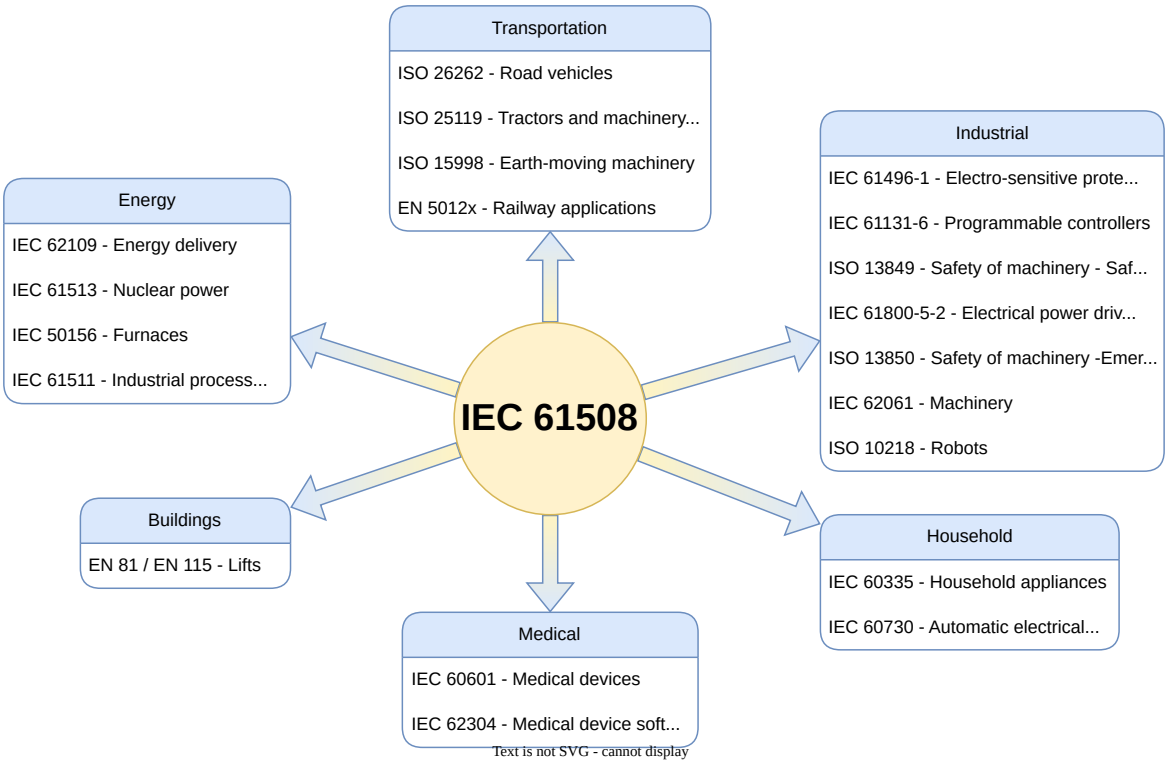


Fig. 1: IEC 61508 relation to other standards

1. Basic software quality standards
   a. *Coding Guidelines* (including: static code analysis, coding style, etc.)
   b. *Safety Requirements* and requirements tracing
   c. Test coverage
2. Software architecture design principles
   a. Layered architecture model
   b. Encapsulated components
   c. Encapsulated single functionality (if not fitable and manageable in safety)

Basic software quality standards - Safety view

In this chapter the Safety Committee describes why they need the above listed quality goals as pre-condition and what needs to be done to achieve an auditable code base from the safety perspective. Generally speaking, it can be said that all of these quality measures regarding safety are used to minimize the error rate during code development.

Coding Guidelines The coding guidelines are the basis to a common understanding and a unified ruleset and development style for industrial software products. For safety the coding guidelines are essential and have another purpose beside the fact of a unified ruleset. It is also necessary to prove that the developers follow a unified development style to prevent **systematic errors** in the process of developing software and thus to minimize the overall **error rate** of the complete software system.

Also the **IEC 61508 standard** sets a pre-condition and recommendation towards the use of coding standards / guidelines to reduce likelihood of errors.

The project TSC and the Safety Committee of the project agreed to implement a staged and incremental approach for complying with a set of coding rules (AKA Coding Guidelines) to improve quality and consistency of the code base. Below are the agreed upon stages:

Stage I (COMPLETED)
Coding guideline rules are available to be followed and referenced, but not enforced. Rules are not yet enforced in CI and pull-requests cannot be blocked by reviewers/approvers due to violations.

Stage II
Reviewers/approvers can block pull-requests due to violations of the coding guidelines in pull-requests across the codebase.

Begin enforcement on a limited scope of the code base. Initially, this would be the safety certification scope. For rules easily applied across codebase, we should not limit compliance to initial scope. This step requires tooling, CI setup and an enforcement strategy.

Stage III
Revisit the coding guideline rules and based on experience from previous stages, refine/iterate on selected rules.

Stage IV
Expand enforcement to the wider codebase. Exceptions may be granted on some areas of the codebase with a proper justification. Exception would require TSC approval.

> **Note**
>
> Coding guideline rules may be removed/changed at any time by filing a GH issue/RFC.

> **Important**
>
> **Current stage:** The prerequisites to complete **Stage II** are currently being looked at: The tooling is in evaluation, CI setup and enforcement strategy is being worked on.

Requirements and requirements tracing Requirements and requirement management are not only important for software development, but also very important in terms of safety. On the one hand, this specifies and describes in detail and on a technical level what the software should do, and on the other hand, it is an important and necessary tool to verify whether the described functionality is implemented as expected. For this purpose, tracing the requirements down to the code level is used. With the requirements management and tracing in hand, it can now be verified whether the functionality has been tested and implemented correctly, thus minimizing the systematic error rate.

Also the IEC 61508 standard highly recommends (which is like a must-have for the certification) requirements and requirements tracing.

Test coverage A high test coverage, in turn, is evidence of safety that the code conforms precisely to what it was developed for and does not execute any unforeseen instructions. If the entire code is tested and has a high (ideally 100%) test coverage, it has the additional advantage of quickly detecting faulty changes and further minimizing the error rate. However, it must be noted that different requirements apply to safety for test coverage, and various metrics must be considered, which are prescribed by the IEC 61508 standard for the SIL 3 / SC3 target. The following must be fulfilled, among other things:

- Structural test coverage (entry points) 100%
- Structural test coverage (statements) 100%
- Structural test coverage (branches) 100%

If the 100% cannot be reached (e.g. statement coverage of defensive code) that part needs to be described and justified in the documentation.

Software architecture design principles

To create and maintain a structured software product it is also necessary to consider individual software architecture designs and implement them in accordance with safety standards because some designs and implementations are not reasonable in safety, so that the overall software and code base can be used as auditable code. However, most of these software architecture designs have already been implemented in the Zephyr project and need to be verified by the Safety Committee / Safety Working Group and the safety architect.

Layered architecture model The **IEC 61508 standard** strongly recommends a modular approach to software architecture. This approach has been pursued in the Zephyr project from the beginning with its layered architecture. The idea behind this architecture is to organize modules or components with similar functionality into layers. As a result, each layer can be assigned a specific role in the system. This model has the advantage in safety that interfaces between different components and layers can be shown at a very high level, and thus it can be determined which functionalities are safety-relevant and can be limited. Furthermore, various analyses and documentations can be built on top of this architecture, which are important for certification and the responsible certification body.

Encapsulated components Encapsulated components are an essential part of the architecture design for safety at this point. The most important aspect is the separation of safety-relevant components from non-safety-relevant components, including their associated interfaces. This ensures that the components have no **repercussions** on other components.

Encapsulated single functionality (if not reasonable and manageable in safety) Another requirement for the overall system and software environment is that individual functionalities can be disabled within components. This is because if a function is absolutely unacceptable for safety (e.g. complete dynamic memory management), then these individual functionalities should be able to be turned off. The Zephyr Project already offers such a possibility through the use of Kconfig and its flexible configurability.

10.1.6 Processes and workflow

The diagram describes the rough process defined by the Safety Committee to ensure safety in the development of the Zephyr project. To ensure understanding, a few points need to be highlighted and some details explained regarding the role of the safety architect and the role of the safety committee in the whole process. The diagram only describes the paths that are possible when a change is related to safety.

1. On the main branch, the safety scope of the project should be identified, which typically represents a small subset of the entire code base. This subset should then be made auditable during normal development on “main”, which means that special attention is paid to quality goals (*Quality*) and safety processes within this scope. The Safety Architect works alongside the Technical Steering Committee (TSC) in this area, monitoring the development process to ensure that the architecture meets the safety requirements.
2. At this point, the safety architect plays an increasingly important role. For PRs/issues that fall within the safety scope, the safety architect should ideally be involved in the discussions and decisions of minor changes in the safety scope to be able to react to safety-relevant changes that are not conformant. If a pull request or issue introduces a significant and influential change or improvement that requires extended discussion or decision-making, the safety architect should bring it to the attention of the Safety Committee or the Technical Steering Committee (TSC) as appropriate, so that they can make a decision on the best course of action.

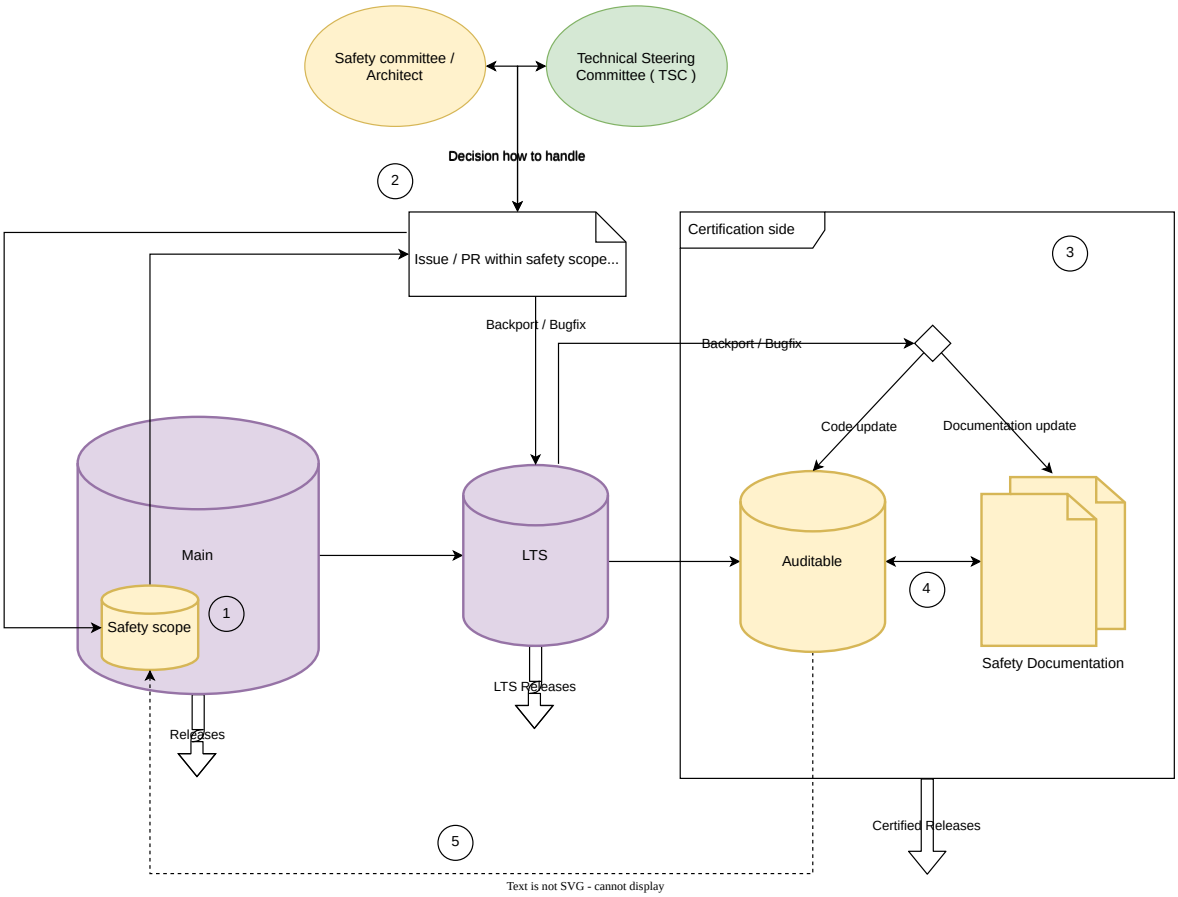


Fig. 2: Safety process and workflow overview

3. This section describes the certification side. At this point, the code base has to be in an "auditable" state, and ideally no further changes should be necessary or made to the code base. There is still a path from the main branch to this area. This is needed in case a serious bug or important change is found or implemented on the main branch in the safety scope, after the LTS and the auditable branch were created. In this case, the Safety Committee, together with the safety architect, must decide whether this bug fix or change should be integrated into the LTS so that the bug fix or change could also be integrated into the auditable branch. This integration can take three forms: First either as only a code change or second as only an update to the safety documentation or third as both.
4. This describes the necessary safety process required for certification itself. Here, the final analyses, tests, and documents are created and conducted which must be created and conducted during the certification, and which are prescribed by the certifying authority and the standard being certified. If the certification body approves everything at this stage and the safety process is completed, a safety release can be created and published.
5. This transition from the auditable branch to the main branch should only occur in exceptional circumstances, specifically when something has been identified during the certification process that needs to be quickly adapted on the "auditable" branch in order to obtain certification. In order to prevent this issue from arising again during the next certification, there needs to be a path to merge these changes back into the main branch so that they are not lost, and to have them ready for the next certification if necessary.

Important

Safety should not block the project and minimize the room to grow in any way.

Important

TODO: Find and define ways, guidelines and processes which minimally impact the daily work of the maintainers, reviewers and contributors and also the safety architect itself. But which are also suitable for safety.

10.2 Safety Requirements Management

These documents describe the processes, developer guidelines and requirements for ensuring the creation, verification and configuration management of requirements follows a suitable approach.

10.2.1 Safety Requirements

Introduction

The safety committee leads the effort to gather requirements that reflect the **actual** state of the implementation, following the *route 3s* approach of the project's safety effort. The goal is **NOT** to create new requirements to request additional features for the project.

The requirements are gathered in the separate repository: Requirement repository

The current rendered version of the Requirement Repository's content can be found at Zephyr Project Requirements.

Objectives of Requirements Management in the Zephyr Project

In the development and for documentation of implemented or expected functionality, effective Requirements Management is essential to ensure that the Zephyr RTOS meets its intended pur-

pose, performs reliably under its constraints, and can be integrated easily both with hardware platforms, within an embedded software stack and software application layers.

Requirements Management provides a structured approach to capturing, organizing, and tracing the needs and expectations of stakeholders through a system's lifecycle. The documentation provided in the project's docs already emphasizes implementation details, from kernel functionality, to API configuration options and specific module behavior. While this is essential for developers, relying solely on low-level documentation can lead to fragmented understanding, limited traceability, and difficulty scaling or certifying a final system using Zephyr RTOS.

To enhance this understanding and add value for non-coding roles like product owners, software architects, quality management, safety management and assessors, the Zephyr Project needs to have higher level requirements that describe overall functionality, structure and implementation constraints.

Guideline to Requirements Management

Below are the guidelines for the requirements repository and the expectations regarding suitable requirement structure and syntax when adding requirements to the repository.

Note: these are the guidelines to create requirements in the requirements repository, which is not written using .rst like the docs published to the Zephyr users. To learn more about the guidelines to create the docs, please refer to *Documentation Guidelines*

Scope The initial scope of the requirements covers the KERNEL functionalities. Requirements for non-kernel functionality are also welcome, but currently not the focus.

Managing Zephyr Requirements: Structure, Tools, and Grammar

Levels of requirements in the repository The safety working group has evaluated a suitable approach to enhance the existing low level descriptions with requirements. For this a two level structure of requirements has been created on top of the existing documentation structure, System Requirements at the highest level to define overarching capabilities of the Zephyr RTOS as a whole, and Software Requirements to break down the System Requirements' expectations into detailed specifications for individual components within the RTOS.

System Requirements

System requirements describe the behavior of the Zephyr RTOS (= the system here). They describe the functionality and constraints of the Zephyr RTOS from a high-level perspective, without going into details of the functionality itself.

The purpose of the system requirements is to get an overview of the currently implemented features of the Zephyr RTOS.

These requirements articulate what the RTOS must achieve from a functional, performance, and interface perspective — such as deterministic task scheduling, interrupt handling latency, memory footprint limits, and compliance with industry standards. System requirements serve as the foundation for design decisions and validation criteria.

To create these requirements it is beneficial if a person writing these requirements already has some knowledge of the Zephyr RTOS Project and/or requirements and constraints that are specific to an RTOS.

Software Requirements

Beneath the System Requirements layer, Software requirements refine the system-level requirements to a more granular level. These component level requirements break down the system-level expectations into detailed specifications for individual components within the RTOS. These include modules such as the scheduler, memory manager and inter-process communication mechanisms.

Software requirements describe the behavior, interfaces, and constraints of each component, ensuring that they collectively fulfill the system-level goals. These requirements define the specific actions the feature shall be able to execute and the behavior of the feature so that each requirement can be verified by tests, analysis and/or inspection.

This level of requirements still has to be implementation free in its definitions. Implementation definitions should be in the docs, if not already there, they need to be added as another task of this requirements creation effort.

Requirements Management Toolchain

Requirement Repository: The Requirement repository represents Zephyr's structured approach to requirements management, currently focusing on the creation of requirements within the scope of its targeted safety certification, but not limited to this scope.

To work with this repository, follow the normal GitHub workflow of branching and pull requests. The pull requests on this repository are currently reviewed by members of the safety working group and need approval at least from one of the following roles to get merged: Zephyr Safety Manager, Zephyr Safety Chair or Zephyr Safety Architect.

Requirement Tooling: For authoring, linking and rendering a browsable .html version of the Zephyr Project's requirements we are using the tool StrictDoc

StrictDoc is a lightweight, open-source tool for writing, browsing, and exporting structured requirements. It supports hierarchical requirements, traceability, and HTML export for easy review.

Requirements are written in StrictDoc's own markdown syntax.

For those that prefer editing in a WYSIWYG fashion, the .html exports of StrictDoc can also be edited using StrictDoc's local server.

To set up your Toolchain, jump to the *Getting started with Requirements Management* section.

Verification of Requirements: New and changed requirements (what is in the PRs to the Requirements Repository) need to be verified by a review. The review comments can be captured directly in the comments during the PR review. A requirements review has to consider both technical correctness and the formal need that requirements need be created following certain criteria.

More details regarding the checklist and the formal expectations can be found in the *Safety Requirements Checklist*. In the tests of the Zephyr Project adherence of the implementation to the functionality described in the requirements must be evaluated.

Glossary of Requirements Grammar Elements All requirements must use the defined grammar:

Table 1: Structure for requirements captured in StrictDoc

Property	Description
UID	A unique ID that enables identification of the requirement. Is assigned automatically by StrictDoc.
STATUS	Describing the status of the requirement. Draft, Approved, Retired
TYPE	Functional, Non-functional
COMPONENT	the component the requirement shall be implemented in
REFERENCE	Reference to parent and/or child requirement
TITLE	A descriptive title of the requirement. Use short and concise requirement titles.
STATEMENT	Description of the requirement
USER_STORY	A user story describing the requirement (optional, only add if it adds additional value to the requirements statement)

Requirement UID (Unique identifier) Handling The tool used to manage requirements, StrictDoc, is responsible for handling the Unique Identifier (UID) associated with each requirement. To manage UIDs, follow these steps:

1. Don't add a requirement UID and UID field for new requirements
2. After completing work on the new requirements execute: `strictDoc manage auto-uid .`
3. Establish links between the requirements with the new attributed UIDs if needed

After doing this, the requirements are ready and a pull request can be created. The CI in the PR will check if the requirements UIDs are valid or if there are duplicates in it. If there are duplicates in the PR, these need to be resolved by rebasing and re-executing the steps above.

Consistency Consistency regarding language and choice of words shall be maintained across all requirements. (See: *Syntax*)

It also is recommended to take a look at the already existing requirements in the Requirement repository as well as at discussions going on in the PRs to this repository to get a feeling what makes sense in this context.

Pull Request requirement repository

- Adhere to the *Contribution Guidelines* of the Zephyr Project.
  - As long as they are applicable to the requirements repository.
- Avoid creating large commits that contain both trivial and non-trivial changes.
- Avoid moving and changing requirements in the same commit.

Characteristics of a good requirement

- Unambiguous
- Verifiable (e.g. testable for functional requirements)
- Clear (concise, brief, simple, precise)
- Correct
- Understandable
- Feasible (realistic, possible)
- Independent
- Atomic

- Necessary
- Implementation-free (abstract)

Characteristics of a set of requirements

- Complete
- Consistent
- Non redundant

Syntax

- Use of a recognized Requirements Syntax is recommended.
  - EARS is a good reference. Particularly if you are unfamiliar with requirements writing.
  - Other formats are accepted as long as the characteristics of a requirement from above are met.

10.2.2 Safety Requirements Checklist

Introduction

The requirements that are created, modified or updated need to be suitable. This means they need to be unambiguous, uniquely identifiable, feasible, testable etc. To ensure this, each requirement needs to be reviewed in its pull request, if it fulfills the basic requirements to be a requirement. This checklist shall serve as the minimum that needs to be considered when doing a review of a requirement PR.

Requirements Structure and Guidelines

You can find the approach taken by the Zephyr Project in *Safety Requirements*

Review Instructions

This checklist lists the topics and questions that shall be considered when approving requirements during a review. The review outcome must be documented in the PR review, in a way that it is clear that this checklist has been used and the actual outcome of the review regarding open questions, findings or approval. This checklist shall be applied at least once for new or changed requirements within the scope of a safety assessment before the safety scope is released.

The requirements are gathered in the separate repository: Requirement repository

The review can be done on two levels - regarding a single requirement or as a group of requirements. The checkpoints are as following:

Review checkpoints for each and every requirement

Each new or changed requirement needs to be analysed, to make sure it is suitable and to identify if it is affecting the safety of Zephyr in any kind.

Table 2: Checkpoints for requirements review

ID	Question	Explanation	Example
Ver-Req_	Is this requirement clearly comprehensible and unambiguous?	To make sure the requirement can be analysed it must be easy to understand and leave no room for (functional) interpretation.	Avoid vague statements such as **some, sufficient, typical, many, several, few, ...**. Statements like **The Zephyr Kernel should provide functionality xyz where possible** are not accurate enough and leave room for interpretation. Also **should** must be replaced with the unambiguous **shall**. The **where possible** must be replaced with a definition where the author of the requirement wants the functionality to be provided. This can be a list of cases, system states or other conditions. As long as they are clearly defined.
Ver-Req_	Is this requirement atomic?	The requirement really must only describe one thing that needs to be fulfilled. Otherwise things like safety analysis, unique linking to architectural components and implementation and verifiability can be impaired.	Statements like **The signal a shall switch on and off the yellow LED and the error state shall be indicated by switching on the red LED** are not atomic. This needs to go into two requirements - **REQ1: The signal a shall switch on and off the yellow LED.** and **REQ2: The error state shall be indicated by switching on the red LED.** This also can be supplemented by a **REQ3(variant a): Yellow LED and the red Error LED must be separate** or **REQ3(variant b): Yellow LED and red Error LED can be combined in case of TBD**.
Ver-Req_	Is the requirement internally consistent?	The requirement itself must be fully consistent within itself.	Statements like **The system shall use a preemtpive RTOS to guarantee deterministic task scheduling. The system also need to ensure that all tasks, including low-priority background logging tasks, complete without preemption to preserve data integrity.** are contradicting. Stuff like this must be avoided. If both options shall be available, you must define the conditions for these options and how to configure the options.
Ver-Req_	Is the requirement feasible, achievable and verifiable?	Requirements are not a fantasy wish-list, they need to be able to reach with reasonable matter (although what is reasonable can differ between projects for sure). So generally this means something that sounds good on paper but is technically impossible, internally contradictory, or violates fundamental constraints of the sys-	A bad example is e.g. **The RTOS shall guarantee zero-latency interrupt handling for all devices under all conditions.**, which is physically impossible, as no hardware can respond in literally zero time. Better would be something more concrete like **The RTOS shall guarantee an interrupt latency of 5 µs on the target HW architecture at 80 MHz under maximum system load.**

Review checkpoints looking at a group of requirements

Table 3: Checkpoints for requirements review

ID	Question	Explanation	Example
Ver-Req_	Are the requirements appropriately grouped and structured?	Usually a suitable requirements structure follows a hierarchical order and within the hierarchical layers the requirements are grouped e.g. regarding their functionality, criticality, parent requirement's source etc.	An unsuitably grouped requirement is e.g. in the wrong section, or in a wrong hierarchy level (e.g. in systems requirements although the requirement defines a specific property of a kernel component and therefore has to be on software component level), sections also might be overloaded and need more structuring in sub-sections.
Ver-Req_	Is each requirement externally consistent?	The requirements must not conflict or contradict each other.	Consider having two requirements: **Requirement A: The RTOS shall support a maximum of 10 concurrent tasks. Requirement B: The RTOS shall support up to 20 concurrent tasks for high-performance applications.** - These two requirements clearly contradict each other, but might both be correct for the persons writing each one of them. Usually having requirements like this needs communication between the authors of the requirements, and a possible solution might be the following: **The RTOS shall support up to 20 concurrent tasks, with a default configuration of 10 tasks for standard applications.**
Ver-Req_	Is each requirement suitably traceable?	Each requirement needs to have a clear source (where does it come from, why is it here...), associated verification and/or acceptance criteria and descending requirements or architectural elements.	A requirement always has to have a parent requirement linked, except for the highest level requirements, but still there needs to be an explanation in the Requirement rationale, note or comment.
Ver-Req_	Is the requirement unique?	Requirements must not be duplicated. If you need a requirement to clarify things in multiple locations, make sure one is the "main requirement" and the others are just linked information.	This is self explanatory, two same or similar requirements must be avoided.

10.2.3 Getting Started with Requirements

These documents contain some guidance for starting to work with requirements in the Zephyr Project.

Getting started with Requirements Management

Zephyr's requirements management is powered by StrictDoc, a lightweight tool for writing, organizing, and publishing structured requirements. This section explains how to set up and use the toolchain provided in the reqmgmt repository.

Overview The `reqmgmt` repository is Zephyr's official requirements workspace. It supports:

- Authoring system and software requirements
- Linking requirements to verification artifacts
- Generating browsable HTML documentation
- Running a local web interface for editing and review

Installation StrictDoc requires Python 3.7 or newer. To install (you might need to use pip3 instead of pip):

```
pip install strictdoc
```

Clone the Zephyr requirements repository:

```
git clone https://github.com/zephyrproject-rtos/reqmgmt
cd reqmgmt
```

Usage To export the requirements to HTML:

```
strictdoc export .
```

This generates a static site in the `output/` directory.

To launch the local web interface:

```
strictdoc server .
```

You should see something like

```
Uvicorn running on http://127.0.0.1:5111 (Press CTRL+C to quit)
```

Bring up your local browser and paste in the address above to access the browser-based editor for navigating and editing requirements.

Structure The repository contains:

- `docs/` — Requirements documents organized by subsystem
- `strictdoc.toml` — Configuration file for StrictDoc
- `tasks.py` — Automation helpers for building and validating requirements

Live Documentation You can view the published requirements online at:

https://zephyrproject-rtos.github.io/reqmgmt

Next Steps

- Explore the `docs/` folder to see example requirement files
- Use the web interface to create or edit your own
- Link requirements to test cases or source modules for traceability

`StrictDoc` helps Zephyr move toward safety-critical readiness by formalizing the connection between system goals and software implementation.

Create your first Zephyr RTOS requirement

Repository Overview The Zephyr requirements are managed in the `reqmgmt` repository using StrictDoc. The repo includes:

- `docs/`: Contains requirement documents in StrictDoc format
- `tools/`: Utility scripts and configuration
- `strictdoc.toml`: Project configuration for StrictDoc
- `tasks.py`: Task automation using Invoke

Step 1: Create or Edit a Requirement File Navigate to the `docs/` folder in the repository.

Folder Overview The `docs/` folder contains all requirement documents written in StrictDoc format.

The `docs/` folder contains two subfolders. These are organized as follows:

- `system_requirements/` Contains high-level system requirements that describe the overall goals, constraints, and expected behavior of the Zephyr RTOS.
  - `system_requirements.sgra` Contains the GRAMMAR, that defines the formal structure of a requirement
  - `index.sdoc` that contains the actual requirements statements
- `software_requirements/` Contains component-level requirements. Each file in this folder corresponds to a specific subsystem or module.
  - `software_requirements.sgra` contains the GRAMMAR for all software requirements
  - a list of `.sdoc` snippet files that compile together to a big software requirements document, e.g.
    * `interrupts.sdoc`
    * `queues.sdoc`
    * `semaphore.sdoc`
    * `threads.sdoc`

These files are modular and allow contributors to work independently on specific areas of the system. You can either edit an existing `.sdoc` file or create a new one. Here's a basic example of a requirement block:

```
[REQUIREMENT]
UID: ZEP-SRS-17-1
STATUS: Draft
TYPE: Functional
COMPONENT: File System
TITLE: Create file
STATEMENT: >>>
```

(continues on next page)

(continued from previous page)

```
Zephyr shall provide file create capabilities for files on the file system.
<<<
```

Set the UID to `TBD` so StrictDoc can generate it automatically in a later step.

Step 2: Save the File Save your file in the systems or software requirements subfolder in `docs/` folder. If you create a new file give it a meaningful name, usually referring to the component its content is meant for or the functionality.

Step 3: Assign a UID Automatically From the root of the repository, run the following command:

```
strictdoc manage auto-uid .
```

This command will:

- Traverse the project directory
- Find requirements with `UID: TBD`
- Assign a unique identifier to each one

Step 4: Verify the UID Assignment Open the file again. You should now see something like:

```
UID: ZEP-SRS-17-2
STATUS: Draft
TYPE: Functional
COMPONENT: File System
TITLE: Create file
STATEMENT: >>>
Zephyr shall provide file create capabilities for files on the file system.
<<<
```

Note: The UID format can be configured in `strictdoc.toml`.

Optional: Generate HTML Documentation To build the HTML version of the requirements documentation:

```
strictdoc export .
```

This will generate browsable documentation in the `output/` folder.

Optional: Launch the Web Editor To interactively browse and edit requirements:

```
strictdoc server .
```

This launches a local web interface for editing and reviewing requirements.

Summary

- Create or edit a requirement in `docs/` with `UID: TBD`
- Run `strictdoc manage auto-uid .` to assign UIDs
- Optionally generate HTML docs or launch the web editor

StrictDoc helps Zephyr maintain structured, traceable, and editable requirements across platforms.

10.2.4 Requirements Catalog

The Zephyr requirements are maintained in the dedicated `reqmgmt` repository using StrictDoc. When that repository is present in the workspace (it is pulled in as a west project), its requirements are exported and rendered directly into this documentation below.

Requirements Catalog

The following requirements are imported from the Zephyr requirements repository (`reqmgmt`).

Zephyr Software Requirements

ZEP-SRS-19-1: Atomic Operations The Zephyr RTOS shall provide an interface to access memory while ensuring mutual exclusion.

Status
Draft

Type
Functional

Component
Hardware Architecture Interface

User Story
As a Zephyr RTOS user I want to read from or write into a memory areas without being disturbed by other threads or ISRs.

Parents
ZEP-SYRS-1

ZEP-SRS-19-2: Thread Context Switching The Zephyr RTOS shall provide a mechanism for context switching between threads.

Status
Draft

Type
Functional

Component
Hardware Architecture Interface

User Story
As a Zephyr RTOS user I want to execute code concurrently in one or more threads and when interrupted at a code location in a thread, to continue at the very same location.

Parents
ZEP-SYRS-1

ZEP-SRS-19-3: Software Exceptions The Zephyr RTOS shall provide an interface to implement software exceptions.

Status
Draft

Type
Functional

Component
Hardware Architecture Interface

User Story
As a Zephyr RTOS user I want to catch any software exception and handle it according to my application needs.

Parents
ZEP-SYRS-1

ZEP-SRS-19-4: Processor Mode Support The Zephyr RTOS shall provide an interface for managing processor modes.

Status
Draft

Type
Functional

Component
Hardware Architecture Interface

User Story
If MCU power state was meant here: As a Zephyr RTOS user I want to control the MCU's power saving mode such e.g. operation, sleep, deep sleep or similar as supported by the selected MCU.

Parents
ZEP-SYRS-1

ZEP-SRS-18-1: Formatted output The Zephyr RTOS shall support formatted output.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to printf with various output formats.

Parents
ZEP-SYRS-3

ZEP-SRS-18-2: Floating Point Math Support The Zephyr RTOS shall support floating point math libraries for processors where floating point is available.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to calculate with floating point numbers.

Parents
ZEP-SYRS-3

ZEP-SRS-18-3: Boolean Primitives Support The Zephyr RTOS shall support boolean primitives.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to work with boolean values and logic.

Parents
ZEP-SYRS-3

ZEP-SRS-18-4: Standard Unix time interface The Zephyr RTOS shall support the standard UNIX time interface.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr User, I want to be able to use system time.

Parents
ZEP-SYRS-3

ZEP-SRS-18-5: Strings support The Zephyr RTOS shall support an interface to manage strings.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to manipulate text strings.

Parents
ZEP-SYRS-3

ZEP-SRS-18-6: Moving/copying regions of memory The Zephyr RTOS shall support an interface to move contents between regions of memory.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to copy the memory contents to different addresses.

Parents
ZEP-SYRS-3

ZEP-SRS-18-7: I/O based interface The Zephyr RTOS shall support a file I/O based interface for driver communication.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to use File I/O functions.

Parents
ZEP-SYRS-3

ZEP-SRS-18-8: C99 integer types The Zephyr RTOS shall support standard C99 integer types.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to calculate with C99 Integers.

Parents
ZEP-SYRS-3

ZEP-SRS-18-9: Standard System Error Numbers (IEEE Std 1003.1-2017) The Zephyr RTOS shall support standard system error numbers as defined by IEEE Std 1003.1-2017.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user, I want to be able to use IEE Std 1003.1-2017 Error Numbers for Interoperability.

Parents
ZEP-SYRS-3

ZEP-SRS-18-10: Document set of Zephyr OS required C library functions in Safety Manual The set of C Library functions required by Zephyr shall be documented in the Zephyr Safety Manual.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr user, I want to know which C library functions are being called by the Zephyr OS safety scope code.

Parents
ZEP-SYRS-3

ZEP-SRS-18-11: Support external C libraries documentation in Zephyr Safety Manual The Zephyr Safety Manual shall specify how to configure the support of external C Libraries.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr User, I need to understand how to configure the external C libraries to align with the validation that has been performed.

Parents
ZEP-SYRS-3

ZEP-SRS-14-1: Device Driver Abstraction The Zephyr RTOS shall provide abstraction of device drivers with common functionalities as an intermediate interface between applications and device drivers, where such interface is implemented by individual device drivers. Proposal for replacement: Zephyr shall provide an interface between application and individual device drivers to provide an abstraction of device drivers with common functionalities.

Status
Draft

Type
Functional

Component
Device Driver API

Parents
ZEP-SYRS-4

ZEP-SRS-14-2: Expose kernel to hardware interrupts The Zephyr RTOS shall provide an interface for managing a defined set of hardware exceptions (including interrupts) across all systems.

Status
Draft

Type
Functional

Component
Device Driver API

User Story
As a Zephyr RTOS user I want hardware exceptions (hardware failures, programming mistakes) to be handled gracefully (no program crashes as far as possible).

Parents
ZEP-SYRS-4

ZEP-SRS-16-1: Fatal Exception Error Handler The Zephyr RTOS shall provide default handlers for exceptions.

Status
Draft

Type
Functional

Component
Exception and Error Handling

User Story
As a Zephyr RTOS user I want exceptions which I did not handle explicitly (by mistake or on purpose) to be caught by a default handler, defined either by Zephyr OS or by myself.

Parents
ZEP-SYRS-5

ZEP-SRS-16-2: Default handler for fatal errors The Zephyr RTOS shall provide default handlers for fatal errors that do not have a dedicated handler.

Status
Draft

Type
Functional

Component
Exception and Error Handling

Parents
ZEP-SYRS-5

ZEP-SRS-16-3: Assigning a specific handler The Zephyr RTOS shall provide an interface to assign a specific handler for an exception.

Status
Draft

Type
Functional

Component
Exception and Error Handling

Parents
ZEP-SYRS-5

ZEP-SRS-16-4: Assigning a specific handler for a fatal error The Zephyr RTOS shall provide an interface to assign a specific handler for a fatal error.

Status
Draft

Type
Functional

Component
Exception and Error Handling

Parents
ZEP-SYRS-5

ZEP-SRS-17-1: Create file Zephyr shall provide file create capabilities for files on the file system.

Status
Draft

Type
Functional

Component
File System

User Story
As a Zephyr OS user I want to be able to create a new file or overwrite an existing file at the same filesystem location / identifier (e.g. path + name).

Parents
ZEP-SYRS-6

ZEP-SRS-17-2: Open files Zephyr shall provide file open capabilities for files on the file system.

Status
Draft

Type
Functional

Component
File System

User Story
As a Zephyr OS user I want to be able to open a file for writing or reading. When opened for writing, I want to have exclusive access to the file.

Parents
ZEP-SYRS-6

ZEP-SRS-17-3: Read files Zephyr shall provide read access to files in the file system.

Status
Draft

Type
Functional

Component
File System

User Story
As a Zephyr OS user I want to be able to read from an existing file, also while the file is read from multiple and write accessed from one other instances.

Parents
ZEP-SYRS-6

ZEP-SRS-17-4: Write to files Zephyr shall provide write access to the files in the file system.

Status
Draft

Type
Functional

Component
File System

User Story
As a Zephyr OS user I want to be able to write to a file either from the beginning of the file or appending at the end.

Parents
ZEP-SYRS-6

ZEP-SRS-17-5: Close file Zephyr shall provide file close capabilities for files on the file system.

Status
Draft

Type
Functional

Component
File System

User Story
As a Zephyr OS user I want to be able to close a file after being finished with my file operations, unlocking any access restrictions.

Parents
ZEP-SYRS-6

ZEP-SRS-17-6: Move file Zephyr shall provide the capability to move files on the file system.

Status
Draft

Type
Functional

Component
File System

Parents
ZEP-SYRS-6

ZEP-SRS-17-7: Delete file Zephyr shall provide file delete capabilities for files on the file system.

Status
Draft

Type
Functional

Component
File System

User Story
As a Zephyr OS user I want to be able to delete an existing file.

Parents
ZEP-SYRS-6

ZEP-SRS-7-1: Installing static IRQ service routines (ISR). Zephyr RTOS shall provide a mechanism to initialize a static IRQ service routine (ISR), providing all parameters needed to configure the hardware and software.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-2: Static IRQ initial status. The static IRQ shall be initially disabled.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-3: Installing direct IRQ service routines (ISR). Zephyr RTOS shall provide a mechanism to initialize a direct IRQ handler, providing all parameters needed to configure the hardware and software.

Status
Draft

Type
Functional

Component
Interrupts

User Story
As the developer of low-power and low-latency applications, I need to implement ISRs that avoid the normal interrupt and power management overhead.

Parents
ZEP-SYRS-7

ZEP-SRS-7-4: Direct IRQ initial status. The direct IRQ shall be initially disabled.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-5: Installing dynamic IRQ service routines (ISR). Zephyr RTOS shall provide a mechanism to initialize a dynamic IRQ service routine, providing all parameters needed to configure the hardware and software.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-6: Dynamic IRQ initial status. The dynamic IRQ shall be initially disabled.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-7: Uninstalling dynamic IRQ service routines (ISR). Zephyr RTOS shall provide a mechanism to uninstall a dynamic ISR.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-8: Global IRQ disable Zephyr RTOS shall provide a mechanism to disable all IRQs on a CPU and return the state of the IRQ hardware prior to being disabled.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-9: Global IRQ enable Zephyr RTOS shall provide a mechanism to enable all IRQs on a CPU and return them to their previous state.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-10: Specific IRQ disable Zephyr RTOS shall provide a mechanism to disable a specified IRQ.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-11: Specific IRQ enable Zephyr RTOS shall provide a mechanism to enable a specified IRQ.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-12: IRQ Enabled status Zephyr RTOS shall provide a mechanism that returns the enabled status of a specified IRQ, where the status is enabled or disabled.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-13: ISR Context status Zephyr RTOS shall provide a mechanism that returns the execution context, where the context is In-ISR or Not In-ISR.

Status
Draft

Type
Functional

Component
Interrupts

User Story
As the developer of functions that may run in either ISR or THREAD context, I need to know the current context to enable conditional behavior.

Parents
ZEP-SYRS-7

ZEP-SRS-7-14: Multi-level interrupts The Zephyr RTOS shall support multi-level preemptive interrupt priorities, when supported by hardware.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-15: Interrupt service routine invocation When an enabled IRQ that has a connected interrupt service routine is asserted, the Zephyr RTOS shall invoke the associated interrupt service routine.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-7-16: Interrupt service routine parameter When the Zephyr RTOS invokes a static or dynamic interrupt service routine, it shall pass to the routine the parameter that was registered with it.

Status
Draft

Type
Functional

Component
Interrupts

Parents
ZEP-SYRS-7

ZEP-SRS-11-1: Dedicated Logging Thread Support The Zephyr RTOS shall support isolation of logging from other functionality.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want to be able to configure logging of events so the execution of logging activities does have no or only a minimal impact to the timing behaviour of my application.

Parents
ZEP-SYRS-8

ZEP-SRS-11-2: Logs available for post processing The Zephyr RTOS logging shall produce logs that are capable of being post processed.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want the logging information to be stored in a format which allows to be read possibly converted or displayed by COTS tools.

Parents
ZEP-SYRS-8

ZEP-SRS-11-3: Formatting log messages The Zephyr RTOS logging shall support formatting of log messages to enable filtering.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want to be able my application to format texts (printf alike) into the log message.

Parents
ZEP-SYRS-8

ZEP-SRS-11-4: Logging Filtering Support The Zephyr RTOS logging system shall support filtering based on severity level.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want to be able to distinguish between different severity level for my log messages (e.g. DEBUG, INFO, WARN, ERROR, PANIC).

Parents
ZEP-SYRS-8

ZEP-SRS-11-5: Multiple Backend Logging Support The Zephyr RTOS shall support logging messages to multiple system resources.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want to be able to simultaneously log to different channels which may store / redirect the information on / to different hardware (EEPROM, Flash, FRAM, UART, Ethernet, USB etc.).

Parents
ZEP-SYRS-8

ZEP-SRS-11-6: Deferred Logging Support The Zephyr RTOS shall support deferred logging.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want a minimal influence of logging activities to the timing behaviour of my application. Time consuming logging threads shall be done in the background.

Parents
ZEP-SYRS-8

ZEP-SRS-8-1: Memory Protection The Zephyr RTOS shall support memory protection features to isolate a thread's memory region.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want memory to be allocated and protected to my application threads preventing mistakenly access to foreign memory as far as the hardware allows.

Parents
ZEP-SYRS-9

ZEP-SRS-8-2: Granting access to kernel objects The Zephyr RTOS shall provide a mechanism to grant user threads access to kernel objects.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want, from the user space, under certain conditions, access to kernel objects.

Parents
ZEP-SYRS-9

ZEP-SRS-8-3: Separation between user and kernel threads for memory access The Zephyr RTOS shall be able to differentiate between user threads and kernel threads for memory access.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want, from the kernel space, unconditioned access to kernel objects.

Parents
ZEP-SYRS-9

ZEP-SRS-8-4: Safely handle unimplemented calls or invalid system calls The Zephyr RTOS shall have a defined behaviour when an invocation of an unimplemented system call is made.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want Zephyr OS to indicate any unimplemented system call by an appropriate error message.

Parents
ZEP-SYRS-9

ZEP-SRS-8-5: Response to invalid system call IDs The Zephyr RTOS shall have a defined behaviour when an invalid system call ID is used.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want Zephyr OS to indicate invalid system call by an appropriate error message.

Parents
ZEP-SYRS-9

ZEP-SRS-8-6: Prevent user threads creating higher priority threads The Zephyr RTOS shall prevent user threads from creating new threads that are higher priority than the caller.

Status
Draft

Type
Functional

Component
Memory Protection

Parents
ZEP-SYRS-9

ZEP-SRS-8-7: Revoking threads permissions on a kernel object The Zephyr RTOS shall support revoking permission to a kernel object. User mode threads may only revoke their own access to an object.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to be protected against other user threads changing access to kernel objects of my thread.

Parents
ZEP-SYRS-9

ZEP-SRS-8-8: Prevent user threads creating supervisor threads The Zephyr RTOS shall prevent user threads from creating kernel threads.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to be protected against user threads creating higher privileged kernel/supervisor threads.

Parents
ZEP-SYRS-9

ZEP-SRS-8-9: Reduced Privilege Level Threads The Zephyr RTOS shall allow the creation of threads that run in reduced privilege level.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to be able to create lower privileged threads than my own.

Parents
ZEP-SYRS-9

ZEP-SRS-8-10: User Mode Threads Performing Privileged Operations The Zephyr RTOS shall provide system calls to allow user mode threads to perform privileged operations.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to be able to perform privileged operations in the kernel mode through a well defined interface.

Parents
ZEP-SYRS-9

ZEP-SRS-8-11: User mode handling of detected stack overflow The Zephyr RTOS shall support a defined mechanism for user mode handling a of detected stack overflow.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want, when a stack overflow is detected, to be able to implement a graceful, application defined handling of the exception.

Parents
ZEP-SYRS-9

ZEP-SRS-8-12: Stack Overflow Detection The Zephyr RTOS shall support detection of stack overflows.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to get an indication when a stack overflow occurs at least during debugging / the development phase, and for safety applications also in a release version of my application.

Parents
ZEP-SYRS-9

ZEP-SRS-8-13: Boot Time Memory Access Policy The Zephyr RTOS shall support configurable access to memory during boot time.

Status
Draft

Type
Functional

Component
Memory Protection

Parents
ZEP-SYRS-9

ZEP-SRS-8-14: System Call Handler Functions The Zephyr RTOS shall provide helper functions for system call handler functions to validate the inputs passed in from user mode before invoking the implementation function to protect the kernel.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want Zepyhr OS to validate system call parameters passed from the user mode to the kernel mode to avoid crashes and undefined behaviour.

Parents
ZEP-SYRS-9

ZEP-SRS-8-15: System Call C strings in user mode The Zephyr RTOS shall support system calls to be able to safely accept C strings passed in from user mode.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want Zepyhr OS to validate system call string type parameters passed from the user mode to the kernel mode to avoid crashes and undefined behaviour. e.g. - verify the string length is smaller or equal to the syscalls defined max. - verify that the length type does not overflow when allocating one more byte ???

Parents
ZEP-SYRS-9

ZEP-SRS-8-16: Tracking kernel objects in used by user mode threads The Zephyr RTOS shall track kernel objects that are used by user mode threads. Note: this means Zephyr shall track the resources used by the user mode thread (associate this with a user story).

Status
Draft

Type
Functional

Component
Memory Protection

Parents
ZEP-SYRS-9

ZEP-SRS-8-17: Granting threads access to specific memory The Zephyr RTOS shall have an interface to request access to specific memory after initial allocation.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to be able to request read-only or read-write access to a dedicated memory area/pool during runtime.

Parents
ZEP-SYRS-9

ZEP-SRS-8-18: Assigning memory pools to act as a thread resource pool The Zephyr RTOS shall support assigning a memory pool to act as that thread's resource pool.

Status
Draft

Type
Functional

Component
Memory Protection

User Story
As a Zephyr RTOS user I want to be able, during runtime from the kernel, to request a memory area/pool which is exclusively available to the requesting thread protected against access from other threads.

Parents
ZEP-SYRS-9

ZEP-SRS-9-1: Dynamic Memory Allocation The Zephyr RTOS shall allow threads to dynamically allocate variable-sized memory regions from a specified range of memory.

Status
Draft

Type
Functional

Component
Memory Objects

User Story
As a Zephyr RTOS user I want my application to be able to dynamically allocate memory of a application defined size.

Parents
ZEP-SYRS-9

ZEP-SRS-9-2: Memory Slab Object The Zephyr RTOS shall allow threads to dynamically allocate fixed-sized memory regions from a specified range of memory.

Status
Draft

Type
Functional

Component
Memory Objects

User Story
As a Zephyr RTOS user I want a most efficient and least fragmentation prone dynamic memory allocation mechanism.

Parents
ZEP-SYRS-9

ZEP-SRS-15-1: Traditional FIFO Queue The Zephyr RTOS shall provide a kernel object that implements a traditional first in, first out (FIFO) queue, allowing threads and ISRs to add and remove a limited number of 32-bit data values.

Status
Draft

Type
Functional

Component
Data Passing

User Story
As a Zephyr RTOS user I want to be able to exchange 1 to N data objects between different threads and ISR in a thread-safe manner with a first-in-first-out (queue) behaviour.

ZEP-SRS-15-2: Traditional LIFO queue The Zephyr RTOS shall provide a kernel object that implements a traditional last in, first out (LIFO) queue, allowing threads and ISRs to add and remove a limited number of 32-bit data values.

Status
Draft

Type
Functional

Component
Data Passing

User Story
As a Zephyr RTOS user I want to be able to exchange 1 to N data objects between different threads and ISR in a thread-safe manner with a last-in-first-out (stack) behaviour.

ZEP-SRS-6-1: Mutex Kernel Object The Zephyr RTOS shall provide a mutex that allows threads to obtain mutually exclusive access to a shared resource.

Status
Draft

Type
Functional

Component
Mutex

User Story
As a Zephyr RTOS user I want to be able to synchronize threads when accessing common resources, where the thread shall have the option to: - wait indefinitely until the resource becomes available. - immediately return with an error if the resource is not available and continue. - wait for a given time for the resource to become available or return with an error.

Parents
ZEP-SYRS-13

ZEP-SRS-6-2: Mutex definition at compile time The Zephyr RTOS shall provide a mechanism to define and initialize a mutex at compile time.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-3: Mutex definition at run time The Zephyr RTOS shall provide a mechanism to initialize a mutex at run time.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-4: Locking a mutex The Zephyr RTOS shall provide a mechanism for a thread to lock a mutex and become its owner.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-5: Locking an unowned mutex While a mutex is not owned by any thread, a thread that locks it shall acquire ownership of the mutex.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-6: Locking a mutex owned by another thread While a mutex is owned by another thread, a thread that attempts to lock it shall be blocked until the mutex becomes available or the specified timeout elapses.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-7: Mutex lock timeout When locking a mutex, the Zephyr RTOS shall accept a timeout that specifies the maximum time to wait for the mutex to become available.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-8: Mutex lock timeout expiry When a mutex does not become available within the specified time, the Zephyr RTOS shall return an error indicating that the mutex was not locked.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-9: Recursive mutex locking While a thread owns a mutex, the Zephyr RTOS shall allow that thread to lock the same mutex again, and shall require the thread to unlock it the same number of times before it is released.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-10: Unlocking a mutex The Zephyr RTOS shall provide a mechanism for the owning thread to unlock a mutex.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-11: Unlocking by a non-owner If a thread that does not own a mutex attempts to unlock it, then the Zephyr RTOS shall return an error and shall not release the mutex.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-6-12: Priority inheritance While a higher-priority thread is blocked waiting on a mutex, the Zephyr RTOS shall raise the priority of the owning thread to that of the highest-priority waiting thread until the mutex is released.

Status
Draft

Type
Functional

Component
Mutex

Parents
ZEP-SYRS-13

ZEP-SRS-5-1: Counting Semaphore Definition At Compile Time The Zephyr RTOS shall provide a mechanism to define and initialize a semaphore at compile time.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-2: Counting Semaphore Definition At Run Time The Zephyr RTOS shall provide a mechanism to define and initialize a semaphore at runtime.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-3: Maximum limit of a semaphore The Zephyr RTOS shall define the maximum limit of a semaphore when the semaphore is used for counting purposes and does not have an explicit limit.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-4: Initialialization with maximum count value When initializing a counting semaphore, the maximum permitted count a semaphore can have shall be set.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-5: Initial semaphore value When initializing a counting semaphore, the initial semaphore value shall be set.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-6: Semaphore acquisition mechanism The Zephyr RTOS shall provide a mechanism allowing threads to acquire a semaphore.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-7: Semaphore acquisition with count greater than zero While the semaphore's count is greater than zero, the requesting thread shall acquire the semaphore and decrement its count.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-8: Semaphore acquisition with zero count While the semaphore's count is zero, the requesting thread shall be blocked until the semaphore is released by another thread.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-9: Semaphore acquisition timeout When attempting to acquire a semaphore, the Zephyr RTOS shall accept options that specify timeout periods, allowing threads to set a maximum wait time for semaphore acquisition.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-10: Semaphore acquisition timeout error handling When attempting to acquire a semaphore, where the semaphore is not acquired within the specified time, the Zephyr RTOS shall return an error indicating a timeout.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-11: Semaphore acquisition no wait error handling When attempting to acquire a semaphore, where the current count is zero and no timeout time was provided, the Zephyr RTOS shall return an error indicating the semaphore is busy.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-12: Semaphore release The Zephyr RTOS shall provide a mechanism allowing threads to release a semaphore.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-13: Semaphore count increment on release The Zephyr RTOS shall increment the semaphore's count upon release.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-14: Highest-priority waiter unblocked on release When there are threads waiting on the semaphore, the highest-priority waiting thread shall be unblocked and acquire the semaphore.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-15: Checking semaphore count The Zephyr RTOS shall provide a mechanism for threads to check the current count of a semaphore without acquiring it.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-16: Semaphore reset The Zephyr RTOS shall provide a mechanism that resets the semaphore count to zero.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-17: Semaphore acquisitions abort after reset When a semaphore is reset, the Zephyr RTOS shall abort all pending take operations on the semaphore, returning to the affected threads the same error used to indicate a wait timeout.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-18: Semaphore Initialization Option Validation When initializing a counting semaphore, where the maximum permitted count of a semaphore is invalid, then the Zephyr RTOS shall return an error indicating invalid values.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-19: Semaphore release at maximum count When a semaphore is released while its count is already at the maximum permitted count, the Zephyr RTOS shall leave the count unchanged.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-5-20: Semaphore operations from interrupt context The Zephyr RTOS shall allow a semaphore to be released from an interrupt service routine, and to be acquired from an interrupt service routine when no waiting is requested.

Status
Draft

Type
Functional

Component
Semaphore

Parents
ZEP-SYRS-14

ZEP-SRS-13-1: Power State Control The Zephyr RTOS shall provide control over changes to system power states.

Status
Draft

Type
Functional

Component
Power Management

User Story
See ZEP104 + When the power state is changed I want to get an notification in order for specific parts of my application to react accordingly.

Parents
ZEP-SYRS-10

ZEP-SRS-13-2: Power Management TBD

Status
Draft

Type
Functional

Component
Power Management

Parents
ZEP-SYRS-10

ZEP-SRS-13-3: Notification of changes to system power states The Zephyr RTOS shall provide notification of changes to system power states.

Status
Draft

Type
Functional

Component
Power Management

Parents
ZEP-SYRS-10

ZEP-SRS-3-1: Exchanging data between threads The Zephyr RTOS shall provide a mechanism for exchanging data between threads.

Status
Draft

Type
Functional

Component
Thread Communication

User Story
(Note: copying data may be needed here - what is Zephyr doing? is it configurable?)

ZEP-SRS-3-2: Waiting for results during communication The Zephyr RTOS shall provide mechanisms to enable waiting for results during communication between threads. (NOTE: waiting for results is really bad and dangerous, want to avoid if at all possible).

Status
Draft

Type
Functional

Component
Thread Communication

User Story
=N73

ZEP-SRS-3-3: Poll Operation Support The Zephyr RTOS shall support a poll operation which enables waiting concurrently for any one of multiple conditions to be fulfilled.

Status
Draft

Type
Functional

Component
Thread Communication

User Story
As a Zephyr RTOS user I want my thread to wait for one of several defined events to occur and only continue when one of them has occurred.

ZEP-SRS-3-4: Pipe Communication Primitive The Zephyr RTOS shall provide a communication primitive that allows a thread to transfer a block of data to another thread.

Status
Draft

Type
Functional

Component
Thread Communication

User Story
As a Zephyr RTOS user I want to send a byte stream from one thread to another in a thread-safe manner.

ZEP-SRS-3-5: Message Queue The Zephyr RTOS shall provide a a communication primitive that allow threads and ISRs to asynchronously exchange fixed-size data items.

Status
Draft

Type
Functional

Component
Thread Communication

ZEP-SRS-3-6: Mailbox Kernel Primitive The Zephyr RTOS shall provide a communication primitive that allows threads to exchange messages of varying sizes asynchronously or synchronously.

Status
Draft

Type
Functional

Component
Thread Communication

User Story
As a Zephyr RTOS user I want a sending thread to deposit a message into the mailbox, and a receiving thread to be able to retrieve the message at its convenience. This allows for asynchronous communication, where threads do not need to synchronize their execution explicitly.

ZEP-SRS-2-1: Support operation on more than one CPU The Zephyr RTOS shall support operation on more than one physical CPU sharing the same kernel state.

Status
Draft

Type
Functional

Component
Multi Core

ZEP-SRS-2-2: Running threads on specific CPUs The Zephyr RTOS shall provide an interface for running threads on specific sets of CPUs (default is 1 CPU).

Status
Draft

Type
Functional

Component
Multi Core

User Story
As a Zephyr RTOS user I want Zephyr OS to be able to specify the CPU core or the set of CPU cores on which a thread shall be executed.

ZEP-SRS-2-3: Exclusion between physical CPUs The Zephyr RTOS shall provide an interface for mutual exclusion between multiple physical CPUs.

Status
Draft

Type
Functional

Component
Multi Core

User Story
As a Zephyr RTOS user I want Zephyr OS to provide synchronization mechanisms between the CPU cores and the access to common resources.

ZEP-SRS-2-4: Scheduling a thread based on an event The Zephyr RTOS shall provide an interface to schedule a thread based on an event.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want to be able to execute work which reacts on events and interrupts the current executed work.

Parents
ZEP-SYRS-15

ZEP-SRS-2-5: Deadline Scheduling Priorities The Zephyr RTOS shall organize running threads by earliest deadline first priority.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want to be able to schedule threads by earliest deadline first.

ZEP-SRS-2-6: Work Queue utility capable of running preemptible work items The Zephyr RTOS shall provide a thread-pooled work queue utility capable of running preemptible work items with specific scheduler priorities.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want to be able to add work items in a thread work queue with different priorities and these shall be scheduled according their priorities.

ZEP-SRS-2-7: Run user supplied functions in-order in a separate thread(s) The Zephyr RTOS shall provide an interface for running user-supplied functions.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want to be able to run functions in-order in a separated thread/threads.

ZEP-SRS-2-8: Organize running threads into a fixed list The Zephyr RTOS shall organize running threads into a fixed list of numeric priorities.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want that the OS organize running threads in a fixed list of numeric priorities.

ZEP-SRS-2-9: Preemption support The Zephyr RTOS shall support preemption of a running thread by a higher priority thread.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want that the OS preempt running threads by a thread with higher priority.

ZEP-SRS-2-10: Un-preemptible thread priorities The Zephyr RTOS shall support thread priorities which cannot be preempted by other user threads.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want to be able to configure thread prioritizes which cannot be preempted by other user threads.

ZEP-SRS-2-11: Time sharing of CPU resources The Zephyr RTOS shall support time sharing of CPU resources among threads of the same priority.

Status
Draft

Type
Functional

Component
Thread Scheduling

User Story
As a Zephyr RTOS user, I want to be able to configure my RTOS in the way, that the CPU resources are shared evenly among executed threads of the same priority.

ZEP-SRS-1-1: Creating threads The Zephyr RTOS shall provide an interface to create (start) a thread.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-2: Setting thread priority The Zephyr RTOS shall provide an interface to set a thread's priority.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-17

ZEP-SRS-1-3: Suspending a thread The Zephyr RTOS shall provide an interface to suspend a thread.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-4: Resuming a suspended thread The Zephyr RTOS shall provide an interface to resume a suspended thread.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-5: Resuming a suspended thread after a timeout The Zephyr RTOS shall provide an interface to resume a suspended thread after a timeout.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-6: Deleting a thread The Zephyr RTOS shall provide an interface to delete (end) a thread.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-7: Thread states Threads shall have different states to fulfill the Life-cycle of a thread

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user, I want to know in what state a specific thread is.

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-8: Thread stack objects Every Thread shall have it's own stack.

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user I want to be able to configure the stack size of a thread. And every thread shall have it's own dedicated stack.

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-9: Thread privileges The Zephyr RTOS shall provide an interface to create threads with defined privilege.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-10: Scheduling multiple threads The Zephyr RTOS shall provide an interface to schedule multiple threads.

Status
Draft

Type
Functional

Component
Threads

Parents
ZEP-SYRS-16

ZEP-SRS-1-11: Thread Options The Zephyr RTOS shall support a set of thread options.

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user, I want to be able to pass specific option to a thread.

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-1-12: Thread Custom Data Every thread shall have a custom data area.

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user, I want to be able to set a thread specific custom area for every thread I create and which can be used only by the thread itself or the can be used by the application

Parents
ZEP-SYRS-15, *ZEP-SYRS-16*

ZEP-SRS-4-1: Timer definition at compile time The Zephyr RTOS shall provide a mechanism to define and statically (i.e. compile time) initialize timers.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-2: Timer expiry function When initializing a timer, the Zephyr RTOS shall support setting a function that gets called when the timer expires.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-3: Timer stop function When initializing a timer, the Zephyr RTOS shall support setting a function that gets called when a running timer is stopped.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-4: Timer definition at run time The Zephyr RTOS shall provide a mechanism to define and initialize timers at run time.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-5: Timer start The Zephyr RTOS shall provide a mechanism to start a timer for a specific duration and periodicity.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-6: Timer stop The Zephyr RTOS shall provide a mechanism to stop a running timer.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-7: Timer status The Zephyr RTOS shall provide a mechanism to read the number of times a timer has expired and then reset this count to zero.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-8: Timer status reinitialization When a timer is initialized, started, or read via the timer status or synchronization mechanism, the Zephyr RTOS shall reset the timer's expiration count to zero.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-9: Timer thread synchronization The Zephyr RTOS shall provide a mechanism to synchronize a thread to a timer and then block the thread's execution until any of the following conditions is satisfied: - The timer is stopped - The timer's expiration count is greater than zero

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-10: Timer next expiration time in system ticks The Zephyr RTOS shall provide a mechanism to get a timer's next expiration time in system ticks.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-11: Timer remaining time until expiration in system ticks The Zephyr RTOS shall provide a mechanism to get a timer's remaining time until its next expiry in system ticks.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-12: Timer remaining time until expiration in milliseconds The Zephyr RTOS shall provide a mechanism to get the timer's remaining time until its next expiry in milliseconds.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-13: Timer set user data The Zephyr RTOS shall support adding user defined data to a timer.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-14: Timer get user data The Zephyr RTOS shall support retrieving user defined data from a timer.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-4-15: Timer expire functions in interrupt context When a timer expiry function is called, the Zephyr RTOS shall do so in the interrupt context.

Status
Draft

Type
Functional

Component
Timers

Parents
ZEP-SYRS-18

ZEP-SRS-10-1: Initializing a trace Zephyr shall provide an interface to initialize a trace.

Status
Draft

Type
Functional

Component
Tracing

Parents
ZEP-SYRS-19

ZEP-SRS-10-2: Triggering a trace Zephyr shall provide an interface to trigger a trace.

Status
Draft

Type
Functional

Component
Tracing

Parents
ZEP-SYRS-19

ZEP-SRS-10-3: Dumping trace results Zephyr shall provide an interface to dump results from a trace.

Status
Draft

Type
Functional

Component
Tracing

Parents
ZEP-SYRS-19

ZEP-SRS-10-4: Removing trace data Zephyr shall provide an interface to remove trace data.

Status
Draft

Type
Functional

Component
Tracing

Parents
ZEP-SYRS-19

ZEP-SRS-10-5: Tracing Object Identification Zephyr shall provide an interface to identify the objects being traced.

Status
Draft

Type
Functional

Component
Tracing

User Story
As a Zephyr OS user, I want to be able to give each object which I create a name / label in order to be able to identify them in a trace log.

Parents
ZEP-SYRS-19

ZEP-SRS-10-6: Tracing Non-Interference Zephyr shall prevent the tracing functionality from interfering with normal operations.

Status
Draft

Type
Functional

Component
Tracing

User Story
As a Zephyr OS user, I want that the trace functionality do not interfere or slow down the operation system functionality.

Parents
ZEP-SYRS-19

ZEP-SRS-21-1: Dynamic initialization of condition variables The Zephyr RTOS shall provide a mechanism to define and initialize a condition variable dynamically (at runtime).

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-2: Static initialization of condition variables The Zephyr RTOS shall provide a mechanism to define and initialize a condition variable statically (at compile time).

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-3: Signal one waiting thread The Zephyr RTOS shall provide a mechanism to signal the highest priority waiting thread when a condition is met.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-4: Signal multiple waiting threads The Zephyr RTOS shall provide a mechanism to signal all waiting threads when a condition is met.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-5: Wait on a condition variable The Zephyr RTOS shall provide a mechanism for a thread to wait on a condition variable.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-6: Wait timeout on a condition variable When waiting on a condition variable, the thread shall specify a timeout value.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-7: Wait timeout occurence When a timeout occurs while waiting on a condition variable, the thread shall be unblocked and a timeout error shall be returned.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-8: Release mutex on wait When a thread waits on a condition variable, the Zephyr RTOS shall atomically release the associated mutex while the thread is blocked and re-acquire it before the thread resumes.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-21-9: Unblock a waiting thread Whenever a thread signals a condition variable, the Zephyr RTOS shall unblock the highest priority thread currently waiting on that condition variable.

Status
Draft

Type
Functional

Component
Condition Variables

Parents
ZEP-SYRS-21

ZEP-SRS-20-1: Queue definition at compile time The Zephyr RTOS shall provide a mechanism to define and initialize a queue at compile time.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-2: Queue definition at run time. The Zephyr RTOS shall provide a mechanism to define and initialize a queue at run time.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-3: Queue enqueue item to the back The Zephyr RTOS shall provide a mechanism to enqueue a data item to the back of a queue (i.e. append).

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-4: Queue enqueue item to the front The Zephyr RTOS shall provide a mechanism to enqueue a data item to the front of a queue (i.e. prepend).

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-5: Queue remove item The Zephyr RTOS shall provide a mechanism to remove a specific data item from a queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-6: Queue get and dequeue item from front The Zephyr RTOS shall provide a mechanism to get and dequeue a data item from the front of a queue, within a timeout.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-7: Queue empty check The Zephyr RTOS shall provide a mechanism to check if a queue is empty.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-8: Queue peek item at back The Zephyr RTOS shall provide a mechanism to peek at the data data item at the back of a queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-9: Queue peek item at front The Zephyr RTOS shall provide a mechanism to peek at the data data item at the front of a queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-10: Queue insert The Zephyr RTOS shall provide a mechanism to insert a data item behind another specific data item in a queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-11: Queue append item list to the back The Zephyr RTOS shall provide a mechanism to append a list of data items to the back of a queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-12: Queue append and empty item list to the back The Zephyr RTOS shall provide a mechanism to append a list of data items to the back of a queue and then empty the list.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-13: Queue append unique data item The Zephyr RTOS shall provide a mechanism to append a data item uniquely to the queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-20-14: Queue implicit memory allocation The Zephyr RTOS shall provide a dedicated mechanism to implicite allocate memory from a thread when appending data items to a queue.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-22

ZEP-SRS-23-1: LIFO definition at compile time The Zephyr RTOS shall provide a mechanism to define and initialize a LIFO at compile time.

Status
Draft

Type
Functional

Component
LIFOs

Parents
ZEP-SYRS-23

ZEP-SRS-23-2: LIFO enqueue item to the back The Zephyr RTOS shall provide a mechanism to enqueue a data item to the back of a LIFO (i.e. prepend).

Status
Draft

Type
Functional

Component
LIFOs

Parents
ZEP-SYRS-23

ZEP-SRS-23-3: LIFO implicit memory allocation The Zephyr RTOS shall provide a mechanism to allocate memory from a thread when enqueuing data items to a LIFO.

Status
Draft

Type
Functional

Component
LIFOs

Parents
ZEP-SYRS-23

ZEP-SRS-23-4: LIFO get and dequeue item from front The Zephyr RTOS shall provide a mechanism to get and dequeue a data item from the back of a LIFO, within a timeout.

Status
Draft

Type
Functional

Component
LIFOs

Parents
ZEP-SYRS-23

ZEP-SRS-23-5: Queue definition at compile time The Zephyr RTOS shall provide a mechanism to define and initialize a LIFO at compile time.

Status
Draft

Type
Functional

Component
Queues

Parents
ZEP-SYRS-23

ZEP-SRS-24-1: FIFO definition at compile time The Zephyr RTOS shall provide a mechanism to define and initialize a FIFO at compile time.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-2: FIFO cancel waiting thread The Zephyr RTOS shall provide a mechanism for the first thread waiting to get and dequeue a data item to return as if the timeout had expired.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-3: FIFO enqueue item to the back The Zephyr RTOS shall provide a mechanism to enqueue a data item to the back of a FIFO (i.e. append).

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-4: FIFO implicit memory allocation The Zephyr RTOS shall provide a mechanism to allocate memory from a thread when enqueuing data items to a FIFO.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-5: FIFO append item list to the back The Zephyr RTOS shall provide a mechanism to append a list of data items to the back of a FIFO.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-6: FIFO append and empty item list to the back The Zephyr RTOS shall provide a mechanism to append a list of data items to the back of a FIFO and then empty the list.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-7: FIFO get and dequeue item from front The Zephyr RTOS shall provide a mechanism to get and dequeue a data item from the front of a FIFO, within a timeout.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-8: FIFO empty check The Zephyr RTOS shall provide a mechanism to check if a FIFO is empty.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-9: FIFO peek item at front The Zephyr RTOS shall provide a mechanism to peek at the data data item at the front of a FIFO.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-24-10: FIFO peek item at back The Zephyr RTOS shall provide a mechanism to peek at the data item at the back of a FIFO.

Status
Draft

Type
Functional

Component
FIFOs

Parents
ZEP-SYRS-24

ZEP-SRS-25-1: Mailbox Initialization At Run Time The Zephyr RTOS shall provide a mechanism to define and initialize a mailbox at run time.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to initialize a mailbox instance so that it is ready to be used for message exchange between threads.

Parents
ZEP-SYRS-25

ZEP-SRS-25-2: Mailbox Initialization At Compile Time The Zephyr RTOS shall provide a mechanism to statically define and initialize a mailbox object at compile time.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to statically define and initialize a mailbox at compile time so that it is available immediately without runtime initialization.

Parents
ZEP-SYRS-25

ZEP-SRS-25-3: Message Size Support The Zephyr RTOS shall support messages containing zero or more bytes of data.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to send messages of different sizes using the same mailbox so that I can efficiently communicate different types of information between threads.

Parents
ZEP-SYRS-25

ZEP-SRS-25-4: Mailbox Data Transfer The Zephyr RTOS shall handle the data transfer between mailbox objects of the sending and receiving threads.

Status
Draft

Type
Functional

Component
Mailboxes

Parents
ZEP-SYRS-25

ZEP-SRS-25-5: Synchronous Message Sending The Zephyr RTOS shall provide a mechanism for a thread to send a message through a mailbox and block until the message is processed or a timeout occurs.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to send a message and wait for it to be processed so that I can be sure the message was received before continuing execution.

Parents
ZEP-SYRS-25

ZEP-SRS-25-6: Synchronous Send Timeout Handling If the synchronous message send operation times out before a receiver processes the message, the Zephyr RTOS shall return an timeout error code to the sending thread.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to limit how long my thread waits for a message to be received so that I can implement fallback behavior if communication takes too long.

Parents
ZEP-SYRS-25

ZEP-SRS-25-7: Asynchronous Message Sending The Zephyr RTOS shall provide a mechanism for a thread to send a message through a mailbox without waiting for it to be processed.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to send a message without waiting for it to be processed so that my thread can continue execution immediately.

Parents
ZEP-SYRS-25

ZEP-SRS-25-8: Asynchronous Completion Signaling When a sending thread asynchronously sends a message to a mailbox object, the Zephyr RTOS shall provide a mechanism to signal to the sending thread that the message has been both received and completely processed by the receiver.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be notified when my asynchronously sent message has been fully processed so that I can perform follow-up actions without polling.

Parents
ZEP-SYRS-25

ZEP-SRS-25-9: Message Reception The Zephyr RTOS shall provide a mechanism for a thread to receive a message via a mailbox object with a timeout parameter.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to receive messages from other threads with control over how long to wait for a message to arrive.

Parents
ZEP-SYRS-25

ZEP-SRS-25-10: Message Data Retrieval The Zephyr RTOS shall provide a mechanism for a thread to retrieve message data via a mailbox object.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to extract message data into my buffer after message reception so that I can process the information contained in the message.

Parents
ZEP-SYRS-25

ZEP-SRS-25-11: Message Reception Blocking When a receiving thread requests a message via a mailbox object and no message is available, the Zephyr RTOS shall block the receiving thread until a message is available or the timeout expires.

Status
Draft

Type
Functional

Component
Mailboxes

Parents
ZEP-SYRS-25

ZEP-SRS-25-12: Message Reception Timeout Handling If the message receive operation times out before a message becomes available, the Zephyr RTOS shall return an appropriate timeout error code to the receiving thread.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to limit how long my thread waits for a message to arrive so that I can implement fallback behavior if communication takes too long.

Parents
ZEP-SYRS-25

ZEP-SRS-25-13: Non-Anonymous Messaging The Zephyr RTOS shall handle message exchange via a mailbox object non-anonymously, allowing both the sending and receiving threads to know the identity of the other thread.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to identify the source or destination of messages so that I can implement thread-specific handling of message content.

Parents
ZEP-SYRS-25

ZEP-SRS-25-14: Priority-Based Message Delivery When multiple threads are waiting on an empty mailbox object, the Zephyr RTOS shall deliver the next message to the highest priority thread.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want message delivery to respect thread priorities so that higher priority tasks receive messages before lower priority ones.

Parents
ZEP-SYRS-25

ZEP-SRS-25-15: Priority-Based Message Delivery When multiple threads of equal priority are waiting on an empty mailbox object, the Zephyr RTOS shall deliver the next message to the thread that has waited the longest.

Status
Draft

Type
Functional

Component
Mailboxes

Parents
ZEP-SYRS-25

ZEP-SRS-25-16: Mailbox Quantity Support The Zephyr RTOS shall support an arbitrary number of mailbox objects, limited only by available RAM in the system.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want to be able to create as many mailboxes as my application needs without arbitrary limitations so that I can design my communication architecture freely.

Parents
ZEP-SYRS-25

ZEP-SRS-25-17: Invalid Parameter Handling The Zephyr RTOS shall handle invalid parameters by returning error codes rather than causing system failures.

Status
Draft

Type
Functional

Component
Mailboxes

User Story
As a Zephyr RTOS developer, I want the system to validate my inputs and return errors rather than crash when I provide invalid parameters so that I can develop more robust applications.

Parents
ZEP-SYRS-25

ZEP-SRS-30-1: Stack definition at compile time The Zephyr RTOS shall provide a mechanism to define and initialize a stack at compile time.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30

ZEP-SRS-30-2: Stack initialization at run time with an user-defined memory area The Zephyr RTOS shall provide a mechanism to initialize a stack at run time, using a data buffer provided by the user.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30

ZEP-SRS-30-3: Stack initialization at run time with automatically allocated memory area The Zephyr RTOS shall provide a mechanism to initialize a stack at run time, with automatic allocation of the stack memory.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30

ZEP-SRS-30-4: Releasing a dynamically allocated stack buffer The Zephyr RTOS shall provide a mechanism to deallocate all resources previously allocated by a run-time stack definition.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30

ZEP-SRS-30-5: Push an item to the stack The Zephyr RTOS shall provide a mechanism to add a new item on top of the stack.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30, *ZEP-SRS-30-7*

ZEP-SRS-30-6: Pop an item from the stack The Zephyr RTOS shall provide a mechanism to retrieve the current item from the top of the stack and to remove such item from the stack.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30, *ZEP-SRS-30-9*

ZEP-SRS-30-7: Pushing onto a full stack If the stack is full when an item is pushed, the Zephyr RTOS shall return an error and not add the item.

Status
Draft

Type
Functional

Component
Stacks

Parents
ZEP-SYRS-30

ZEP-SRS-30-8: Concurrent access protection The Zephyr RTOS shall provide a mechanism to push items to or pop items from the same stack object from multiple execution contexts which can run concurrently.

Status
Draft

Type
Functional

Component
Stacks

User Story
As a Zephyr RTOS user I want to access a stack from different execution contexts (like threads), including interrupt handlers.

Parents
ZEP-SYRS-30

ZEP-SRS-30-9: Stack data retrieval with timeout The Zephyr RTOS shall provide a mechanism to allow waiting for a stack item availability within a specified timeout.

Status
Draft

Type
Functional

Component
Stacks

User Story
As a Zephyr RTOS user I want to share a stack object between a producer and a consumer threads. The producer thread pushes items into the stack, while the consumer thread is waiting for an item to appear in the stack.

Parents
ZEP-SYRS-30

ZEP-SRS-26-1: Atomic variable The Zephyr RTOS shall define an atomic variable type whose size matches the native word size of the target architecture: 32 bits on 32-bit architectures and 64 bits on 64-bit architectures.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-2: Atomic value The Zephyr RTOS shall define a signed integer type, whose size is determined by the target processor architecture, used as the parameter and return type of operations on atomic variables.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-3: Tearing freedom The Zephyr RTOS shall guarantee that each operation on an atomic variable is free from torn reads or writes.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-4: Cross-CPU visibility The Zephyr RTOS shall guarantee that the result of each operation on an atomic variable is visible to all processors in the system.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-5: Indivisibility The Zephyr RTOS shall provide a mechanism for performing all operations on an atomic variable in an indivisible manner.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-6: Memory ordering When performing an operation on an atomic variable, the Zephyr RTOS shall provide full memory barrier semantics, such that memory operations issued before the atomic operation are observed before it and memory operations issued after it are observed after it.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-7: Software implementation fallback The Zephyr RTOS shall support executing atomic operations on hardware that does not provide native atomic instruction support.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-8: Native atomic instructions Where the target hardware provides native atomic instructions, the Zephyr RTOS shall support implementing atomic operations using those instructions.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-9: Compute size of atomic variables The Zephyr RTOS shall provide a compile-time mechanism for computing the minimum number of atomic variables necessary to represent a bit array of a given number of bits.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-10: Define an array of atomic variables The Zephyr RTOS shall provide a compile-time mechanism for defining an array of atomic variables with the minimum size necessary to represent a bit array of a given number of bits.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-11: Initialize atomic variable The Zephyr RTOS shall provide a compile-time mechanism for initializing an atomic variable.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-12: Set atomic variable The Zephyr RTOS shall provide a mechanism for setting an atomic variable and returning its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-13: Get atomic variable The Zephyr RTOS shall provide a mechanism for getting the value from an atomic variable.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-14: Atomic bitwise operations When performing a bitwise operation on an atomic variable, the Zephyr RTOS shall store the result in the atomic variable and return its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-15: Atomic bitwise AND operation The Zephyr RTOS shall provide a mechanism for performing a bitwise AND operation on an atomic variable with a given value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-14

ZEP-SRS-26-16: Atomic bitwise OR operation The Zephyr RTOS shall provide a mechanism for performing a bitwise OR operation on an atomic variable with a given value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-14

ZEP-SRS-26-17: Atomic bitwise NAND operation The Zephyr RTOS shall provide a mechanism for performing a bitwise NAND operation on an atomic variable with a given value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-14

ZEP-SRS-26-18: Atomic bitwise XOR operation The Zephyr RTOS shall provide a mechanism for performing a bitwise XOR operation on an atomic variable with a given value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-14

ZEP-SRS-26-19: Atomic addition The Zephyr RTOS shall provide a mechanism for adding a value to an atomic variable and returning its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-20: Atomic increment The Zephyr RTOS shall provide a mechanism for incrementing an atomic variable by 1 and returning its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-19

ZEP-SRS-26-21: Atomic subtraction The Zephyr RTOS shall provide a mechanism for subtracting a value from an atomic variable and returning its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-22: Atomic decrement The Zephyr RTOS shall provide a mechanism for decrementing an atomic variable by 1 and returning its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-21

ZEP-SRS-26-23: Atomic compare and set The Zephyr RTOS shall provide a mechanism for setting an atomic variable to a new value if and only if its current value equals an expected value, leaving the atomic variable unchanged otherwise, and returning whether the atomic variable was updated.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-24: Atomic clear The Zephyr RTOS shall provide a mechanism for setting an atomic variable to zero and returning its previous value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-25: Atomic bit operations The Zephyr RTOS shall support performing bit-level operations on individual bits within either a single atomic variable or an array of atomic variables.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-26: Atomic set bit The Zephyr RTOS shall provide a mechanism for setting a specific bit within an atomic variable or an array of atomic variables to 1.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-25

ZEP-SRS-26-27: Atomic clear bit The Zephyr RTOS shall provide a mechanism for clearing a specific bit within an atomic variable or an array of atomic variables to 0.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-25

ZEP-SRS-26-28: Atomic set bit to a given state The Zephyr RTOS shall provide a mechanism for setting a specific bit within an atomic variable or an array of atomic variables to a given boolean value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-25

ZEP-SRS-26-29: Atomic test bit The Zephyr RTOS shall provide a mechanism for getting the state of a specific bit within an atomic variable or an array of atomic variables, returning its current state as a boolean value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-25

ZEP-SRS-26-30: Atomic test and set bit The Zephyr RTOS shall provide a mechanism for setting a specific bit within an atomic variable or an array of atomic variables to 1 and returning its previous state.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-25

ZEP-SRS-26-31: Atomic test and clear bit The Zephyr RTOS shall provide a mechanism for clearing a specific bit within an atomic variable or an array of atomic variables to 0 and returning its previous state.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-25

ZEP-SRS-26-32: Atomic pointer The Zephyr RTOS shall define an atomic variable type for storing a pointer value atomically.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-33: Atomic pointer value The Zephyr RTOS shall define a pointer type used as the parameter and return type of all atomic pointer operations.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-34: Initialize atomic pointer The Zephyr RTOS shall provide a compile-time mechanism for initializing an atomic pointer to a specific pointer value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-35: Atomic pointer operations The Zephyr RTOS shall provide operations that read and write atomic pointers using the atomic pointer value type.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

ZEP-SRS-26-36: Get atomic pointer The Zephyr RTOS shall provide a mechanism for getting the pointer value from an atomic pointer.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-35

ZEP-SRS-26-37: Set atomic pointer The Zephyr RTOS shall provide a mechanism for setting an atomic pointer and returning its previous pointer value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-35

ZEP-SRS-26-38: Atomic pointer compare and set The Zephyr RTOS shall provide a mechanism for setting an atomic pointer to a new value if and only if its current value equals an expected value, leaving the atomic pointer unchanged otherwise, and returning whether the atomic pointer was updated.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-35

ZEP-SRS-26-39: Atomic pointer clear The Zephyr RTOS shall provide a mechanism for setting an atomic pointer to NULL and returning its previous pointer value.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SRS-26-35

ZEP-SRS-26-40: Usable from threads and interrupt service routines The Zephyr RTOS shall support performing atomic operations from threads and from interrupt service routines, guaranteeing correct results when multiple contexts operate concurrently on the same atomic variable.

Status
Draft

Type
Functional

Component
Atomic Service

Parents
ZEP-SYRS-26

Zephyr System Requirements

ZEP-SYRS-1: Architecture Layer Interface The Zephyr RTOS shall provide a framework to communicate with a set of hardware architectural services.

Status
Draft

Type
Functional

Component
Hardware Architecture Interface

User Story
As a Zephyr RTOS user I want to be able to easily switch my application to a different MCU architecture (x86, ARM Cortex-M/A, RISCV etc.).

ZEP-SYRS-2: Support multiprocessor management The Zephyr RTOS shall support symmetric multiprocessing on multiple cores.

Status
Draft

Type
Functional

Component
Hardware Architecture Interface

User Story
As a Zephyr RTOS user I want to use Zephyr OS on multi core (SMP-)MCUs/MPUs.

Parents
ZEP-SYRS-1

ZEP-SYRS-3: Support Subset of Standard C Library The Zephyr RTOS shall support a subset of the standard C library.

Status
Draft

Type
Functional

Component
C Library

User Story
As a Zephyr RTOS user I want to have a selection of standard C library implementations e.g. a full extend and a minimal with a smaller footprint or a particular fast executing implementation.

ZEP-SYRS-4: Device Driver Abstraction The Zephyr RTOS shall provide a framework for managing device drivers and peripherals.

Status
Draft

Type
Functional

Component
Device Drivers

User Story
As a Zephyr RTOS user I want my application to be portable between different MCU architectures (ARM Cortex-M/A, Intel x86, RISCV etc.) and MCU vendors (STM, NXP, Intel, etc.) without having to change the MCU peripherals access.

ZEP-SYRS-5: Fatal error and exception handling The Zephyr RTOS shall provide a framework for error and exception handling.

Status
Draft

Type
Functional

Component
Exception and Error Handling

User Story
As a Zephyr RTOS user I want errors and exceptions to handled and react according to my applications requirements (e.g. reach/establish the applications safety state).

ZEP-SYRS-6: Common File system operation support The Zephyr RTOS shall provide a framework for managing file system access.

Status
Draft

Type
Functional

Component
File Systems

User Story
As a Zephyr RTOS user I want a posix / c like file system access to store data.

ZEP-SYRS-7: Interrupt Management The Zephyr RTOS shall provide a framework for interrupt and interrupt service routine management.

Status
Draft

Type
Functional

Component
Interrupts

User Story
As the Zephyr RTOS user I want the Kernel to provide abstracted interfaces to the platform enabling me to implement standard interrupts interrupt service routines without detailed knowledge of the platform architecture and programming model.

ZEP-SYRS-20: Direct ISR, Platform Specific helpers. Zephyr RTOS shall support development of direct ISRs by providing platform specific code fragments and factory functions including, minimal header instructions, minimal footer instructions, low power termination, and ISR construction.

Status
Draft

Type
Functional

Component
Interrupts

User Story
As the Zephyr RTOS user I want the Kernel to provide support for implementing standard low latency and low power interrupt service routines without detailed knowledge of the platform architecture and programming model.

Parents
ZEP-SYRS-7

ZEP-SYRS-8: Logging The Zephyr RTOS shall provide a framework for logging events.

Status
Draft

Type
Functional

Component
Logging

User Story
As a Zephyr RTOS user I want to be able to log application defined events as well as framework exceptions.

ZEP-SYRS-9: Memory Management framework The Zephyr RTOS shall support a memory management framework.

Status
Draft

Type
Functional

Component
Memory Management

User Story
As a Zephyr RTOS user I want memory to be allocated and protected to my application threads preventing mistakenly access to foreign memory as far as the hardware allows.

ZEP-SYRS-10: Power Management The Zephyr RTOS shall provide an interface to control hardware power states.

Status
Draft

Type
Functional

Component
Power Management

User Story
As a Zephyr RTOS user I want to be able to control the power mode of the MCU and its peripherals to take advantage of the hardware features and to be able to implement low power or battery driven long life applications.

ZEP-SYRS-11: Multiple CPU scheduling The Zephyr RTOS shall support scheduling of threads on multiple hardware CPUs.

Status
Draft

Type
Functional

Component
SMP and Multi core

User Story
As a Zephyr RTOS user I want Zephyr OS to run on MCUs/CPUs with one or more CPU cores.

ZEP-SYRS-12: Scheduling The Zephyr RTOS shall provide an interface to assign a thread to a specific CPU.

Status
Draft

Type
Functional

Component
SMP and Multi core

User Story
As a Zephyr RTOS user, I want to be able to control which thread will run on which CPU.

ZEP-SYRS-13: Mutex The Zephyr RTOS shall provide an interface for managing communication between threads.

Status
Draft

Type
Functional

Component
Mutex

User Story
As a Zephyr RTOS user I want to able to exchange information between threads in a thread-safe manner guaranteeing data consistence.

ZEP-SYRS-14: Counting Semaphore The system shall implement a semaphore synchronization primitive for coordinating access to shared resources among multiple threads.

Status
Draft

Type
Functional

Component
Semaphore

ZEP-SYRS-15: Thread support The Zephyr RTOS shall support threads.

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user, I want to be able to have support for the kernel objects named threads for processing work.

ZEP-SYRS-16: Thread management The Zephyr RTOS shall provide a framework for managing multiple threads of execution.

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user, I want to be able to manage the execute of multiple threads with different priorities.

ZEP-SYRS-17: Thread priority Threads shall have a priority.

Status
Draft

Type
Functional

Component
Threads

User Story
As a Zephyr RTOS user, I want to be able to give my threads different priorities for execution.

ZEP-SYRS-18: Timers The Zephyr RTOS shall provide a framework for managing time-based events.

Status
Draft

Type
Functional

Component
Timers

User Story
As a Zephyr RTOS user, I want to start, suspend, resume and stop timers which shall trigger an event on a set expiration time.

ZEP-SYRS-19: Tracing Zepyhr shall provide a framework mechanism for tracing low level system operations (NOTE: system calls, interrupts, kernel calls, thread, synchronization, etc.).

Status
Draft

Type
Functional

Component
Tracing

User Story
As a Zephyr RTOS user, I want to be able to trace different OS operations.

ZEP-SYRS-21: Condition Variables The Zephyr RTOS shall provide a framework to synchronize threads based on a condition variable.

Status
Draft

Type
Non-Functional

Component
Condition Variables

ZEP-SYRS-22: Queues data passing The Zephyr RTOS shall implement a queue which can be used to pass data between threads and interrupt service routines.

Status
Draft

Type
Functional

Component
Queues

ZEP-SYRS-23: LIFOs data passing The Zephyr RTOS shall implement a LIFOs which can be used to pass data between threads and interrupt service routines.

Status
Draft

Type
Functional

Component
LIFOs

ZEP-SYRS-24: FIFOs data passing The Zephyr RTOS shall implement FIFOs which can be used to pass data between threads and interrupt service routines.

Status
Draft

Type
Functional

Component
FIFOs

ZEP-SYRS-25: Mailboxes The Zephyr RTOS shall provide a framework to pass messages of arbitrary size between threads.

Status
Draft

Type
Functional

Component
Mailboxes

ZEP-SYRS-26: Atomic Service The Zephyr RTOS shall provide an interface to perform atomic operations on variables, guaranteeing that all reads and writes are free from tearing across all processors.

Status
Draft

Type
Functional

Component
Atomic Service

ZEP-SYRS-30: Stacks The Zephyr RTOS shall implement a stack which can be used to pass data between threads and interrupt service routines.

Status
Draft

Type
Functional

Component
Stacks

Chapter 11

Glossary of Terms

API
(Application Program Interface) A defined set of routines and protocols for building application software.

application
The set of user-supplied files that the Zephyr build system uses to build an application image for a specified board configuration. It can contain application-specific code, kernel configuration settings, and at least one CMakeLists.txt file. The application's kernel configuration settings direct the build system to create a custom kernel that makes efficient use of the board's resources. An application can sometimes be built for more than one type of board configuration (including boards with different CPU architectures), if it does not require any board-specific capabilities.

application image
A binary file that is loaded and executed by the board for which it was built. Each application image contains both the application's code and the Zephyr kernel code needed to support it. They are compiled as a single, fully-linked binary. Once an application image is loaded onto a board, the image takes control of the system, initializes it, and runs as the system's sole application. Both application code and kernel code execute as privileged code within a single shared address space.

architecture
An instruction set architecture (ISA) along with a programming model.

board
A target system with a defined set of devices and capabilities, which can load and execute an application image. It may be an actual hardware system or a simulated system running under QEMU. A board can contain one or more *SoCs*. The Zephyr kernel supports a variety of boards.

board configuration
A set of kernel configuration options that specify how the devices present on a board are used by the kernel. The Zephyr build system defines one or more board configurations for each board it supports. The kernel configuration settings that are specified by the build system can be over-ridden by the application, if desired.

board name
The human-readable name of a *board*. Uniquely and descriptively identifies a particular system, but does not include additional information that may be required to actually build a Zephyr image for it. See *Board terminology* for additional details.

board qualifiers
The set of additional tokens, separated by a forward slash (/) that follow the *board name* (and optionally *board revision*) to form the *board target*. The currently accepted qualifiers are *SoC*, *CPU cluster* and *variant*. See *Board terminology* for additional details.

board revision
An optional version string that identifies a particular revision of a hardware system. This is useful to avoid duplication of board files whenever small changes are introduced to a hardware system. See *Multiple board revisions* and *Building for a board revision* for more information.

board target
The full string that can be provided to any of the Zephyr build tools to compile and link an image for a particular hardware system. This string uniquely identifies the combination of *board name*, *board revision* and *board qualifiers*. See *Board terminology* for additional details.

CPU cluster
A group of one or more *CPU cores*, all executing the same image within the same address space and in a symmetrical (SMP) configuration. Only *CPU cores* of the same *architecture* can be in a single cluster. Multiple CPU clusters (each of one or more cores) can coexist in the same *SoC*.

CPU core
A single processing unit, with its own Program Counter, executing program instructions sequentially. CPU cores are part of a *CPU cluster*, which can contain one or more cores.

device runtime power management
Device Runtime Power Management (PM) refers the capability of devices to save energy independently of the system power state. Devices will keep reference of their usage and will automatically be suspended or resumed. This feature is enabled via the `CONFIG_PM_DEVICE_RUNTIME` Kconfig option.

idle thread
A system thread that runs when there are no other threads ready to run.

IDT
(Interrupt Descriptor Table) a data structure used by the x86 architecture to implement an interrupt vector table. The IDT is used to determine the correct response to interrupts and exceptions.

internal API
Any internal function, structure or macro defined anywhere in the Zephyr source tree. Internal APIs are intended to "extend" Zephyr, and are to be used only between certain *software components*, usually in-tree but in some cases out-of-tree (e.g. adding an out-of-tree architecture or driver). Applications must not invoke internal APIs outside their own scope. The context where the API is called or implemented is well defined. For example, functions prefixed with `arch_` are intended for use by the Zephyr kernel to invoke architecture-specific code. Internal APIs are mostly kept stable, but with fewer guarantees than *public APIs*.

ISR
(Interrupt Service Routine) Also known as an interrupt handler, an ISR is a callback function whose execution is triggered by a hardware interrupt (or software interrupt instructions) and is used to handle high-priority conditions that require interrupting the current code executing on the processor.

kernel
The set of Zephyr-supplied files that implement the Zephyr kernel, including its core services, device drivers, network stack, and so on.

power domain
A power domain is a collection of devices for which power is applied and removed collectively in a single action. Power domains are represented by `device`.

power gating
Power gating reduces power consumption by shutting off areas of an integrated circuit that are not in use.

private API
Any function, structure or macro defined anywhere in the Zephyr source tree which are only intended for consumption inside the *software component* where they are defined. Private APIs may change at any time and must not be used by code outside the corresponding software component.

public API
Any function, structure or macro defined inside the `include/zephyr` folder that is not explicitly marked as private. Public APIs are intended for consumption by any and all in-tree or out-of-tree *software components*. Public APIs cannot be modified without following the provisions described in the *API lifecycle* section, which means they provide guarantees that they will remain stable over time.

SoC
A System on a chip, that is, an integrated circuit that contains at least one *CPU cluster* (in turn with at least one *CPU core*), as well as peripherals and memory.

SoC family
One or more *SoCs* or *SoC series* that share enough in common to consider them related and under a single family denomination.

SoC series
A number of different *SoCs* that share similar characteristics and features, and that the vendor typically names and markets together.

software component
A software component is a self-contained, modular, and replaceable part of the Zephyr source code. A driver, a subsystem or an applications are all examples of software components present in Zephyr.

subsystem
A subsystem refers to a logically distinct part of the operating system that handles specific functionality or provides certain services.

system power state
System power states describe the power consumption of the system as a whole. System power states are represented by `pm_state`.

variant
In the context of *board qualifiers*, a variant designates a particular type or configuration of a build for a combination of *SoC* and *CPU cluster*. Common uses of the variant concept include introducing both secure and non-secure builds for platforms with Trusted Execution Environment support, or selecting the type of RAM used in a build.

west
A multi-repo meta-tool developed for the Zephyr project. See *West (Zephyr's meta-tool)*.

west installation
An obsolete term for a *west workspace* used prior to west 0.7.

west manifest
A YAML file, usually named `west.yml`, which describes projects, or the Git repositories which make up a *west workspace*, along with additional metadata. See *Basics* for general information and *West Manifests* for details.

west manifest repository
The Git repository in a *west workspace* which contains the *west manifest*. Its location is given by the *manifest.path configuration option*. See *Basics*.

west project
Each of the entries in a *west manifest*, which describe a Git repository that will be cloned and managed by west when working with the corresponding *west manifest repository*. Note that a west project is different from a *zephyr module*, although many projects are also modules. See *Projects* for additional information.

west workspace
A folder on your system with a `.west` subdirectory and a *west manifest repository* in it. You clone the Zephyr source code, as well as that of its *west projects* onto your system by creating a west workspace using the `west init` command. See *Basics*.

XIP
(eXecute In Place) a method of executing programs directly from long term storage rather than copying it into RAM, saving writable memory for dynamic data and not the static program code.

zephyr module
A Git repository containing a `zephyr/module.yml` file, used by the Zephyr build system to integrate the source code and configuration files of the module into a regular Zephyr build. Zephyr modules may be west projects, but they do not have to. See *Modules (External projects)* for additional details.

Bibliography

[a] Real Time Counter (RTC)

[b] Programmable Peripheral Interconnect (PPI)

[c] Distributed Programmable Peripheral Interconnect (DPPI)

[d] Software Interrupt (SWI)

[e] Random Number Generator (RNG)

[f] AES Electronic Codebook Mode Encryption (ECB)

[g] Cipher Block Chaining (CBC) - Message Authentication Code with Counter Mode encryption (CCM)

[h] Accelerated Address Resolver (AAR)

[i] General Purpose Input Output (GPIO)

[j] GPIO tasks and events (GPIOTE)

[k] Temperature sensor (TEMP)

[l] Universal Asynchronous Receiver Transmitter (UART)

[m] Interprocess Communication peripheral (IPC)

[th-imboot] Must boot with an immutable bootloader.

[th-authrepl] Application image shall only be replaced with an authorized image.

[th-timely-update] Application updates shall be done in a timely manner.

[th-atomic-update] Application updates shall be atomic.

[th-root-certs] TLS must have a list of trusted root certificates.

[th-root-check] TLS must verify root certificate from server is valid.

[th-secret-storage] There must be a mechanism to securely store client secrets. The least amount of code necessary shall have access to these secrets.

[th-time] System must have moderately accurate notion of the current date/time.

[th-conf] The system must receive, and keep configuration data.

[th-logs] The system must log security-related events, and either store them locally, or send to a service.

[th-all-tls] All communications with the cloud service shall use TLS.

[th-tls-ciphers] TLS shall be configured to allow only generally agreed cipher suites (including forward secrecy).

[th-tls-client-auth] The device shall authenticate itself with the cloud provider using one of the methods described.

[th-entropy] The TLS layer shall use a modern, accepted cryptographic random-bit generator seeded by an entropy source within the SoC.

[th-initial-provision] The device shall have a per-device secret loaded before deployment.

[th-initial-secret] The initial secret shall be securely maintained, and destroyed in any external location as soon as the device is provisioned.

[th-reprovision] Reprovisioning a device shall be done securely.

[th-destruction] Upon decommissioning, the device secret shall be rendered ineffective.

Python Module Index

r

runners.core, 235

Index

Non-alphabetical

`:app:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2248
`:attach-args:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:board:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2248
`:build-args:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:build-dir:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:build-dir-fmt:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:categories:` (*directive option*)
    `zephyr:code-sample-listing` (*directive*), 2254
`:cd-into:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2248
`:compact:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:conf:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:debug-args:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:debugserver-args:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:flash-args:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:gen-args:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:generator:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2248
`:glob:` (*directive option*)
    `zephyr:code-sample-category` (*directive*), 2253
`:goals:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:host-os:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2248
`:live-search:` (*directive option*)
    `zephyr:code-sample-listing` (*directive*), 2254
`:maybe-skip-config:` (*directive option*)
    `zephyr-app-commands` (*directive*), 2249
`:name:` (*directive option*)
    `zephyr:code-sample` (*directive*), 2252
    `zephyr:code-sample-category` (*directive*), 2253
`:project:` (*directive option*)
    `doxygengroup` (*directive*), 2251

:relevant-api: (*directive option*)
zephyr:code-sample (*directive*), 2252
:shield: (*directive option*)
zephyr-app-commands (*directive*), 2248
:show-listing: (*directive option*)
zephyr:code-sample-category (*directive*), 2253
:snippets: (*directive option*)
zephyr-app-commands (*directive*), 2249
:tool: (*directive option*)
zephyr-app-commands (*directive*), 2248
:west-args: (*directive option*)
zephyr-app-commands (*directive*), 2249
:zephyr-app: (*directive option*)
zephyr-app-commands (*directive*), 2248

A

ACTIVE_BOARD_REVISION
variable, 1554, **1591**, 2150
add_llext_target
command, 1467, 1594
add_parser() (*runners.core.ZephyrBinaryRunner class method*), 239
add_subdirectory_ifdef
command, **1557**
API, **2547**
app
variable, **1545**
APP_BUILD_VERSION
variable, **1591**, 1593, 1981
APP_DIR
variable, 1585, **1591**, 1595
app_set_runner_args
command, 1553
application, **2547**
application image, **2547**
APPLICATION_CONFIG_DIR
variable, 1544, 1585, **1591**, 1594
ARCH
variable, 1543, **1591**
arch
module, **1543**, 1591, 1592
ARCH_DIR
variable, 1543, 1544, **1591**
ARCH_ROOT
variable, 1543, 1544, 1585, **1591**
architecture, **2547**
ARCMWDT_TOOLCHAIN_PATH, 368
args_from_previous_runner() (*runners.core.ZephyrBinaryRunner class method*), 239
ARM_PRODUCT_DEF, 365
ARMCLANG_TOOLCHAIN_PATH, 365
ARMLMD_LICENSE_FILE, 365
AUTOCONF_H
variable, **1581**

B

basic_settings
module, **1545**
bin_file (*runners.core.RunnerConfig attribute*), 237
BOARD, 225

variable, 1543, 1544, 1582, **1592**
board, **2547**
board configuration, **2547**
board name, **2547**
board qualifiers, **2547**
board revision, **2548**
board target, **2548**
BOARD_DIR
variable, 1544, **1592**
board_dir (*runners.core.RunnerConfig attribute*), 237
BOARD_QUALIFIERS
variable, 1544, **1592**
BOARD_REVISION
variable, 1544, **1592**
BOARD_ROOT
variable, 1544, 1585, **1592**
board_runner_args
command, 1553
board_set_debugger_ifnset
command, 1553
board_set_flasher_ifnset
command, 1553
board_set_runner
command, 1553
board_set_runner_ifnset
command, 1553
board_set_xxx_ifnset
command, 1552
boards
module, **1543**, 1591, 1592, 1594
BSIM_COMPONENTS_PATH
variable, **1592**
BSIM_OUT_PATH
variable, **1592**
build_conf (*runners.core.ZephyrBinaryRunner property*), 239
build_dir (*runners.core.RunnerConfig attribute*), 237
BUILD_VERSION
variable, 1591, **1592**
BuildConfiguration (*class in runners.core*), 235

C

c:data (*role*), 2244
c:enum (*role*), 2244
c:enumerator (*role*), 2244
c:func (*role*), 2244
c:group (*role*), 2251
c:macro (*role*), 2244
c:member (*role*), 2244
c:struct (*role*), 2244
c:type (*role*), 2244
c:union (*role*), 2244
c:var (*role*), 2244
call() (*runners.core.ZephyrBinaryRunner method*), 239
capabilities() (*runners.core.ZephyrBinaryRunner class method*), 239
cfg (*runners.core.ZephyrBinaryRunner attribute*), 239
check_call() (*runners.core.ZephyrBinaryRunner method*), 239
check_call_ignore_sigint() (*runners.core.ZephyrBinaryRunner method*), 239
check_output() (*runners.core.ZephyrBinaryRunner method*), 239

close() (*twister_harness.DeviceAdapter method*), 271
cmake:command (*role*), 2245
cmake:module (*role*), 2245
cmake:variable (*role*), 2245
CMAKE_C_COMPILER
    variable, 2245
code_data_relocation_target
    variable, **1545**
command
    add_llext_target, 1467, 1594
    add_subdirectory_ifdef, **1557**
    app_set_runner_args, 1553
    board_runner_args, 1553
    board_set_debugger_ifnset, 1553
    board_set_flasher_ifnset, 1553
    board_set_runner, 1553
    board_set_runner_ifnset, 1553
    board_set_xxx_ifnset, 1552
    command:add_custom_command, 1581
    command:add_custom_target, 1467
    command:add_dependencies, 1551, 1971
    command:add_subdirectory, 1584
    command:cmake_parse_arguments, 15771579
    command:configure_file, 1560
    command:file, 1559
    command:get_target_property, 1580
    command:include, 1584
    command:list, 1560
    command:string, 1559
    command:target_compile_definitions, 1548, 1579
    command:target_compile_features, 1579
    command:target_compile_options, 1548, 1579
    command:target_include_directories, 1547, 1548, 1580
    command:target_ld_options, 1548
    command:target_link_libraries, 1548
    command:target_link_options, 1580
    command:target_sources, 1547, 2245
    command-add_executable, 1562
    dt_node_exists, 1566
    dt_prop, 1566
    externalzephyrproject_add, 1583, 1593, 1967, 1971, 1977
    externalzephyrvariantproject_add, 1968
    find_package, 1545, 1553
    generate_inc_file_for_target, 1552
    git_describe, **1587**
    llext_compile_definitions, 1467
    llext_compile_features, 1467
    llext_compile_options, 1467
    llext_include_directories, 1467
    llext_link_options, 1467
    native_simulator_set_child_images, 1585
    sysbuild_add_dependencies, 1584, 1971
    yaml_context, **1588**
    yaml_create, **1589**
    yaml_get, **1589**
    yaml_length, **1589**
    yaml_load, **1589**, 2245
    yaml_remove, **1590**

yaml_save, 1589, **1590**
yaml_set, **1590**
zephyr_get, 1561
zephyr_interface_library_named, 1545
zephyr_libc_link_libraries, 1591
zephyr_library, 1545, 1549
zephyr_library_link_libraries, 1551
zephyr_library_named, 1549
zephyr_linker_memory, 1571
zephyr_linker_section, 1574, 1576, 1577
zephyr_linker_section_configure, 1574
zephyr_set, 1561
command:add_custom_command
command, 1581
command:add_custom_target
command, 1467
command:add_dependencies
command, 1551, 1971
command:add_subdirectory
command, 1584
command:cmake_parse_arguments
command, 15771579
command:configure_file
command, 1560
command:file
command, 1559
command:get_target_property
command, 1580
command:include
command, 1584
command:list
command, 1560
command:string
command, 1559
command:target_compile_definitions
command, 1548, 1579
command:target_compile_features
command, 1579
command:target_compile_options
command, 1548, 1579
command:target_include_directories
command, 1547, 1548, 1580
command:target_ld_options
command, 1548
command:target_link_libraries
command, 1548
command:target_link_options
command, 1580
command:target_sources
command, 1547, 2245
command-add_executable
command, 1562
Common Vulnerabilities and Exposures
CVE 2017-14199, 2333
CVE 2017-14201, 2333
CVE 2017-14202, 2333
CVE 2019-9506, 2334
CVE 2020-10019, 2334

CVE 2020-10021, 2334
CVE 2020-10022, 2334
CVE 2020-10023, 2335
CVE 2020-10024, 2335
CVE 2020-10027, 2335
CVE 2020-10028, 2335
CVE 2020-10058, 2336
CVE 2020-10059, 2336
CVE 2020-10060, 2336
CVE 2020-10061, 2337
CVE 2020-10062, 2337
CVE 2020-10063, 2337
CVE 2020-10064, 2337
CVE 2020-10065, 2338
CVE 2020-10066, 2338
CVE 2020-10067, 2338
CVE 2020-10068, 2338
CVE 2020-10069, 2338
CVE 2020-10070, 2339
CVE 2020-10071, 2339
CVE 2020-10072, 2339
CVE 2020-10136, 2339
CVE 2020-13598, 2340
CVE 2020-13599, 2340
CVE 2020-13600, 2340
CVE 2020-13601, 2340
CVE 2020-13602, 2340
CVE 2020-13603, 2340
CVE 2021-3319, 2341
CVE 2021-3320, 2341
CVE 2021-3321, 2341
CVE 2021-3323, 2341
CVE 2021-3430, 2341
CVE 2021-3431, 2342
CVE 2021-3432, 2342
CVE 2021-3433, 2342
CVE 2021-3434, 2342
CVE 2021-3435, 2342
CVE 2021-3436, 2343
CVE 2021-3454, 2343
CVE 2021-3455, 2343
CVE 2021-3510, 2343
CVE 2021-3581, 2344
CVE 2021-3625, 2344
CVE 2021-3835, 2344
CVE 2021-3861, 2344
CVE 2021-3966, 2344
CVE 2022-0553, 2345
CVE 2022-1041, 2345
CVE 2022-1042, 2345
CVE 2022-1841, 2345
CVE 2022-2741, 2345
CVE 2022-2993, 2346
CVE 2022-3806, 2346
CVE 2023-0396, 2346
CVE 2023-0397, 2346
CVE 2023-0779, 2346
CVE 2023-1901, 2346

CVE 2023-1902, 2347
CVE 2023-3725, 2347
CVE 2023-4257, 2347
CVE 2023-4258, 2347
CVE 2023-4259, 2347
CVE 2023-4260, 2348
CVE 2023-4262, 2348
CVE 2023-4263, 2348
CVE 2023-4264, 2348
CVE 2023-4265, 2348
CVE 2023-4424, 2348
CVE 2023-5055, 2349
CVE 2023-5139, 2349
CVE 2023-5184, 2349
CVE 2023-5563, 2349
CVE 2023-5753, 2349
CVE 2023-5779, 2350
CVE 2023-6249, 2350
CVE 2023-6749, 2350
CVE 2023-6881, 2350
CVE 2023-7060, 2350
CVE 2024-1638, 2351
CVE 2024-3077, 2351
CVE 2024-3332, 2351
CVE 2024-4785, 2351
CVE 2024-5754, 2351
CVE 2024-5931, 2352
CVE 2024-6135, 2352
CVE 2024-6137, 2352
CVE 2024-6258, 2352
CVE 2024-6259, 2352
CVE 2024-6442, 2352
CVE 2024-6443, 2353
CVE 2024-6444, 2353
CVE 2024-8798, 2353
CVE 2024-10395, 2353
CVE 2024-11263, 2353
CVE 2025-1673, 2354
CVE 2025-1674, 2354
CVE 2025-1675, 2354
CVE 2025-2962, 2354
CVE 2025-7403, 2355
CVE 2025-9408, 2355
CVE 2025-9557, 2356
CVE 2025-9558, 2356
CVE 2025-10456, 2355
CVE 2025-10457, 2355
CVE 2025-10458, 2355
CVE 2025-12035, 2356
CVE 2025-12890, 2357
CVE 2025-12899, 2357
CVE 2026-0849, 2357
CVE 2026-1677, 2357
CVE 2026-1678, 2358
CVE 2026-1679, 2358
CVE 2026-1681, 2358
CVE 2026-2411, 2433
CVE 2026-4179, 2358

CVE 2026-5066, 2359
CVE 2026-5067, 2359
CVE 2026-5068, 2359
CVE 2026-5071, 2360
CVE 2026-5072, 2360
CVE 2026-5589, 2360
CVE 2026-5590, 2361
CVE 2026-6682, 2426
CVE 2026-6683, 2426
CVE 2026-6685, 2426
CVE 2026-6686, 2427
CVE 2026-6687, 2427
CVE 2026-7007, 2400
CVE 2026-7656, 2407
CVE 2026-8023, 2401
CVE 2026-8718, 2361
CVE 2026-9263, 2361
CVE 2026-9728, 2401
CVE 2026-9771, 2402
CVE 2026-10593, 2362
CVE 2026-10634, 2362
CVE 2026-10635, 2363
CVE 2026-10636, 2363
CVE 2026-10637, 2364
CVE 2026-10638, 2364
CVE 2026-10639, 2365
CVE 2026-10640, 2366
CVE 2026-10641, 2366
CVE 2026-10642, 2367
CVE 2026-10643, 2367
CVE 2026-10644, 2368
CVE 2026-10645, 2368
CVE 2026-10646, 2369
CVE 2026-10647, 2370
CVE 2026-10648, 2370
CVE 2026-10651, 2371
CVE 2026-10652, 2371
CVE 2026-10653, 2372
CVE 2026-10654, 2372
CVE 2026-10655, 2373
CVE 2026-10656, 2373
CVE 2026-10657, 2374
CVE 2026-10658, 2374
CVE 2026-10659, 2375
CVE 2026-10660, 2375
CVE 2026-10663, 2376
CVE 2026-10664, 2377
CVE 2026-10665, 2377
CVE 2026-10666, 2408
CVE 2026-10667, 2377
CVE 2026-10668, 2378
CVE 2026-10669, 2379
CVE 2026-10670, 2379
CVE 2026-10671, 2380
CVE 2026-10672, 2380
CVE 2026-10673, 2408
CVE 2026-10674, 2381
CVE 2026-10675, 2381

CVE 2026-10677, 2382
CVE 2026-10678, 2383
CVE 2026-10679, 2383
CVE 2026-10680, 2384
CVE 2026-10681, 2384
CVE 2026-10682, 2385
CVE 2026-10683, 2385
CVE 2026-10684, 2386
CVE 2026-10685, 2386
CVE 2026-10686, 2387
CVE 2026-10687, 2388
CVE 2026-10772, 2388
CVE 2026-10773, 2388
CVE 2026-10774, 2388
CVE 2026-10848, 2389
CVE 2026-10849, 2389
CVE 2026-11368, 2390
CVE 2026-11742, 2391
CVE 2026-11743, 2391
CVE 2026-11809, 2392
CVE 2026-11810, 2392
CVE 2026-11811, 2393
CVE 2026-11812, 2394
CVE 2026-11893, 2394
CVE 2026-11894, 2395
CVE 2026-11985, 2395
CVE 2026-12051, 2396
CVE 2026-12052, 2396
CVE 2026-12232, 2397
CVE 2026-12233, 2398
CVE 2026-12234, 2398
CVE 2026-12235, 2399
CVE 2026-12236, 2399
CVE 2026-12363, 2402
CVE 2026-12364, 2403
CVE 2026-12365, 2403
CVE 2026-12366, 2404
CVE 2026-12519, 2405
CVE 2026-12520, 2405
CVE 2026-12521, 2406
CVE 2026-12522, 2406
CVE 2026-12629, 2409
CVE 2026-12630, 2409
CVE 2026-12631, 2410
CVE 2026-12632, 2410
CVE 2026-12633, 2411
CVE 2026-12634, 2412
CVE 2026-12999, 2412
CVE 2026-13212, 2413
CVE 2026-13213, 2413
CVE 2026-13214, 2414
CVE 2026-13215, 2415
CVE 2026-13216, 2415
CVE 2026-13217, 2416
CVE 2026-13343, 2416
CVE 2026-13351, 2417
CVE 2026-13478, 2417
CVE 2026-13479, 2418

CVE 2026-13480, 2419
CVE 2026-13481, 2419
CVE 2026-13734, 2420
CVE 2026-13735, 2421
CVE 2026-14366, 2421
CVE 2026-14367, 2422
CVE 2026-14368, 2422
CVE 2026-14696, 2423
CVE 2026-14697, 2424
CVE 2026-14986, 2424
CVE 2026-15460, 2424
CVE 2026-15461, 2425
CVE 2026-15890, 2427
CVE 2026-15891, 2428
CVE 2026-15892, 2428
CVE 2026-15893, 2429
CVE 2026-15894, 2429
CVE 2026-15923, 2429
CVE 2026-15924, 2430
CVE 2026-16147, 2431
CVE 2026-16148, 2431
CVE 2026-16511, 2432
CVE 2026-16512, 2432
CVE 2026-16513, 2432
CVE 2026-16514, 2432
CVE 2026-16515, 2432
CVE 2026-17050, 2432
CVE 2026-17051, 2432
CVE 2026-17052, 2432
CVE 2026-17053, 2432
CVE 2026-17054, 2432
CVE 2026-18413, 2433
CVE 2026-18414, 2433
CVE 2026-18415, 2433
CVE 2026-18416, 2433
CVE 2026-18417, 2434
CVE 2026-18418, 2434
CVE 2026-18746, 2434
CVE 2026-18747, 2434
CVE 2026-18748, 2434
CVE 2026-19184, 2434
CVE 2026-19185, 2434
CVE 2026-19186, 2434
CVE 2026-19569, 2434
CVE 2026-19570, 2434
CVE 2026-19571, 2434
CVE 2026-19574, 2434
CVE 2026-19575, 2434
CVE 2026-19576, 2434
CVE 2026-19577, 2435
CVE 2026-19595, 2435
CVE 2026-19669, 2435
CVE 2026-19673, 2435
CVE 2026-19676, 2435
CVE 2026-19735, 2435
CVE 2026-19736, 2435
CVE 2026-19737, 2435
CVE 2026-19738, 2435

CVE 2026-19739, 2435
CVE 2026-19740, 2435
CVE 2026-19741, 2435
CVE 2026-19742, 2435
CVE 2026-19809, 2435
CVE 2026-19935, 2436
CVE 2026-19936, 2436
CVE 2026-19937, 2436
CVE 2026-19938, 2436
CVE 2026-19939, 2436
CVE 2026-19940, 2436
CVE 2026-19947, 2436
CVE 2026-75083, 2436
CVE 2026-75084, 2436
CVE 2026-75085, 2436
CVE 2026-76787, 2436
CVE 2026-76788, 2436
CVE 2026-77684, 2436
CVE 2026-77685, 2436
CVE 2026-78116, 2437
CVE 2026-78117, 2437
CVE 2026-78118, 2437
CVE 2026-78119, 2437
CVE 2026-79976, 2437
CVE 2026-79977, 2437
CVE 2026-79978, 2437
CVE 2026-79979, 2437
CVE 2026-79980, 2437
CVE 2026-79981, 2437
CVE 2026-79982, 2437
CVE 2026-81038, 2437
CVE 2026-81039, 2437
CVE 2026-82388, 2437
CVE 2026-82389, 2438
CVE 2026-82390, 2438
CVE 2026-82391, 2438
CVE 2026-82961, 2438
CVE 2026-82962, 2438
CVE 2026-85032, 2438
CVE 2026-85033, 2438
CVE 2026-85034, 2438
CVE 2026-85035, 2438
CVE 2026-85036, 2438
CVE 2026-86086, 2438
CVE 2026-86092, 2438
CVE 2026-87038, 2438
CVE 2026-87039, 2438
CVE 2026-87040, 2439
CVE 2026-87041, 2439
CVE 2026-87042, 2439
CVE 2026-87043, 2439
CVE 2026-87044, 2439
CVE 2026-87045, 2439
CVE 2026-90585, 2439
CVE 2026-90586, 2439
CVE 2026-90587, 2439
CVE 2026-90588, 2439
CVE 2026-90589, 2439

CVE 2026-90590, 2439
CVE 2026-90591, 2439
CVE 2026-90832, 2439
CVE 2026-90833, 2440
CVE 2026-90834, 2440
CVE 2026-91007, 2440
CONF_FILE
variable, 1544, 1581, 1583, **1593**
CONFIG_COMPILER_RT_RTLIB=y, 366
CONFIG_LIBGCC_RTLIB=y, 366
CONFIG_LLVM_USE_LD=y, 366
CONFIG_LLVM_USE_LLD=y, 366
configuration_files
module, **1544**, 1581, 1591, 1593
COVERITY_OUTPUT_DIR, 354
CPU cluster, **2548**
CPU core, **2548**
create() (*runners.core.ZephyrBinaryRunner class method*), 239

D

DEFAULT_IMAGE
variable, **1593**
DeprecatedAction (*class in runners.core*), 235
dev_id_help() (*runners.core.ZephyrBinaryRunner class method*), 240
device runtime power management, **2548**
DeviceAdapter (*class in twister_harness*), 270
DISPLAY_TEST_DIR, 304
do_add_parser() (*runners.core.ZephyrBinaryRunner class method*), 240
do_create() (*runners.core.ZephyrBinaryRunner class method*), 240
do_run() (*runners.core.ZephyrBinaryRunner method*), 240
doc
module, **1586**
DOTCONFIG
variable, **1581**
doxygengroup (*directive*), 2251
:project: (*directive option*), 2251
dry_run (*runners.core.ZephyrBinaryRunner attribute*), 240
dt_node_exists
command, 1566
dt_prop
command, 1566
DTC_OVERLAY_FILE
variable, 1544, **1593**
dtcompatible (*role*), 2252
DTS_EXTRA_CPPFLAGS
variable, 1544, **1593**

E

edt (*runners.core.BuildConfiguration property*), 235
elf_file (*runners.core.RunnerConfig attribute*), 237
ensure_output() (*runners.core.ZephyrBinaryRunner method*), 240
environment variable
ARCMWDT_TOOLCHAIN_PATH, 368
ARM_PRODUCT_DEF, 365
ARMCLANG_TOOLCHAIN_PATH, 365
ARMLMD_LICENSE_FILE, 365
BOARD, 23, 225
CONF_FILE, 23

CONFIG_COMPILER_RT_RTLIB=y, 366
CONFIG_LIBGCC_RTLIB=y, 366
CONFIG_LLVM_USE_LD=y, 366
CONFIG_LLVM_USE_LLD=y, 366
COVERITY_OUTPUT_DIR, 354
DISPLAY_TEST_DIR, 304
EXTRA_ZEPHYR_MODULES, 23
GNUARMEMB_TOOLCHAIN_PATH, 369
HEXAGON_TOOLCHAIN_PATH, 370
IAR_LMS_BEARER_TOKEN, 370
IAR_TOOLCHAIN_PATH, 370
LLVM_TOOLCHAIN_PATH, 366, 370, 374
METAWARE_ROOT, 368
PATH, 7, 13, 22, 24, 126, 167, 172, 354, 359, 374
QEMU_EXTRA_FLAGS, 43
SHIELD, 23
T32_DIR, 129
{TOOLCHAIN}_TOOLCHAIN_PATH, 17, 23
TOOLCHAIN_VAR_{normalized_board_target}, 367
TOOLCHAIN_VER, 367
WIFI_TEST_CERTS_DIR, 699
XC32_TOOLCHAIN_PATH, 371
XC_PACK_DIR, 371
XCC_NO_G_FLAG, 368
XTENSA_CORE, 367
XTENSA_CORE_{normalized_board_target}, 367
XTENSA_TOOLCHAIN_PATH, 367
ZEPHYR_BASE, 22, 23, 173, 174, 219, 388, 1248, 1592, 1956, 2240
ZEPHYR_BOARD_ALIASES, 19, 23, 1544
ZEPHYR_MODULES, 23
ZEPHYR_SCA_VARIANT, 350
ZEPHYR_SDK_INSTALL_DIR, 16, 17, 23, 362
ZEPHYR_TOOLCHAIN_VARIANT, 17, 23, 281, 283, 287, 362, 365373, 1561
envvar
envvar:CMAKE_BUILD_PARALLEL_LEVEL, 1963
envvar:VERBOSE, 1963
envvar:CMAKE_BUILD_PARALLEL_LEVEL
envvar, 1963
envvar:VERBOSE
envvar, 1963
exe_file (*runners.core.RunnerConfig attribute*), 237
exec_command() (*twister_harness.Shell method*), 272
extensions
module, **1546**, 1586, 2245
externalzephyrproject_add
command, 1583, 1593, 1967, 1971, 1977
externalzephyrvariantproject_add
command, 1968
extload_help() (*runners.core.ZephyrBinaryRunner class method*), 240
EXTRA_CONF_FILE
variable, 1544, 1581, 1583, **1593**
EXTRA_DTC_OVERLAY_FILE
variable, 1544, 1583, **1593**
EXTRA_KCONFIG_TARGET_COMMAND_FOR_<target>
variable, **1581**
EXTRA_KCONFIG_TARGETS
variable, **1581**

F

file (*runners.core.RunnerConfig attribute*), 237
file_type (*runners.core.RunnerConfig attribute*), 237
FileType (*class in runners.core*), 235
find_package
    command, 1545, 1553
flash_address_from_build_conf() (*runners.core.ZephyrBinaryRunner static method*), 240

G

gdb (*runners.core.RunnerConfig attribute*), 237
gdb_ex_args() (*runners.core.ZephyrBinaryRunner static method*), 240
generate_inc_file_for_target
    command, 1552
GENERATED_APP_SOURCE_FILES
    variable, **1545**
GENERATED_KERNEL_OBJECT_FILES
    variable, **1545**
GENERATED_KERNEL_SOURCE_FILES
    variable, **1545**
get_filtered_output() (*twister_harness.Shell method*), 272
get_flash_address() (*runners.core.ZephyrBinaryRunner static method*), 240
get_rtt_address() (*runners.core.ZephyrBinaryRunner method*), 240
get_runners() (*runners.core.ZephyrBinaryRunner static method*), 240
get_unused_ports() (*runners.core.NetworkPortHelper method*), 235
getboolean() (*runners.core.BuildConfiguration method*), 235
git
    module, **1587**
git_describe
    command, **1587**
github (*role*), 2250
GNUARMEMB_TOOLCHAIN_PATH, 369

H

hex_file (*runners.core.RunnerConfig attribute*), 237
HEXAGON_TOOLCHAIN_PATH, 370

I

IAR_LMS_BEARER_TOKEN, 370
IAR_TOOLCHAIN_PATH, 370
idle thread, **2548**
IDT, **2548**
internal API, **2548**
ISR, **2548**

K

kconfig
    module, **1581**
kconfig:option (*role*), 2251
kconfig:option-regex (*role*), 2251
KCONFIG_ROOT
    variable, **1581**
kernel, **2548**
    module, **1545**

L

launch() (*twister_harness.DeviceAdapter method*), 270
LIBC_LINK_LIBRARIES
    target property, 1548, **1591**

LLEXT_APPEND_FLAGS
    variable, 1580, **1593**
llext_compile_definitions
    command, 1467
llext_compile_features
    command, 1467
llext_compile_options
    command, 1467
llext_include_directories
    command, 1467
llext_link_options
    command, 1467
LLEXT_REMOVE_FLAGS
    variable, 1580, **1594**
LLVM_TOOLCHAIN_PATH, 366, 370, 374
logger (*runners.core.ZephyrBinaryRunner attribute*), 241

M

manual
    manual:cmake-commands(7), 1549
    manual:cmake-presets(7), 1979
manual:cmake-commands(7)
    manual, 1549
manual:cmake-presets(7)
    manual, 1979
METAWARE_ROOT, 368
MissingProgram, 235
module
    arch, **1543**, 1591, 1592
    basic_settings, **1545**
    boards, **1543**, 1591, 1592, 1594
    configuration_files, **1544**, 1581, 1591, 1593
    doc, **1586**
    extensions, **1546**, 1586, 2245
    git, **1587**
    kconfig, **1581**
    kernel, **1545**
    module:ExternalProject, 1972
    native_simulator_sb_extensions, **1585**
    python, **1586**, 1594
    runners.core, 235
    soc, **1544**
    sysbuild_extensions, **1581**
    sysbuild_kconfig, **1584**, 1594, 1595
    sysbuild_root, **1585**, 1594, 1595
    version, 1591, 1593
    west, **1586**, 1595
    yaml, **1587**
module:ExternalProject
    module, 1972
MODULE_EXT_ROOT
    variable, 1585, **1594**
module_file (*role*), 2250
mot_file (*runners.core.RunnerConfig attribute*), 237

N

name() (*runners.core.ZephyrBinaryRunner class method*), 241
native_simulator_sb_extensions

module, **1585**
native_simulator_set_child_images
command, 1585
NetworkPortHelper (*class in runners.core*), 235
normalize_chip() (*runners.core.ZephyrBinaryRunner static method*), 241
NORMALIZED_BOARD_QUALIFIERS
variable, 1544, **1594**
NORMALIZED_BOARD_TARGET
variable, 1544, **1594**

O

openocd (*runners.core.RunnerConfig attribute*), 237
openocd_search (*runners.core.RunnerConfig attribute*), 237

P

PATH, 7, 13, 22, 126, 167, 172, 354, 359, 374
popen_ignore_int() (*runners.core.ZephyrBinaryRunner method*), 241
power domain, **2548**
power gating, **2548**
private API, **2549**
public API, **2549**
python
module, **1586**, 1594
PYTHON_EXECUTABLE
variable, 1586, **1594**
PYTHON_MINIMUM_REQUIRED
variable, 1586, **1594**

Q

QEMU_EXTRA_FLAGS, 43

R

readline() (*twister_harness.DeviceAdapter method*), 271
readlines() (*twister_harness.DeviceAdapter method*), 271
readlines_until() (*twister_harness.DeviceAdapter method*), 271
require() (*runners.core.ZephyrBinaryRunner static method*), 241
resolve_port_by_chip() (*runners.core.ZephyrBinaryRunner static method*), 241
RFC
RFC 768, 608
RFC 791, 608
RFC 826, 613
RFC 959, 609, 719
RFC 1035, 608, 664
RFC 1191, 608
RFC 1332, 693
RFC 1350, 609
RFC 1661, 609, 693
RFC 1662, 693
RFC 1928, 609, 674
RFC 2018, 608
RFC 2131, 608
RFC 2236, 608
RFC 2616, 608, 721
RFC 3376, 608
RFC 3550, 609, 753
RFC 3810, 607
RFC 3927, 608
RFC 4158, 2326

RFC 4251, 609
RFC 4254, 609
RFC 4330, 674, 2326
RFC 4795, 608, 664
RFC 4861, 607
RFC 4861 Section 6.2.6, 775
RFC 4862, 607
RFC 4944, 607
RFC 5072, 693
RFC 5227, 608
RFC 5234, 1634
RFC 5452, 653
RFC 5869, 1366
RFC 5905, 609
RFC 6125, 2326
RFC 6206, 675
RFC 6282, 613
RFC 6455, 608, 675
RFC 6690, 704
RFC 6762, 608, 652, 664
RFC 6763, 608, 664
RFC 7252, 608, 704, 718
RFC 7405, 1634
RFC 7519, 1361, 2326
RFC 7641, 608, 704
RFC 7959, 608, 704, 708, 709
RFC 8106, 607
RFC 8200, 607
RFC 8201, 608
RFC 8323, 608, 707
RFC 8415, 607
RFC 8415 Section 18.2.7, 775
RFC 8536 Section 2, 590, 592
RFC 8613, 608, 704, 715, 717, 718
RFC 8899, 608, 666
RFC 8981, 608
RFC 9000, 608, 756, 757, 761, 762
RFC 9001, 608, 756, 760
RFC 9113, 608, 721
RFC 9114, 608, 721
RFC 9293, 608
RFC 9562, 1372
RFC 9868, 608
RFC 9869, 608
rtt_address (*runners.core.RunnerConfig attribute*), 237
run() (*runners.core.ZephyrBinaryRunner method*), 241
run_server_and_client() (*runners.core.ZephyrBinaryRunner method*), 241
run_telnet_client() (*runners.core.ZephyrBinaryRunner method*), 242
RunnerCaps (*class in runners.core*), 236
RunnerConfig (*class in runners.core*), 237
runners.core
module, 235

S

SB_APPLICATION_CONFIG_DIR
variable, 1585, **1594**
SB_CONF_FILE
variable, 1585, **1594**, 1595

SB_EXTRA_CONF_FILE
    variable, 1585, **1595**
SB_SNIPPET
    variable, **1595**
SCA_ROOT
    variable, 1585, **1595**
Shell (*class in twister_harness*), 272
SNIPPET_ROOT
    variable, 1585, **1595**
SoC, **2549**
soc
    module, **1544**
SoC family, **2549**
SoC series, **2549**
SOC_ROOT
    variable, 1585, **1595**
software component, **2549**
sram_address_from_build_conf() (*runners.core.ZephyrBinaryRunner static method*), 242
subsystem, **2549**
sysbuild_add_dependencies
    command, 1584, 1971
sysbuild_conf (*runners.core.ZephyrBinaryRunner property*), 242
sysbuild_extensions
    module, **1581**
sysbuild_kconfig
    module, **1584**, 1594, 1595
sysbuild_root
    module, **1585**, 1594, 1595
SysbuildConfiguration (*class in runners.core*), 237
system power state, **2549**

T

T32_DIR, 129
target property
    LIBC_LINK_LIBRARIES, 1548, **1591**
thread_info_enabled (*runners.core.ZephyrBinaryRunner property*), 242
tool_opt_help() (*runners.core.ZephyrBinaryRunner class method*), 242
{TOOLCHAIN}_TOOLCHAIN_PATH, 17
TOOLCHAIN_VAR_{normalized_board_target}, 367
TOOLCHAIN_VER, 367

U

uf2_file (*runners.core.RunnerConfig attribute*), 237
USE_CCACHE
    variable, **1595**

V

variable
    ACTIVE_BOARD_REVISION, 1554, **1591**, 2150
    app, **1545**
    APP_BUILD_VERSION, **1591**, 1593, 1981
    APP_DIR, 1585, **1591**, 1595
    APPLICATION_CONFIG_DIR, 1544, 1585, **1591**, 1594
    ARCH, 1543, **1591**
    ARCH_DIR, 1543, 1544, **1591**
    ARCH_ROOT, 1543, 1544, 1585, **1591**
    AUTOCONF_H, **1581**
    BOARD, 1543, 1544, 1582, **1592**

BOARD_DIR, 1544, **1592**
BOARD_QUALIFIERS, 1544, **1592**
BOARD_REVISION, 1544, **1592**
BOARD_ROOT, 1544, 1585, **1592**
BSIM_COMPONENTS_PATH, **1592**
BSIM_OUT_PATH, **1592**
BUILD_VERSION, 1591, **1592**
CMAKE_C_COMPILER, 2245
code_data_relocation_target, **1545**
CONF_FILE, 1544, 1581, 1583, **1593**
DEFAULT_IMAGE, **1593**
DOTCONFIG, **1581**
DTC_OVERLAY_FILE, 1544, **1593**
DTS_EXTRA_CPPFLAGS, 1544, **1593**
EXTRA_CONF_FILE, 1544, 1581, 1583, **1593**
EXTRA_DTC_OVERLAY_FILE, 1544, 1583, **1593**
EXTRA_KCONFIG_TARGET_COMMAND_FOR_<target>, **1581**
EXTRA_KCONFIG_TARGETS, **1581**
GENERATED_APP_SOURCE_FILES, **1545**
GENERATED_KERNEL_OBJECT_FILES, **1545**
GENERATED_KERNEL_SOURCE_FILES, **1545**
KCONFIG_ROOT, **1581**
LLEXT_APPEND_FLAGS, 1580, **1593**
LLEXT_REMOVE_FLAGS, 1580, **1594**
MODULE_EXT_ROOT, 1585, **1594**
NORMALIZED_BOARD_QUALIFIERS, 1544, **1594**
NORMALIZED_BOARD_TARGET, 1544, **1594**
PYTHON_EXECUTABLE, 1586, **1594**
PYTHON_MINIMUM_REQUIRED, 1586, **1594**
SB_APPLICATION_CONFIG_DIR, 1585, **1594**
SB_CONF_FILE, 1585, **1594**, 1595
SB_EXTRA_CONF_FILE, 1585, **1595**
SB_SNIPPET, **1595**
SCA_ROOT, 1585, **1595**
SNIPPET_ROOT, 1585, **1595**
SOC_ROOT, 1585, **1595**
USE_CCACHE, **1595**
WEST, 1587, **1595**
WEST_PYTHON, **1595**
WEST_TOPDIR, 1587, **1595**
ZEPHYR_CURRENT_LIBRARY, 1549, 1550, **1595**
ZEPHYR_INTERFACE_LIBS, **1545**
ZEPHYR_LIBS, **1545**
ZEPHYR_TOOLCHAIN_VARIANT, 1561
variant, **2549**
version
module, 1591, 1593

W

wait_for_prompt() (*twister_harness.Shell method*), 272
WEST
variable, 1587, **1595**
west, **2549**
module, **1586**, 1595
west installation, **2549**
west manifest, **2549**
west manifest repository, **2549**
west project, **2549**

west workspace, **2550**
WEST_PYTHON
    variable, **1595**
WEST_TOPDIR
    variable, 1587, **1595**
WIFI_TEST_CERTS_DIR, 699
write() (*twister_harness.DeviceAdapter method*), 271

X

XC32_TOOLCHAIN_PATH, 371
XC_PACK_DIR, 371
XCC_NO_G_FLAG, 368
XIP, **2550**
XTENSA_CORE, 367
XTENSA_CORE_{normalized_board_target}, 367
XTENSA_TOOLCHAIN_PATH, 367

Y

yaml
    module, **1587**
yaml_context
    command, **1588**
yaml_create
    command, **1589**
yaml_get
    command, **1589**
yaml_length
    command, **1589**
yaml_load
    command, **1589**, 2245
yaml_remove
    command, **1590**
yaml_save
    command, 1589, **1590**
yaml_set
    command, **1590**

Z

zephyr module, **2550**
zephyr:board (*directive*), 2254
zephyr:board (*role*), 2254
zephyr:board-catalog (*directive*), 2254
zephyr:board-catalog (*role*), 2254
zephyr:board-supported-hw (*directive*), 2255
zephyr:board-supported-runners (*directive*), 2255
zephyr:code-sample (*directive*), 2252
    :name: (*directive option*), 2252
    :relevant-api: (*directive option*), 2252
zephyr:code-sample (*role*), 2253
zephyr:code-sample-category (*directive*), 2253
    :glob: (*directive option*), 2253
    :name: (*directive option*), 2253
    :show-listing: (*directive option*), 2253
zephyr:code-sample-category (*role*), 2253
zephyr:code-sample-listing (*directive*), 2254
    :categories: (*directive option*), 2254
    :live-search: (*directive option*), 2254
ZEPHYR_BASE, 22, 23, 173, 174, 219, 388, 1248, 1592, 1956, 2240

ZEPHYR_BOARD_ALIASES, 19, 1544
ZEPHYR_CURRENT_LIBRARY
    variable, 1549, 1550, **1595**
zephyr_file (*role*), 2250
zephyr_get
    command, 1561
zephyr_interface_library_named
    command, 1545
ZEPHYR_INTERFACE_LIBS
    variable, **1545**
zephyr_libc_link_libraries
    command, 1591
zephyr_library
    command, 1545, 1549
zephyr_library_link_libraries
    command, 1551
zephyr_library_named
    command, 1549
ZEPHYR_LIBS
    variable, **1545**
zephyr_linker_memory
    command, 1571
zephyr_linker_section
    command, 1574, 1576, 1577
zephyr_linker_section_configure
    command, 1574
zephyr_raw (*role*), 2250
ZEPHYR_SCA_VARIANT, 350
ZEPHYR_SDK_INSTALL_DIR, 16, 17, 362
zephyr_set
    command, 1561
ZEPHYR_TOOLCHAIN_VARIANT, 17, 23, 281, 283, 287, 362, 365373, 1561
    variable, 1561
zephyr-app-commands (*directive*), 2248
    :app: (*directive option*), 2248
    :attach-args: (*directive option*), 2249
    :board: (*directive option*), 2248
    :build-args: (*directive option*), 2249
    :build-dir: (*directive option*), 2249
    :build-dir-fmt: (*directive option*), 2249
    :cd-into: (*directive option*), 2248
    :compact: (*directive option*), 2249
    :conf: (*directive option*), 2249
    :debug-args: (*directive option*), 2249
    :debugserver-args: (*directive option*), 2249
    :flash-args: (*directive option*), 2249
    :gen-args: (*directive option*), 2249
    :generator: (*directive option*), 2248
    :goals: (*directive option*), 2249
    :host-os: (*directive option*), 2248
    :maybe-skip-config: (*directive option*), 2249
    :shield: (*directive option*), 2248
    :snippets: (*directive option*), 2249
    :tool: (*directive option*), 2248
    :west-args: (*directive option*), 2249
    :zephyr-app: (*directive option*), 2248
ZephyrBinaryRunner (*class in runners.core*), 238