MCUXpresso SDK Documentation

Release 25.09.00-pvw1

NXP
Jul 17, 2025

Table of contents

This documentation contains information specific to the frdmmcxw71 board.

Chapter 1

Middleware

1.1 Boot

1.1.1 MCUXpresso SDK : mcuxsdk-middleware-mcuboot_opensource

Overview

This repository is a fork of MCUboot (https://github.com/mcu-tools/mcuboot) for MCUXpresso SDK delivery and it contains the components officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository (mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

Documentation

Overall details can be reviewed here: MCUXpresso SDK Online Documentation

Visit MCUboot - Documentation to review details on the contents in this sub-repo.

Setup

Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest Getting Started with SDK - Detailed Installation Instructions

Contribution

Contributions are not currently accepted. If the intended contribution is not related to NXP specific code, consider contributing directly to the upstream MCUboot project. Once this MCUboot fork is synchronized with the upstream project, such contributions will end up here as well. If the intended contribution is a bugfix or improvement for NXP porting layer or for code added or modified by NXP, please open an issue or contact NXP support.

NXP Fork

This fork of MCUboot contains specific modifications and enhancements for NXP MCUXpresso SDK integration.

See *changelog* for details.

1.1.2 MCUboot

Sim passing Mynewt passing Espressif passing imgtool passing

This is MCUboot version 2.2.0

MCUboot is a secure bootloader for 32-bits microcontrollers. It defines a common infrastructure for the bootloader and the system flash layout on microcontroller systems, and provides a secure bootloader that enables easy software upgrade.

MCUboot is not dependent on any specific operating system and hardware and relies on hardware porting layers from the operating system it works with. Currently, MCUboot works with the following operating systems and SoCs:

- Zephyr
- Apache Mynewt
- Apache NuttX
- RIOT
- Mbed OS
- Espressif
- Cypress/Infineon

RIOT is supported only as a boot target. We will accept any new port contributed by the community once it is good enough.

MCUboot How-tos

See the following pages for instructions on using MCUboot with different operating systems and SoCs:

- Zephyr
- Apache Mynewt
- Apache NuttX
- RIOT
- Mbed OS
- Espressif
- *Cypress/Infineon*

There are also instructions for the *Simulator*.

Roadmap

The issues being planned and worked on are tracked using GitHub issues. To give your input, visit MCUboot GitHub Issues.

Source files

You can find additional documentation on the bootloader in the source files. For more information, use the following links:

- boot/bootutil - The core of the bootloader itself.
- boot/boot_serial - Support for serial upgrade within the bootloader itself.
- boot/zephyr - Port of the bootloader to Zephyr.
- boot/mynewt - Bootloader application for Apache Mynewt.
- boot/nuttx - Bootloader application and port of MCUboot interfaces for Apache NuttX.
- boot/mbed - Port of the bootloader to Mbed OS.
- boot/espressif - Bootloader application and MCUboot port for Espressif SoCs.
- boot/cypress - Bootloader application and MCUboot port for Cypress/Infineon SoCs.
- imgtool - A tool to securely sign firmware images for booting by MCUboot.
- sim - A bootloader simulator for testing and regression.

Joining the project

Developers are welcome!

Use the following links to join or see more about the project:

- Our developer mailing list
- Our Discord channel Get your invite

1.2 Cloud

1.2.1 AWS IoT

Device Shadow Library

AWS IoT Device Shadow library The AWS IoT Device Shadow library enables you to store and retrieve the current state (the "shadow") of every registered device. The device's shadow is a persistent, virtual representation of your device that you can interact with from AWS IoT Core even if the device is offline. The device state is captured as its "shadow" within a JSON document. The device can send commands over MQTT to get, update and delete its latest state as well as receive notifications over MQTT about changes in its state. Each device's shadow is uniquely identified by the name of the corresponding "thing", a representation of a specific device or logical entity on the AWS Cloud. See Managing Devices with AWS IoT for more information on IoT "thing". More details about AWS IoT Device Shadow can be found in AWS IoT documentation. This library is distributed under the *MIT Open Source License*.

Note: From v1.1.0 release onwards, you can used named shadow, a feature of the AWS IoT Device Shadow service that allows you to create multiple shadows for a single IoT device.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library here.

AWS IoT Device Shadow v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.

AWS IoT Device Shadow v1.0.2 source code is part of the FreeRTOS 202012.00 LTS release.

AWS IoT Device Shadow Config File The AWS IoT Device Shadow library exposes configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *shadow_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named `shadow_config.h` can be provided by the user application to the library.

By default, a `shadow_config.h` custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide `SHADOW_DO_NOT_USE_CUSTOM_CONFIG` as a compile time preprocessor macro.

Building the Library The *shadowFilePaths.cmake* file contains the information of all source files and the header include path required to build the AWS IoT Device Shadow library.

As mentioned in the *previous section*, either a custom config file (i.e. `shadow_config.h`) OR the `SHADOW_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the AWS IoT Device Shadow library.

For a CMake example of building the AWS IoT Device Shadow library with the `shadowFilePaths.cmake` file, refer to the `coverity_analysis` library target in *test/CMakeLists.txt* file.

Building Unit Tests

Checkout CMock Submodule By default, the submodules in this repository are configured with `update=none` in *.gitmodules* to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive --test/unit-test/CMock
```

Platform Prerequisites

- For building the library, **CMake 3.13.0** or later and a **C90 compiler**.
- For running unit tests, **Ruby 2.0.0** or later is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build unit tests

1. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described *above*.)
2. Run the *cmake* command: `cmake -S test -B build`

3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples Please refer to the demos of the AWS IoT Device Shadow library in the following locations for reference examples on POSIX and FreeRTOS platforms:

Platform	Location	Transport Interface Implementation (for coreMQTT stack)
POSIX	AWS IoT Device SDK for Embedded C	POSIX sockets for TCP/IP and OpenSSL for TLS stack
FreeRTOS	FreeRTOS/FreeRTOS	FreeRTOS+TCP for TCP/IP and mbedTLS for TLS stack
FreeRTOS	FreeRTOS AWS Reference Integrations	Based on Secure Sockets Abstraction

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of IoT Device Shadow library may differ across repositories.

Generating documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

Device Defender Library

AWS IoT Device Defender Library The Device Defender library enables you to send device metrics to the AWS IoT Device Defender Service. This library also supports custom metrics, a feature that helps you monitor operational health metrics that are unique to your fleet or use case. For example, you can define a new metric to monitor the memory usage or CPU usage on your devices. This library has no dependencies on any additional libraries other than the standard C library, and therefore, can be used with any MQTT client library. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone static code analysis using Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

AWS IoT Device Defender v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.

AWS IoT Device Defender v1.1.0 source code is part of the FreeRTOS 202012.01 LTS release.

AWS IoT Device Defender Client Config File

The AWS IoT Device Defender Client Library exposes build configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *defender_config_defaults.h*. To provide custom values for the configuration macros, a config file named `defender_config.h` can be provided by the application to the library.

By default, a `defender_config.h` config file is required to build the library. To disable this requirement and build the library with default configuration values, provide `DEFENDER_DO_NOT_USE_CUSTOM_CONFIG` as a compile time preprocessor macro.

Thus, the Device Defender client library can be built by either:

- Defining a `defender_config.h` file in the application, and adding it to the include directories list of the library.

OR

- Defining the `DEFENDER_DO_NOT_USE_CUSTOM_CONFIG` preprocessor macro for the library build.

Building the Library

The *defenderFilePaths.cmake* file contains the information of all source files and the header include paths required to build the Device Defender client library.

As mentioned in the previous section, either a custom config file (i.e. `defender_config.h`) or `DEFENDER_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the Device Defender client library.

For a CMake example of building the Device Defender client library with the `defenderFilePaths.cmake` file, refer to the `coverity_analysis` library target in *test/CMakeLists.txt* file.

Building Unit Tests

Platform Prerequisites

- For running unit tests:
  - **C90 compiler** like gcc.
  - **CMake 3.13.0 or later**.
  - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository.
2. Run the *cmake* command: `cmake -S test -B build -DBUILD_CLONE_SUBMODULES=ON`.
3. Run this command to build the library and unit tests: `make -C build all`.
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples The AWS IoT Embedded C-SDK repository contains a demo showing the use of AWS IoT Device Defender Client Library here on a POSIX platform.

Documentation

Existing documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of the AWS IoT Device Defender library may differ across repositories.

Generating documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

Jobs Library

README

AWS IoT Jobs library The AWS IoT Jobs library helps you notify connected IoT devices of a pending Job. A Job can be used to manage your fleet of devices, update firmware and security certificates on your devices, or perform administrative tasks such as restarting devices and performing diagnostics. It interacts with the AWS IoT Jobs service using MQTT, a lightweight publish-subscribe protocol. This library provides a convenience API to compose and recognize the MQTT topic strings used by the Jobs service. The library is written in C compliant with ISO C90 and MISRA C:2012, and is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard . Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity, and validation of memory safety with the CBMC bounded model checker.

See memory requirements for this library *here*.

AWS IoT Jobs v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.

AWS IoT Jobs v1.1.0 source code is part of the FreeRTOS 202012.01 LTS release.

Building the Jobs library A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Given an application in a file named example.c, *gcc* can be used like so:

```
gcc -I source/include example.c source/jobs.c -o example
```

gcc can also produce an object file to be linked later:

```
gcc -I source/include -c source/jobs.c
```

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The test/cbmc/proofs directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference example The AWS IoT Device SDK for Embedded C repository contains a demo using the jobs library on a POSIX platform. https://github.com/aws/aws-iot-device-sdk-embedded-C/tree/main/demos/jobs/jobs_demo_mosquitto

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of the AWS IoT Jobs library may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Building unit tests

Checkout Unity Submodule By default, the submodules in this repository are configured with `update=none` in *.gitmodules* to avoid increasing clone time and disk space usage of other repositories that submodule this repository.

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive --test/unit-test/Unity
```

Platform Prerequisites

- For running unit tests
  - C90 compiler like gcc
  - CMake 3.13.0 or later
  - Ruby 2.0.0 or later is additionally required for the Unity test framework (that we use).
- For running the coverage target, lcov is additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described *above*.)
2. Create build directory: `mkdir build && cd build`
3. Run *cmake* while inside build directory: `cmake -S ../test`
4. Run this command to build the library and unit tests: `make all`
5. The generated test executables will be present in `build/bin/tests` folder.
6. Run `ctest` to execute all tests and view the test run summary.

Contributing See *CONTRIBUTING.md* for information on contributing.

Over-the-air Update Library

AWS IoT Over-the-air Update Library The OTA library enables you to manage the notification of a newly available update, download the update, and perform cryptographic verification of the firmware update. Using the library, you can logically separate firmware updates from the application running on your devices. The OTA library can share a network connection with the application, saving memory in resource-constrained devices. In addition, the OTA library lets you define application-specific logic for testing, committing, or rolling back a firmware update. The library supports different application protocols like Message Queuing Telemetry Transport (MQTT) and Hypertext Transfer Protocol (HTTP), and provides various configuration options you can fine tune depending on network type and conditions. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8. This library has also undergone static code analysis from Coverity static analysis.

See memory requirements for this library *here*.

AWS IoT Over-the-air Update Library v3.4.0 source code is part of the FreeRTOS 202210.00 LTS release.

AWS IoT Over-the-air Update Library v3.3.0 source code is part of the FreeRTOS 202012.01 LTS release.

AWS IoT Over-the-air Updates Config File

The AWS IoT Over-the-air Updates library exposes configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *ota_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named `ota_config.h` can be provided by the user application to the library.

By default, a `ota_config.h` custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide `OTA_DO_NOT_USE_CUSTOM_CONFIG` as a compile time preprocessor macro.

Building the Library

The *otaFilePaths.cmake* file contains the information of all source files and the header include paths required to build the AWS IoT Over-the-air Updates library.

As mentioned in the previous section, either a custom config file (i.e. `ota_config.h`) OR the `OTA_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the AWS IoT Over-the-air Updates library.

For a CMake example of building the AWS IoT Over-the-air Updates library with the `otaFilePaths.cmake` file, refer to the `coverity_analysis` library target in the *test/CMakeLists.txt* file.

Building Unit Tests

Checkout CMock Submodule

By default, the submodules in this repository are configured with `update=none` in *.gitmodules* to avoid increasing clone time and disk space usage of other repositories (like AWS IoT Device SDK for Embedded C that submodules this repository).

To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

Platform Prerequisites

- For building the library, **CMake 3.13.0** or later and a **C90 compiler**.
- For running unit tests, **Ruby 2.0.0** or later is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build unit tests

1. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described *above*.)
2. Run the *cmake* command: `cmake -S test -B build`

3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

Migration Guide

How to migrate from v2.0.0 (Release Candidate) to v3.4.0 The following table lists equivalent API function signatures in v2.0.0 (Release Candidate) and v3.4.0 declared in *ota.h*

v2.0.0 (Release Candidate)	v3.4.0	Notes
`OtaState_t OTA_Shutdown( uint32_t ticksToWait );`	`OtaState_t OTA_Shutdown( uint32_t ticksToWait, uint8_t unsubscribeFlag );`	`unsubscribeFlag` indicates if unsubscribe operations should be performed from the job topics when shutdown is called. Set this as 1 to unsubscribe, 0 otherwise.

How to migrate from version 1.0.0 to version 3.4.0 for OTA applications Refer to OTA Migration document for the summary of updates to the API. Migration document for OTA PAL also provides a summary of updates required for upgrading the OTA-PAL to work with v3.4.0 of the library.

Porting In order to support AWS IoT Over-the-air Updates on your device, it is necessary to provide the following components:

1. Port for the OTA Portable Abstraction Layer (PAL).
2. OS Interface
3. MQTT Interface

For enabling data transfer over HTTP dataplane the following component should also be provided:

1. HTTP Interface

NOTE When using OTA over HTTP dataplane, MQTT is required for control plane operations and should also be provided.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples Please refer to the demos of the AWS IoT Over-the-air Updates library in the following location for reference examples on POSIX and FreeRTOS:

Platform	Location
POSIX	AWS IoT Device SDK for Embedded C
FreeRTOS	FreeRTOS/FreeRTOS
FreeRTOS	FreeRTOS AWS Reference Integrations

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of coreMQTT may differ across repositories.

Generating documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

1.3 Connectivity

1.3.1 lwIP

This is the NXP fork of the lwIP networking stack.

- For details about changes and additions made by NXP, see CHANGELOG.
- For details about the NXP porting layer, see *The NXP lwIP Port*.
- For usage and API of lwIP, use official documentation at http://www.nongnu.org/lwip/.

The NXP lwIP Port

Below is description of possible settings of the port layer and an overview of a few helper functions.

The best place for redefinition of any mentioned macro is `lwipopts.h`.

The declaration of every mentioned function is in `ethernetif.h`. Please check the doxygen comments of those functions before.

Link state Physical link state (up/down) and its speed and duplex must be read out from PHY over MDIO bus. Especially link information is useful for lwIP stack so it can for example send DHCP discovery immediately when a link becomes up.

To simplify this port layer offers a function `ethernetif_probe_link()` which reads those data from PHY and forwards them into lwIP stack.

In almost all examples this function is called every `ETH_LINK_POLLING_INTERVAL_MS` (1500ms) by a function `probe_link_cyclic()`.

By setting `ETH_LINK_POLLING_INTERVAL_MS` to 0 polling will be disabled. On FreeRTOS, `probe_link_cyclic()` will be then called on an interrupt generated by PHY. GPIO port and pin for the interrupt line must be set in the ethernetifConfig struct passed to ethernetif_init(). On bare metal interrupts are not supported right now.

Rx task To improve the reaction time of the app, reception of packets is done in a dedicated task. The rx task stack size can be set by ETH_RX_TASK_STACK_SIZE macro, its priority by ETH_RX_TASK_PRIO.

If you want to save memory you can set reception to be done in an interrupt by setting ETH_DO_RX_IN_SEPARATE_TASK macro to 0.

Disabling Rx interrupt when out of buffers If ETH_DISABLE_RX_INT_WHEN_OUT_OF_BUFFERS is set to 1, then when the port gets out of Rx buffers, Rx enet interrupt will be disabled for a particular controller. Everytime Rx buffer is freed, Rx interrupt will be enabled.

This prevents your app from never getting out of Rx interrupt when the network is flooded with traffic.

ETH_DISABLE_RX_INT_WHEN_OUT_OF_BUFFERS is by default turned on, on FreeRTOS and off on bare metal.

Limit the number of packets read out from the driver at once on bare metal. You may define macro ETH_MAX_RX_PKTS_AT_ONCE to limit the number of received packets read out from the driver at once.

In case of heavy Rx traffic, lowering this number improves the realtime behaviour of an app. Increasing improves Rx throughput.

Setting it to value < 1 or not defining means "no limit".

Helper functions If your application needs to wait for the link to become up you can use one of the following functions:

- ethernetif_wait_linkup()- Blocks until the link on the passed netif is not up.
- ethernetif_wait_linkup_array() - Blocks until the link on at least one netif from the passed list of netifs becomes up.

If your app needs to wait for the IPv4 address on a particular netif to become different than "ANY" address (255.255.255.255) function ethernetif_wait_ipv4_valid() does this.

1.4 File System

1.4.1 FatFs

MCUXpresso SDK : mcuxsdk-middleware-fatfs

Overview This repository is for FatFs middleware delivery and it contains the components officially provided in NXP MCUXpresso SDK. This repository is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository (mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

Documentation Overall details can be reviewed here: MCUXpresso SDK Online Documentation

Visit FatFs - Documentation to review details on the contents in this sub-repo.

Setup Instructions on how to install the MCUXpresso SDK provided from GitHub via west manifest Getting Started with SDK - Detailed Installation Instructions

Contribution Contributions are not currently accepted. Guidelines to contribute will be posted in the future.

Repo Specific Content This is MCUXpresso SDK fork of FatFs (FAT file system created by ChaN). Official documentation is available at http://elm-chan.org/fsw/ff/

MCUXpresso version is extending original content by following hardware specific porting layers:

- mmc_disk
- nand_disk
- ram_disk
- sd_disk
- sdspi_disk
- usb_disk

Changelog FatFs

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog

[R0.15_rev0]

- Upgraded to version 0.15
- Applied patches from http://elm-chan.org/fsw/ff/patches.html

[R0.14b_rev1]

- Applied patches from http://elm-chan.org/fsw/ff/patches.html

[R0.14b_rev0]

- Upgraded to version 0.14b

[R0.14a_rev0]

- Upgraded to version 0.14a
- Applied patch ff14a_p1.diff and ff14a_p2.diff

[R0.14_rev0]

- Upgraded to version 0.14
- Applied patch ff14_p1.diff and ff14_p2.diff

[R0.13c_rev0]

- Upgraded to version 0.13c
- Applied patches ff_13c_p1.diff,ff_13c_p2.diff, ff_13c_p3.diff and ff_13c_p4.diff.

[R0.13b_rev0]

- Upgraded to version 0.13b

[R0.13a_rev0]

- Upgraded to version 0.13a. Added patch ff_13a_p1.diff.

[R0.12c_rev1]

- Add NAND disk support.

[R0.12c_rev0]

- Upgraded to version 0.12c and applied patches ff_12c_p1.diff and ff_12c_p2.diff.

[R0.12b_rev0]

- Upgraded to version 0.12b.

[R0.11a]

- Added glue functions for low-level drivers (SDHC, SDSPI, RAM, MMC). Modified diskio.c.
- Added RTOS wrappers to make FatFs thread safe. Modified syscall.c.
- Renamed ffconf.h to ffconf_template.h. Each application should contain its own ffconf.h.
- Included ffconf.h into diskio.c to enable the selection of physical disk from ffconf.h by macro definition.
- Conditional compilation of physical disk interfaces in diskio.c.

1.5 Motor Control

1.5.1 FreeMASTER

Communication Driver User Guide

Introduction

What is FreeMASTER? FreeMASTER is a PC-based application developed by NXP for NXP customers. It is a versatile tool usable as a real-time monitor, visualization tool, and a graphical control panel of embedded applications based on the NXP processing units.

This document describes the embedded-side software driver which implements an interface between the application and the host PC. The interface covers the following communication:

- **Serial** UART communication either over plain RS232 interface or more typically over a USB-to-Serial either external or built in a debugger probe.

- **USB** direct connection to target microcontroller
- **CAN bus**
- **TCP/IP network** wired or WiFi
- **Segger J-Link RTT**
- **JTAG** debug port communication
- …and all of the above also using a **Zephyr** generic drivers.

The driver also supports so-called "packet-driven BDM" interface which enables a protocol-based communication over a debugging port. The BDM stands for Background Debugging Module and its physical implementation is different on each platform. Some platforms leverage a semi-standard JTAG interface, other platforms provide a custom implementation called BDM. Regardless of the name, this debugging interface enables non-intrusive access to the memory space while the target CPU is running. For basic memory read and write operations, there is no communication driver required on the target when communicating with the host PC. Use this driver to get more advanced FreeMASTER protocol features over the BDM interface. The driver must be configured for the packet-driven BDM mode, in which the host PC uses the debugging interface to write serial command frames directly to the target memory buffer. The same method is then used to read response frames from that memory buffer.

Similar to "packet-driven BDM", the FreeMASTER also supports a communication over [J-Link RTT]((https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/) interface defined by SEGGER Microcontroller GmbH for ARM CortexM-based microcontrollers. This method also uses JTAG physical interface and enables high-speed real time communication to run over the same channel as used for application debugging.

Driver version 3 This document describes version 3 of the FreeMASTER Communication Driver. This version features the implementation of the new Serial Protocol, which significantly extends the features and security of its predecessor. The new protocol internal number is v4 and its specification is available in the documentation accompanying the driver code.

Driver V3 is deployed to modern 32-bit MCU platforms first, so the portfolio of supported platforms is smaller than for the previous V2 versions. It is recommended to keep using the V2 driver for legacy platforms, such as S08, S12, ColdFire, or Power Architecture. Reach out to FreeMASTER community or to the local NXP representative with requests for more information or to port the V3 driver to legacy MCU devices.

Thanks to a layered approach, the new driver simplifies the porting of the driver to new UART, CAN or networking communication interfaces significantly. Users are encouraged to port the driver to more NXP MCU platforms and contribute the code back to NXP for integration into future releases. Existing code and low-level driver layers may be used as an example when porting to new targets.

Note: Using the FreeMASTER tool and FreeMASTER Communication Driver is only allowed in systems based on NXP microcontroller or microprocessor unit. Use with non-NXP MCU platforms is **not permitted** by the license terms.

Target platforms The driver implementation uses the following abstraction mechanisms which simplify driver porting and supporting new communication modules:

- **General CPU Platform** (see source code in the src/platforms directory). The code in this layer is only specific to native data type sizes and CPU architectures (for example; alignment-aware memory copy routines). This driver version brings two generic implementations of 32-bit platforms supporting both little-endian and big-endian architectures. There are also implementations customized for the 56F800E family of digital signal controllers and S12Z MCUs. **Zephyr** is treated as a specific CPU platform as it brings unified user configuration (Kconfig) and generic hardware device drivers. With Zephyr, the transport layer and low-level communication layers described below are configured automatically using Kconfig and Device Tree technologies.

- **Transport Communication Layer** - The Serial, CAN, Networking, PD-BDM, and other methods of transport logic are implemented as a driver layer called FMSTR_TRANSPORT with a uniform API. A support of the Network transport also extends single-client modes of operation which are native for Serial, USB and CAN by a concept of multiple client sessions.
- **Low-level Communication Driver** - Each type of transport further defines a low-level API used to access the physical communication module. For example, the Serial transport defines a character-oriented API implemented by different serial communication modules like UART, LPUART, USART, and also USB-CDC. Similarly, the CAN transport defines a message-oriented API implemented by the FlexCAN or MCAN modules. Moreover, there are multiple different implementations for the same kind of communication peripherals. The difference between the implementation is in the way the low-level hardware registers are accessed. The *mcuxsdk* folder contains implementations which use MCUXpresso SDK drivers. These drivers should be used in applications based on the NXP MCUXpresso SDK. The "ampsdk" drivers target automotive-specific MCUs and their respective SDKs. The "dreg" implementations use a plain C-language access to hardware register addresses which makes it a universal and the most portable solution. In this case, users are encouraged to add more drivers for other communication modules or other respective SDKs and contribute the code back to NXP for integration.

  The low-level drivers defined for the Networking transport enable datagram-oriented UDP and stream TCP communication. This implementation is demonstrated using the lwIP software stack but shall be portable to other TCP/IP stacks. It may sound surprisingly, but also the Segger J-Link RTT communication driver is linked to the Networking transport (RTT is stream oriented communication handled similarly to TCP).

Replacing existing drivers For all supported platforms, the driver described in this document replaces the V2 implementation and also older driver implementations that were available separately for individual platforms (PC Master SCI drivers).

Clocks, pins, and peripheral initialization The FreeMASTER communication driver is only responsible for runtime processing of the communication and must be integrated with an user application code to function properly. The user application code is responsible for general initialization of clock sources, pin multiplexers, and peripheral registers related to the communication speed. Such initialization should be done before calling the FMSTR_Init function.

It is recommended to develop the user application using one of the Software Development Kits (SDKs) available from third parties or directly from NXP, such as MCUXpresso SDK, MCUXpresso IDE, and related tools. This approach simplifies the general configuration process significantly.

MCUXpresso SDK The MCUXpresso SDK is a software package provided by NXP which contains the device initialization code, linker files, and software drivers with example applications for the NXP family of MCUs. The MCUXpresso Config Tools may be used to generate the clock-setup and pin-multiplexer setup code suitable for the selected processor.

The MCUXpresso SDK also contains this FreeMASTER communication driver as a "middleware" component which may be downloaded along with the example applications from https://mcuxpresso.nxp.com/en/welcome.

MCUXpresso SDK on GitHub The FreeMASTER communication driver is also released as one of the middleware components of the MCUXpresso SDK on the GitHub. This release enables direct integration of the FreeMASTER source code Git repository into a target applications including Zephyr applications.

Related links:

- The official FreeMASTER middleware repository.
- Online version of this document

FreeMASTER in Zephyr The FreeMASTER middleware repository can be used with MCUXpresso SDK as well as a Zephyr module. Zephyr-specific samples which include examples of Kconfig and Device Tree configurations for Serial, USB and Network communications are available in separate repository. West manifest in this sample repository fetches the full Zephyr package including the FreeMASTER middleware repository used as a Zephyr module.

Example applications

MCUX SDK Example applications There are several example applications available for each supported MCU platform.

- **fmstr_uart** demonstrates a plain serial transmission, typically connecting to a computer's physical or virtual COM port. The typical transmission speed is 115200 bps.
- **fmstr_can** demonstrates CAN bus communication. This requires a suitable CAN interface connected to the computer and interconnected with the target MCU using a properly terminated CAN bus. The typical transmission speed is 500 kbps. A FreeMASTER-over-CAN communication plug-in must be used.
- **fmstr_usb_cdc** uses an on-chip USB controller to implement a CDC communication class. It is connected directly to a computer's USB port and creates a virtual COM port device. The typical transmission speed is above 1 Mbps.
- **fmstr_net** demonstrates the Network communication over UDP or TCP protocol. Existing examples use lwIP stack to implement the communication, but in general, it shall be possible to use any other TCP/IP stack to achieve the same functionality.
- **fmstr_wifi** is the fmstr_net application modified to use a WiFi network interface instead of a wired Ethernet connection.
- **fmstr_rtt** demonstrates the communication over SEGGER J-Link RTT interface. Both fmstr_net and fmstr_rtt examples require the FreeMASTER TCP/UDP communication plug-in to be used on the PC host side.
- **fmstr_eonce** uses the real-time data unit on the JTAG EOnCE module of the 56F800E family to implement pseudo-serial communication over the JTAG port. The typical transmission speed is around 10 kbps. This communication requires FreeMASTER JTAG/EOnCE communication plug-in.
- **fmstr_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER "middleware" driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

Zephyr sample spplications Zephyr sample applications demonstrate Kconfig and Device Tree configuration which configure the FreeMASTER middleware module for a selected communication option (Serial, CAN, Network or RTT).

Refer to *readme.md* files in each sample directory for description of configuration options required to implement FreeMASTER connectivity.

Description

This section shows how to add the FreeMASTER Communication Driver into application and how to configure the connection to the FreeMASTER visualization tool.

Features The FreeMASTER driver implements the FreeMASTER protocol V4 and provides the following features which may be accessed using the FreeMASTER visualization tool:

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.
- Atomic bit manipulation on the target memory (bit-wise write access).
- Optimal size-aligned access to memory which is also suitable to access the peripheral register space.
- Oscilloscope access—real-time access to target variables. The sample rate may be limited by the communication speed.
- Recorder— access to the fast transient recorder running on the board as a part of the FreeMASTER driver. The sample rate is only limited by the MCU CPU speed. The length of the data recorded depends on the amount of available memory.
- Multiple instances of Oscilloscopes and Recorders without the limitation of maximum number of variables.
- Application commands—high-level message delivery from the PC to the application.
- TSA tables—describing the data types, variables, files, or hyperlinks exported by the target application. The TSA newly supports also non-memory mapped resources like external EEPROM or SD Card files.
- Pipes—enabling the buffered stream-oriented data exchange for a general-purpose terminal-like communication, diagnostic data streaming, or other data exchange.

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.

- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

The following sections briefly describe all FreeMASTER features implemented by the driver. See the PC-based FreeMASTER User Manual for more details on how to use the features to monitor, tune, or control an embedded application.

Board Detection The FreeMASTER protocol V4 defines the standard set of configuration values which the host PC tool reads to identify the target and to access other target resources properly. The configuration includes the following parameters:

- Version of the driver and the version of the protocol implemented.
- MTU as the Maximum size of the Transmission Unit (for example; communication buffer size).
- Application name, description, and version strings.
- Application build date and time as a string.
- Target processor byte ordering (little/big endian).
- Protection level that requires password authentication.
- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

Memory Read This basic feature enables the host PC to read any data memory location by specifying the address and size of the required memory area. The device response frame must be shorter than the MTU to fit into the outgoing communication buffer. To read a device memory of any size, the host uses the information retrieved during the Board Detection and splits the large-block request to multiple partial requests.

The driver uses size-aligned operations to read the target memory (for example; uses proper read-word instruction when an address is aligned to 4 bytes).

Memory Write Similarly to the Memory Read operation, the Memory Write feature enables to write to any RAM memory location on the target device. A single write command frame must be shorter than the MTU to fit into the target communication buffer. Larger requests must be split into smaller ones.

The driver uses size-aligned operations to write to the target memory (for example; uses proper write-word instruction when an address is aligned to 4 bytes).

Masked Memory Write To implement the write access to a single bit or a group of bits of target variables, the Masked Memory Write feature is available in the FreeMASTER protocol and it is supported by the driver using the Read-Modify-Write approach.

Be careful when writing to bit fields of volatile variables that are also modified in an application interrupt. The interrupt may be serviced in the middle of a read-modify-write operation and it may cause data corruption.

Oscilloscope The protocol and driver enables any number of variables to be read at once with a single request from the host. This feature is called Oscilloscope and the FreeMASTER tool uses it to display a real-time graph of variable values.

The driver can be configured to support any number of Oscilloscope instances and enable simultaneously running graphs to be displayed on the host computer screen.

Recorder The protocol enables the host to select target variables whose values are then periodically recorded into a dedicated on-board memory buffer. After such data sampling stops (either on a host request or by evaluating a threshold-crossing condition), the data buffer is downloaded to the host and displayed as a graph. The data sampling rate is not limited by the speed of the communication line, so it enables displaying the variable transitions in a very high resolution.

The driver can be configured to support multiple Recorder instances and enable multiple recorder graphs to be displayed on the host screen. Having multiple recorders also enables setting the recording point differently for each instance. For example; one instance may be recording data in a general timer interrupt while another instance may record at a specific control algorithm time in the PWM interrupt.

TSA With the TSA feature, data types and variables can be described directly in the application source code. Such information is later provided to the FreeMASTER tool which may use it instead of reading symbol data from the application ELF executable file.

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

The driver also supports special types of TSA table entries to describe user resources like external EEPROM and SD Card files, memory-mapped files, virtual directories, web URL hyperlinks, and constant enumerations.

TSA Safety When the TSA is enabled in the application, the TSA Safety can be enabled and validate the memory accesses directly by the embedded-side driver. When the TSA Safety is turned on, any memory request received from the host is validated and accepted only if it belongs to a TSA-described object. The TSA entries can be declared as Read-Write or Read-Only so that the driver can actively deny the write access to the Read-Only objects.

Application commands The Application Commands are high-level messages that can be delivered from the PC Host to the embedded application for further processing. The embedded application can either poll the status, or be called back when a new Application Command arrives to be processed. After the embedded application acknowledges that the command is handled, the host receives the Result Code and reads the other return data from memory. Both the Application Commands and the Result Codes are specific to a given application and it is user's responsibility to define them. The FreeMASTER protocol and the FreeMASTER driver only implement the delivery channel and a set of API calls to enable the Application Command processing in general.

Pipes The Pipes enable buffered and stream-oriented data exchange between the PC Host and the target application. Any pipe can be written to and read from at both ends (either on the PC or the MCU). The data transmission is acknowledged using the special FreeMASTER protocol commands. It is guaranteed that the data bytes are delivered from the writer to the reader in a proper order and without losses.

Serial single-wire operation The MCU Serial Communication Driver natively supports normal dual-wire operation. Because the protocol is half-duplex only, the driver can also operate in two single-wire modes:

- "External" single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.

- "True" single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

Multi-session support With networking interface it is possible for multiple clients to access the target MCU simultaneously. Reading and writing of target memory is processed atomically so there is no risk of data corruption. The state-full resources such as Recorders or Oscilloscopes are locked to a client session upon first use and access is denied to other clients until lock is released..

Zephyr-specific

Dedicated communication task FreeMASTER communication may run isolated in a dedicated task. The task automates the FMSTR_Init and FMSTR_Poll calls together with periodic activities enabling the FreeMASTER UI to fetch information about tasks and CPU utilization. The task can be started automatically or manually, and it must be assigned a priority to be able to react on interrupts and other communication events. Refer to Zephyr FreeMASTER sample applications which all use this communication task.

Zephyr shell and logging over FreeMASTER pipe FreeMASTER implements a shell backend which may use FreeMASTER pipe as a I/O terminal and logging output. Refer to Zephyr FreeMASTER sample applications which all use this feature.

Automatic TSA tables TSA tables can be declared as "automatic" in Zephyr which make them automatically registered in the table list. This may be very useful when there are many TSA tables or when the tables are defined in different (often unrelated) libraries linked together. In this case user does not need to build a list of all tables manually.

Driver files The driver source files can be found in a top-level src folder, further divided into the sub-folders:

- ***src/platforms*** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- ***src/common*** folder—contains the common driver source files shared by the driver for all supported platforms. All the *.c* files must be added to the project, compiled, and linked together with the application.
  - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
  - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
  - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
  - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
  - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.

- *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
- *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
- *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
- *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
- *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
- *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
- *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
- *freemaster_serial.h* - defines the low-level character-oriented Serial API.
- *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
- *freemaster_can.h* - defines the low-level message-oriented CAN API.
- *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
- *freemaster_net.h* - definitions related to the Network transport.
- *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
- *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
- *freemaster_utils.h* - definitions related to utility code.

- ***src/drivers/[sdk]/serial*** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
  - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.

- ***src/drivers/[sdk]/can*** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
  - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- ***src/drivers/[sdk]/network*** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
  - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
  - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

Driver configuration The driver is configured using a single header file (*freemaster_cfg.h*). Create this file and save it together with other project source files before compiling the driver code. All FreeMASTER driver source files include the *freemaster_cfg.h* file and use the macros defined here for the conditional and parameterized compilation. The C compiler must locate the configuration file when compiling the driver files. Typically, it can be achieved by putting this file into a folder where the other project-specific included files are stored.

As a starting point to create the configuration file, get the *freemaster_cfg.h.example* file, rename it to *freemaster_cfg.h*, and save it into the project area.

Note: It is NOT recommended to leave the *freemaster_cfg.h* file in the FreeMASTER driver source code folder. The configuration file must be placed at a project-specific location, so that it does not affect the other applications that use the same driver.

Configurable items This section describes the configuration options which can be defined in *freemaster_cfg.h*.

Interrupt modes

```
#define FMSTR_LONG_INTR    [0|1]
#define FMSTR_SHORT_INTR   [0|1]
#define FMSTR_POLL_DRIVEN  [0|1]
```

Value Type boolean (0 or 1)

Description Exactly one of the three macros must be defined to non-zero. The others must be defined to zero or left undefined. The non-zero-defined constant selects the interrupt mode of the driver. See *Driver interrupt modes*.

- FMSTR_LONG_INTR — long interrupt mode
- FMSTR_SHORT_INTR — short interrupt mode
- FMSTR_POLL_DRIVEN — poll-driven mode

Note: Some options may not be supported by all communication interfaces. For example, the FMSTR_SHORT_INTR option is not supported by the USB_CDC interface.

Protocol transport

```
#define FMSTR_TRANSPORT [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER source code. Specify one of existing instances to make use of the protocol transport.

Description Use one of the pre-defined constants, as implemented by the FreeMASTER code. The current driver supports the following transports:

- **FMSTR_SERIAL** - serial communication protocol
- **FMSTR_CAN** - using CAN communication
- **FMSTR_PDBDM** - using packet-driven BDM communication
- **FMSTR_NET** - network communication using TCP or UDP protocol

Serial transport This section describes configuration parameters used when serial transport is used:

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

FMSTR_SERIAL_DRV Select what low-level driver interface will be used when implementing the Serial communication.

```
#define FMSTR_SERIAL_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing serial driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/serial* implementation):

- **FMSTR_SERIAL_MCUX_UART** - UART driver
- **FMSTR_SERIAL_MCUX_LPUART** - LPUART driver
- **FMSTR_SERIAL_MCUX_USART** - USART driver
- **FMSTR_SERIAL_MCUX_MINIUSART** - miniUSART driver
- **FMSTR_SERIAL_MCUX_QSCI** - DSC QSCI driver
- **FMSTR_SERIAL_MCUX_USB** - USB/CDC class driver (also see code in the */support/mcuxsdk_usb* folder)
- **FMSTR_SERIAL_56F800E_EONCE** - DSC JTAG EOnCE driver

Other SDKs or BSPs may define custom low-level driver interface structure which may be used as FMSTR_SERIAL_DRV. For example:

- **FMSTR_SERIAL_DREG_UART** - demonstrates the low-level interface implemented without the MCUXpresso SDK and using direct access to peripheral registers.

FMSTR_SERIAL_BASE

```
#define FMSTR_SERIAL_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the UART, LPUART, USART, or other serial peripheral module to be used for the communication. This value is not defined by default. User application should call FMSTR_SetSerialBaseAddress() to select the peripheral module.

FMSTR_COMM_BUFFER_SIZE

```
#define FMSTR_COMM_BUFFER_SIZE [number]
```

Value Type 0 or a value in range 32…255

Description Specify the size of the communication buffer to be allocated by the driver. Default value, which suits all driver features, is used when this option is defined as 0.

FMSTR_COMM_RQUEUE_SIZE

```
#define FMSTR_COMM_RQUEUE_SIZE [number]
```

Value Type Value in range 0…255

Description Specify the size of the FIFO receiver queue used to quickly receive and store characters in the FMSTR_SHORT_INTR interrupt mode.
The default value is 32 B.

FMSTR_SERIAL_SINGLEWIRE

```
#define FMSTR_SERIAL_SINGLEWIRE [0|1]
```

Value Type Boolean 0 or 1.

Description Set to non-zero to enable the "True" single-wire mode which uses a single MCU pin to communicate. The low-level driver enables the pin direction switching when the MCU peripheral supports it.

CAN Bus transport This section describes configuration parameters used when CAN transport is used:

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

FMSTR_CAN_DRV Select what low-level driver interface will be used when implementing the CAN communication.

```
#define FMSTR_CAN_DRV [identifier]
```

Value Type Driver identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing CAN driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/can implementation*):

- **FMSTR_CAN_MCUX_FLEXCAN** - FlexCAN driver
- **FMSTR_CAN_MCUX_MCAN** - MCAN driver
- **FMSTR_CAN_MCUX_MSCAN** - msCAN driver
- **FMSTR_CAN_MCUX_DSCFLEXCAN** - DSC FlexCAN driver
- **FMSTR_CAN_MCUX_DSCMSCAN** - DSC msCAN driver

Other SDKs or BSPs may define the custom low-level driver interface structure which may be used as FMSTR_CAN_DRV.

FMSTR_CAN_BASE

```
#define FMSTR_CAN_BASE [address|symbol]
```

Value Type Optional address value (numeric or symbolic)

Description Specify the base address of the FlexCAN, msCAN, or other CAN peripheral module to be used for the communication. This value is not defined by default. User application should call FMSTR_SetCanBaseAddress() to select the peripheral module.

FMSTR_CAN_CMDID

```
#define FMSTR_CAN_CMDID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for FreeMASTER commands (direction from PC Host tool to target application). When declaring 29-bit identifier, combine the numeric value with FMSTR_CAN_EXTID bit. Default value is 0x7AA.

FMSTR_CAN_RSPID

```
#define FMSTR_CAN_RSPID [number]
```

Value Type CAN identifier (11-bit or 29-bit number)

Description CAN message identifier used for responding messages (direction from target application to PC Host tool). When declaring 29-bit identifier, combine the numeric value with FMSTR_CAN_EXTID bit. Note that both *CMDID* and *RSPID* values may be the same. Default value is 0x7AA.

FMSTR_FLEXCAN_TXMB

```
#define FMSTR_FLEXCAN_TXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame transmission. Default value is 0.

FMSTR_FLEXCAN_RXMB

```
#define FMSTR_FLEXCAN_RXMB [number]
```

Value Type Number in range of 0..N where N is number of CAN message-buffers supported by HW module.

Description Only used when the FlexCAN low-level driver is used. Define the FlexCAN message buffer for CAN frame reception. Note that the FreeMASTER driver may also operate with a common message buffer used by both TX and RX directions. Default value is 1.

Network transport This section describes configuration parameters used when Network transport is used:

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

```
#define FMSTR_NET_DRV [identifier]
```

Value Type Identifiers are structure instance names defined in FreeMASTER drivers code. Specify one of existing NET driver instances.

Description When using MCUXpresso SDK, use one of the following constants (see */drivers/mcuxsdk/network implementation*):

- **FMSTR_NET_LWIP_TCP** - TCP communication using lwIP stack
- **FMSTR_NET_LWIP_UDP** - UDP communication using lwIP stack
- **FMSTR_NET_SEGGER_RTT** - Communication using SEGGER J-Link RTT interface

Other SDKs or BSPs may define the custom networking interface which may be used as FMSTR_CAN_DRV.

Add another row below:

FMSTR_NET_PORT

```
#define FMSTR_NET_PORT [number]
```

Value Type TCP or UDP port number (short integer)

Description Specifies the server port number used by TCP or UDP protocols.

FMSTR_NET_BLOCKING_TIMEOUT

```
#define FMSTR_NET_BLOCKING_TIMEOUT [number]
```

Value Type Timeout as number of milliseconds

Description This value specifies a timeout in milliseconds for which the network socket operations may block the execution inside *FMSTR_Poll*. This may be set high (e.g. 250) when a dedicated RTOS task is used to handle FreeMASTER protocol polling. Set to a lower value when the polling task is also responsible for other operations. Set to 0 to attempt to use non-blocking socket operations.

FMSTR_NET_AUTODISCOVERY

```
#define FMSTR_NET_AUTODISCOVERY [0|1]
```

Value Type Boolean 0 or 1.

Description This option enables the FreeMASTER driver to use a separate UDP socket to broadcast auto-discovery messages to network. This helps the FreeMASTER tool to discover the target device address, port and protocol options.

Debugging options

FMSTR_DISABLE

```
#define FMSTR_DISABLE [0|1]
```

Value Type boolean (0 or 1)

Description Define as non-zero to disable all FreeMASTER features, exclude the driver code from build, and compile all its API functions empty. This may be useful to remove FreeMASTER without modifying any application source code. Default value is 0 (false).

FMSTR_DEBUG_TX

```
#define FMSTR_DEBUG_TX [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to enable the driver to periodically transmit test frames out on the selected communication interface (SCI or CAN). With the debug transmission enabled, it is simpler to detect problems in the baudrate or other communication configuration settings.

The test frames are transmitted until the first valid command frame is received from the PC Host tool. The test frame is a valid error status frame, as defined by the protocol format. On the serial line, the test frame consists of three printable characters (+©W) which are easy to capture using the serial terminal tools.

This feature requires the FMSTR_Poll() function to be called periodically. Default value is 0 (false).

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

```
#define FMSTR_USE_READMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Read command and enable FreeMASTER to have read access to memory and variables. The access can be further restricted by using a TSA feature.
Default value is 1 (true).

FMSTR_USE_WRITEMEM

```
#define FMSTR_USE_WRITEMEM [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Memory Write command.
The default value is 1 (true).

Oscilloscope options

FMSTR_USE_SCOPE

```
#define FMSTR_USE_SCOPE [number]
```

Value Type Integer number.

Description Number of Oscilloscope instances to be supported. Set to 0 to disable the Oscilloscope feature.
Default value is 0.

FMSTR_MAX_SCOPE_VARS

```
#define FMSTR_MAX_SCOPE_VARS [number]
```

Value Type Integer number larger than 2.

Description Number of variables to be supported by each Oscilloscope instance.
Default value is 8.

Recorder options

FMSTR_USE_RECORDER

```
#define FMSTR_USE_RECORDER [number]
```

Value Type Integer number.

Description Number of Recorder instances to be supported. Set to 0 to disable the Recorder feature.
Default value is 0.

FMSTR_REC_BUFF_SIZE

```
#define FMSTR_REC_BUFF_SIZE [number]
```

Value Type Integer number larger than 2.

Description Defines the size of the memory buffer used by the Recorder instance #0.
Default: not defined, user shall call 'FMSTR_RecorderCreate()" API function to specify this parameter in run time.

FMSTR_REC_TIMEBASE

```
#define FMSTR_REC_TIMEBASE [time specification]
```

Value Type Number (nanoseconds time).

Description Defines the base sampling rate in nanoseconds (sampling speed) Recorder instance #0.

Use one of the following macros:

- FMSTR_REC_BASE_SECONDS(x)
- FMSTR_REC_BASE_MILLISEC(x)
- FMSTR_REC_BASE_MICROSEC(x)
- FMSTR_REC_BASE_NANOSEC(x)

Default: not defined, user shall call 'FMSTR_RecorderCreate()" API function to specify this parameter in run time.

FMSTR_REC_FLOAT_TRIG

```
#define FMSTR_REC_FLOAT_TRIG [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the floating-point triggering. Be aware that floating-point triggering may grow the code size by linking the floating-point standard library.

Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

```
#define FMSTR_USE_APPCMD [0|1]
```

Value Type Boolean 0 or 1.

Description Define as non-zero to implement the Application Commands feature.
Default value is 0 (false).

FMSTR_APPCMD_BUFF_SIZE

```
#define FMSTR_APPCMD_BUFF_SIZE [size]
```

Value Type Numeric buffer size in range 1..255

Description The size of the Application Command data buffer allocated by the driver. The buffer stores the (optional) parameters of the Application Command which waits to be processed.

FMSTR_MAX_APPCMD_CALLS

```
#define FMSTR_MAX_APPCMD_CALLS [number]
```

Value Type Number in range 0..255

Description The number of different Application Commands that can be assigned a callback handler function using `FMSTR_RegisterAppCmdCall()`. Default value is 0.

TSA options

FMSTR_USE_TSA

```
#define FMSTR_USE_TSA [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER TSA feature to be used. With this option enabled, the TSA tables defined in the applications are made available to the FreeMASTER host tool.
Default value is 0 (false).

FMSTR_USE_TSA_SAFETY

```
#define FMSTR_USE_TSA_SAFETY [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the memory access validation in the FreeMASTER driver. With this option, the host tool is not able to access the memory which is not described by at least one TSA descriptor. Also a write access is denied for objects defined as read-only in TSA tables.
Default value is 0 (false).

FMSTR_USE_TSA_INROM

```
#define FMSTR_USE_TSA_INROM [0|1]
```

Value Type Boolean 0 or 1.

Description Declare all TSA descriptors as *const*, which enables the linker to put the data into the flash memory. The actual result depends on linker settings or the linker commands used in the project.
Default value is 0 (false).

FMSTR_USE_TSA_DYNAMIC

```
#define FMSTR_USE_TSA_DYNAMIC [0|1]
```

Value Type Boolean 0 or 1.

Description Enable runtime-defined TSA entries to be added to the TSA table by the FMSTR_SetUpTsaBuff() and FMSTR_TsaAddVar() functions.
Default value is 0 (false).

Pipes options

FMSTR_USE_PIPES

```
#define FMSTR_USE_PIPES [0|1]
```

Value Type Boolean 0 or 1.

Description Enable the FreeMASTER Pipes feature to be used.
Default value is 0 (false).

FMSTR_MAX_PIPES_COUNT

```
#define FMSTR_MAX_PIPES_COUNT [number]
```

Value Type Number in range 1..63.

Description The number of simultaneous pipe connections to support.
The default value is 1.

Driver interrupt modes To implement the communication, the FreeMASTER driver handles the Serial or CAN module's receive and transmit requests. Use the *freemaster_cfg.h* configuration file to select whether the driver processes the communication automatically in the interrupt service routine handler or if it only polls the status of the module (typically during the application idle time).

This section describes each of the interrupt mode in more details.

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

In this mode, both the communication and the FreeMASTER protocol decoding is done in the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, or other interrupt service routine. Because the protocol execution may be a lengthy task (especially with the TSA-Safety enabled) it is recommended to use this mode only if the interrupt prioritization scheme is possible in the application and the FreeMASTER interrupt is assigned to a lower (the lowest) priority.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from that handler.

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

In this mode, the communication processing time is split between the interrupt routine and the main application loop or task. The raw communication is handled by the *FMSTR_SerialIsr, FMSTR_CanIsr*, or other interrupt service routine, while the protocol decoding and execution is handled by the *FMSTR_Poll* routine. Call *FMSTR_Poll* during the idle time in the application main loop.

The interrupt processing in this mode is relatively fast and deterministic. Upon a serial-receive event, the received character is only placed into a FIFO-like queue and it is not further processed. Upon a CAN receive event, the received frame is stored into a receive buffer. When transmitting, the characters are fetched from the prepared transmit buffer.

In this mode, the application code must register its own interrupt handler for all interrupt vectors related to the selected communication interface and call the *FMSTR_SerialIsr* or *FMSTR_CanIsr* functions from that handler.

When the serial interface is used as the serial communication interface, ensure that the *FMSTR_Poll* function is called at least once per *N* character time periods. *N* is the length of the FreeMASTER FIFO queue (*FMSTR_COMM_RQUEUE_SIZE*) and the character time is the time needed to transmit or receive a single byte over the SCI line.

Completely Poll-driven

```
#define FMSTR_POLL_DRIVEN 1
```

In this mode, both the communication and the FreeMASTER protocol decoding are done in the *FMSTR_Poll* routine. No interrupts are needed and the *FMSTR_SerialIsr*, *FMSTR_CanIsr*, and similar handlers compile to an empty code.

When using this mode, ensure that the *FMSTR_Poll* function is called by the application at least once per the serial "character time" which is the time needed to transmit or receive a single character.

In the latter two modes (*FMSTR_SHORT_INTR* and *FMSTR_POLL_DRIVEN*), the protocol handling takes place in the FMSTR_Poll routine. An application interrupt can occur in the middle of the Read Memory or Write Memory commands' execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, some issues or glitches may occur when using FreeMASTER to visualize or monitor volatile variables modified in interrupt servicing code.

The same issue may appear even in the full interrupt mode (FMSTR_LONG_INTR), if volatile variables are modified in the interrupt code with a priority higher than the priority of the communication interrupt.

Data types Simple portability was one of the main requirements when writing the FreeMASTER driver. This is why the driver code uses the privately-declared data types and the vast majority of the platform-dependent code is separated in the platform-dependent source files. The data types used in the driver API are all defined in the platform-specific header file.

To prevent name conflicts with the symbols used in the application, all data types, macros, and functions have the FMSTR_ prefix. The only global variables used in the driver are the transport and low-level API structures exported from the driver-implementation layer to upper layers. Other than that, all private variables are declared as static and named using the fmstr_ prefix.

Communication interface initialization The FreeMASTER driver does not perform neither the initialization nor the configuration of the peripheral module that it uses to communicate. It is the application startup code responsibility to configure the communication module before the FreeMASTER driver is initialized by the FMSTR_Init call.

When the Serial communication module is used as the FreeMASTER communication interface, configure the UART receive and transmit pins, the serial communication baud rate, parity (no-parity), the character length (eight bits), and the number of stop bits (one) before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see *Driver interrupt modes*), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected serial peripheral module. Call the FMSTR_SerialIsr function from the application handler.

When a CAN module is used as the FreeMASTER communication interface, configure the CAN receive and transmit pins and the CAN module bit rate before initializing the FreeMASTER driver. For either the long or the short interrupt modes of the driver (see *Driver interrupt modes*), configure the interrupt controller and register an application-specific interrupt handler for all interrupt sources related to the selected CAN peripheral module. Call the FMSTR_CanIsr function from the application handler.

Note: It is not necessary to enable or unmask the serial nor the CAN interrupts before initializing the FreeMASTER driver. The driver enables or disables the interrupts and communication lines, as required during runtime.

FreeMASTER Recorder calls When using the FreeMASTER Recorder in the application (FMSTR_USE_RECORDER > 0), call the FMSTR_RecorderCreate function early after FMSTR_Init to set up each recorder instance to be used in the application. Then call the FMSTR_Recorder function periodically in the code where the data recording should occur. A typical place to call the Recorder routine is at the timer or PWM interrupts, but it can be anywhere else. The example applications provided together with the driver code call the FMSTR_Recorder in the main application loop.

In applications where FMSTR_Recorder is called periodically with a constant period, specify the period in the Recorder configuration structure before calling FMSTR_RecorderCreate. This setting enables the PC Host FreeMASTER tool to display the X-axis of the Recorder graph properly scaled for the time domain.

Driver usage Start using or evaluating FreeMASTER by opening some of the example applications available in the driver setup package.

Follow these steps to enable the basic FreeMASTER connectivity in the application:

- Make sure that all **.c* files of the FreeMASTER driver from the *src/common/platforms/[your_platform]* folder are a part of the project. See *Driver files* for more details.
- Configure the FreeMASTER driver by creating or editing the *freemaster_cfg.h* file and by saving it into the application project directory. See *Driver configuration* for more details.
- Include the *freemaster.h* file into any application source file that makes the FreeMASTER API calls.
- Initialize the Serial or CAN modules. Set the baud rate, parity, and other parameters of the communication. Do not enable the communication interrupts in the interrupt mask registers.
- For the FMSTR_LONG_INTR and FMSTR_SHORT_INTR modes, install the application-specific interrupt routine and call the FMSTR_SerialIsr or FMSTR_CanIsr functions from this handler.
- Call the FMSTR_Init function early on in the application initialization code.
- Call the FMSTR_RecorderCreate functions for each Recorder instance to enable the Recorder feature.
- In the main application loop, call the FMSTR_Poll API function periodically when the application is idle.
- For the FMSTR_SHORT_INTR and FMSTR_LONG_INTR modes, enable the interrupts globally so that the interrupts can be handled by the CPU.

Communication troubleshooting The most common problem that causes communication issues is a wrong baud rate setting or a wrong pin multiplexer setting of the target MCU. When a communication between the PC Host running FreeMASTER and the target MCU cannot be established, try enabling the FMSTR_DEBUG_TX option in the *freemaster_cfg.h* file and call the FMSTR_Poll function periodically in the main application task loop.

With this feature enabled, the FreeMASTER driver periodically transmits a test frame through the Serial or CAN lines. Use a logic analyzer or an oscilloscope to monitor the signals at the communication pins of the CPU device to examine whether the bit rate and signal polarity are configured properly.

Driver API

This section describes the driver Application Programmers' Interface (API) needed to initialize and use the FreeMASTER serial communication driver.

Control API There are three key functions to initialize and use the driver.

FMSTR_Init

Prototype

```
FMSTR_BOOL FMSTR_Init(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description This function initializes the internal variables of the FreeMASTER driver and enables the communication interface. This function does not change the configuration of the selected communication module. The hardware module must be initialized before the *FMSTR_Init* function is called.

A call to this function must occur before calling any other FreeMASTER driver API functions.

FMSTR_Poll

Prototype

```
void FMSTR_Poll(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_protocol.c*

Description In the poll-driven or short interrupt modes, this function handles the protocol decoding and execution (see *Driver interrupt modes*). In the poll-driven mode, this function also handles the communication interface with the PC. Typically, the *FMSTR_Poll* function is called during the "idle" time in the main application task loop.

To prevent the receive data overflow (loss) on a serial interface, make sure that the FMSTR_Poll function is called at least once per the time calculated as:

*N * Tchar*

where:

- *N* is equal to the length of the receive FIFO queue (configured by the FMSTR_COMM_RQUEUE_SIZE macro). *N* is 1 for the poll-driven mode.
- *Tchar* is the character time, which is the time needed to transmit or receive a single byte over the SCI line.

Note: In the long interrupt mode, this function typically compiles as an empty function and can still be called. It is worthwhile to call this function regardless of the interrupt mode used in the application. This approach enables a convenient switching between the different interrupt modes only by changing the configuration macros in the *freemaster_cfg.h* file.

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

```
void FMSTR_SerialIsr(void);
void FMSTR_CanIsr(void);
```

- Declaration: *freemaster.h*
- Implementation: *hw-specific low-level driver C file*

Description This function contains the interrupt-processing code of the FreeMASTER driver. In long or short interrupt modes (see *Driver interrupt modes*), this function must be called from the application interrupt service routine registered for the communication interrupt vector. On platforms where the communication module uses multiple interrupt vectors, the application should register a handler for all vectors and call this function at each interrupt.

Note: In a poll-driven mode, this function is compiled as an empty function and does not have to be used.

Recorder API

FMSTR_RecorderCreate

Prototype

```
FMSTR_BOOL FMSTR_RecorderCreate(FMSTR_INDEX recIndex, FMSTR_REC_BUFF* buffCfg);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function registers a recorder instance and enables it to be used by the PC Host tool. Call this function for all recorder instances from 0 to the maximum number defined by the FMSTR_USE_RECORDER configuration option (minus one). An exception to this requirement is the recorder of instance *0* which may be automatically configured by FMSTR_Init when the *freemaster_cfg.h* configuration file defines the *FMSTR_REC_BUFF_SIZE* and *FMSTR_REC_TIMEBASE* options.

For more information, see *Configurable items*.

FMSTR_Recorder

Prototype

```
void FMSTR_Recorder(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function takes a sample of the variables being recorded using the FreeMASTER Recorder instance *recIndex*. If the selected Recorder is not active when the *FMSTR_Recorder* function is being called, the function returns immediately. When the Recorder is active, the values of the variables being recorded are copied into the recorder buffer and the trigger conditions are evaluated.

If a trigger condition is satisfied, the Recorder enters the post-trigger mode, where it counts down the follow-up samples (number of *FMSTR_Recorder* function calls) and de-activates the Recorder when the required post-trigger samples are finished.

The *FMSTR_Recorder* function is typically called in the timer or PWM interrupt service routines. This function can also be called in the application main loop (for testing purposes).

FMSTR_RecorderTrigger

Prototype

```
void FMSTR_RecorderTrigger(FMSTR_INDEX recIndex);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_rec.c*

Description This function forces the Recorder trigger condition to happen, which causes the Recorder to be automatically deactivated after the post-trigger samples are sampled. Use this function in the application code for programmatic control over the Recorder triggering. This can be useful when a more complex triggering conditions need to be used.

Fast Recorder API The Fast Recorder feature is not available in the FreeMASTER driver version 3. This feature was heavily dependent on the target platform and it was only available for the 56F8xxxx DSCs.

TSA Tables When the TSA is enabled in the FreeMASTER driver configuration file (by setting the FMSTR_USE_TSA macro to a non-zero value), it defines the so-called TSA tables in the application. This section describes the macros that must to be used to define the TSA tables.

There can be any number of TSA tables spread across the application source files. There must be always exactly one TSA Table List defined, which informs the FreeMASTER driver about the active TSA tables.

When there is at least one TSA table and one TSA Table List defined in the application, the TSA information automatically appears in the FreeMASTER symbols list. The symbols can then be used to create FreeMASTER variables for visualization or control.

TSA table definition The TSA table describes the static or global variables together with their address, size, type, and access-protection information. If the TSA-described variables are of a structure type, the TSA table may also describe this type and provide an access to the individual structure members of the variable.

The TSA table definition begins with the FMSTR_TSA_TABLE_BEGIN macro with a *table_id* identifying the table. The *table_id* shall be a valid C-langiage symbol.

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

After this opening macro, the TSA descriptors are placed using these macros:

```
/* Adding variable descriptors */
FMSTR_TSA_RW_VAR(name, type)  /* read/write variable entry */
FMSTR_TSA_RO_VAR(name, type)  /* read-only variable entry */

/* Description of complex data types */
FMSTR_TSA_STRUCT(struct_name) /* structure or union type entry */
```

(continues on next page)

(continued from previous page)

```
FMSTR_TSA_MEMBER(struct_name, member_name, type)  /* structure member entry */

/* Memory blocks */
FMSTR_TSA_RW_MEM(name, type, address, size) /* read/write memory block */
FMSTR_TSA_RO_MEM(name, type, address, size) /* read-only memory block */
```

The table is closed using the FMSTR_TSA_TABLE_END macro:

```
FMSTR_TSA_TABLE_END()
```

TSA descriptor parameters The TSA descriptor macros accept these parameters:

- *name* — variable name. The variable must be defined before the TSA descriptor references it.
- *type* — variable or member type. Only one of the pre-defined type constants may be used (see below).
- *struct_name* — structure type name. The type must be defined (typedef) before the TSA descriptor references it.
- *member_name* — structure member name.

Note: The structure member descriptors (FMSTR_TSA_MEMBER) must immediately follow the parent structure descriptor (FMSTR_TSA_STRUCT) in the table.

Note: To write-protect the variables in the FreeMASTER driver (FMSTR_TSA_RO_VAR), enable the TSA-Safety feature in the configuration file.

TSA variable types The table lists *type* identifiers which can be used in TSA descriptors:

Constant	Description
FMSTR_TSA_UINT*n*	Unsigned integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_SINT*n*	Signed integer type of size *n* bits (n=8,16,32,64)
FMSTR_TSA_FRAC*n*	Fractional number of size *n* bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(*m*,*n*)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(*m*,*n*)	Unsigned fractional number in general UQ form (m+n total bits)
FMSTR_TSA_FLOAT	4-byte standard IEEE floating-point type
FMSTR_TSA_DOUBLE	8-byte standard IEEE floating-point type
FMSTR_TSA_POINTER	Generic pointer type defined (platform-specific 16 or 32 bit)
FM-STR_TSA_USERTYPE(*name*)	Structure or union type declared with FMSTR_TSA_STRUCT record

TSA table list There shall be exactly one TSA Table List in the application. The list contains one entry for each TSA table defined anywhere in the application.

The TSA Table List begins with the FMSTR_TSA_TABLE_LIST_BEGIN macro and continues with the TSA table entries for each table.

```
FMSTR_TSA_TABLE_LIST_BEGIN()

FMSTR_TSA_TABLE(table_id)
FMSTR_TSA_TABLE(table_id2)
FMSTR_TSA_TABLE(table_id3)
...
```

The list is closed with the FMSTR_TSA_TABLE_LIST_END macro:

```
FMSTR_TSA_TABLE_LIST_END()
```

TSA Active Content entries FreeMASTER v2.0 and higher supports TSA Active Content, enabling the TSA tables to describe the memory-mapped files, virtual directories, and URL hyperlinks. FreeMASTER can access such objects similarly to accessing the files and folders on the local hard drive.

With this set of TSA entries, the FreeMASTER pages can be embedded directly into the target MCU flash and accessed by FreeMASTER directly over the communication line. The HTML-coded pages rendered inside the FreeMASTER window can access the TSA Active Content resources using a special URL referencing the *fmstr:* protocol.

This example provides an overview of the supported TSA Active Content entries:

```
FMSTR_TSA_TABLE_BEGIN(files_and_links)

/* Directory entry applies to all subsequent MEMFILE entries */
FMSTR_TSA_DIRECTORY("/text_files")     /* entering a new virtual directory */

/* The readme.txt file will be accessible at the fmstr://text_files/readme.txt URL */
FMSTR_TSA_MEMFILE("readme.txt", readme_txt, sizeof(readme_txt)) /* memory-mapped file */

/* Files can also be specified with a full path so the DIRECTORY entry does not apply */
FMSTR_TSA_MEMFILE("/index.htm", index, sizeof(index))          /* memory-mapped file */
FMSTR_TSA_MEMFILE("/prj/demo.pmp", demo_pmp, sizeof(demo_pmp)) /* memory-mapped file */

/* Hyperlinks can point to a local MEMFILE object or to the Internet */
FMSTR_TSA_HREF("Board's Built-in Welcome Page", "/index.htm")
FMSTR_TSA_HREF("FreeMASTER Home Page", "http://www.nxp.com/freemaster")

/* Project file links simplify opening the projects from any URLs */
FMSTR_TSA_PROJECT("Demonstration Project (embedded)", "/prj/demo.pmp")
FMSTR_TSA_PROJECT("Full Project (online)", "http://mycompany.com/prj/demo.pmp")

FMSTR_TSA_TABLE_END()
```

TSA API

FMSTR_SetUpTsaBuff

Prototype

```
FMSTR_BOOL FMSTR_SetUpTsaBuff(FMSTR_ADDR buffAddr, FMSTR_SIZE buffSize);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_tsa.c*

Arguments

- *buffAddr* [in] - address of the memory buffer for the dynamic TSA table
- *buffSize* [in] - size of the memory buffer which determines the maximum number of TSA entries to be added in the runtime

Description This function must be used to assign the RAM memory buffer to the TSA subsystem when FMSTR_USE_TSA_DYNAMIC is enabled. The memory buffer is then used to store the TSA entries added dynamically to the runtime TSA table using the FMSTR_TsaAddVar function call. The runtime TSA table is processed by the FreeMASTER PC Host tool along with all static tables as soon as the communication port is open.

The size of the memory buffer determines the number of TSA entries that can be added dynamically. Depending on the MCU platform, one TSA entry takes either 8 or 16 bytes.

FMSTR_TsaAddVar

Prototype

```
FMSTR_BOOL FMSTR_TsaAddVar(FMSTR_TSATBL_STRPTR tsaName, FMSTR_TSATBL_STRPTR
↪tsaType,
        FMSTR_TSATBL_VOIDPTR varAddr, FMSTR_SIZE32 varSize,
        FMSTR_SIZE flags);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_tsa.c*

Arguments

- *tsaName* [in] - name of the object
- *tsaType* [in] - name of the object type
- *varAddr* [in] - address of the object
- *varSize* [in] - size of the object
- *flags* [in] - access flags; a combination of these values:
  - *FMSTR_TSA_INFO_RO_VAR* — read-only memory-mapped object (typically a variable)
  - *FMSTR_TSA_INFO_RW_VAR* — read/write memory-mapped object
  - *FMSTR_TSA_INFO_NON_VAR* — other entry, describing structure types, structure members, enumerations, and other types

Description This function can be called only when the dynamic TSA table is enabled by the FMSTR_USE_TSA_DYNAMIC configuration option and when the FMSTR_SetUpTsaBuff function call is made to assign the dynamic TSA table memory. This function adds an entry into the dynamic TSA table. It can be used to register a read-only or read/write memory object or describe an item of the user-defined type.

See *TSA table definition* for more details about the TSA table entries.

Application Commands API

FMSTR_GetAppCmd

Prototype

```
FMSTR_APPCMD_CODE FMSTR_GetAppCmd(void);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Description This function can be used to detect if there is an Application Command waiting to be processed by the application. If no command is pending, this function returns the FMSTR_APPCMDRESULT_NOCMD constant. Otherwise, this function returns the code of the Application Command that must be processed. Use the FMSTR_AppCmdAck call to acknowledge the Application Command after it is processed and to return the appropriate result code to the host.

The FMSTR_GetAppCmd function does not report the commands for which a callback handler function exists. If the FMSTR_GetAppCmd function is called when a callback-registered command is pending (and before it is actually processed by the callback function), this function returns FMSTR_APPCMDRESULT_NOCMD.

FMSTR_GetAppCmdData

Prototype

```
FMSTR_APPCMD_PDATA FMSTR_GetAppCmdData(FMSTR_SIZE* dataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *dataLen* [out] - pointer to the variable that receives the length of the data available in the buffer. It can be NULL when this information is not needed.

Description This function can be used to retrieve the Application Command data when the application determines that an Application Command is pending (see *FMSTR_GetAppCmd*).

There is just a single buffer to hold the Application Command data (the buffer length is FMSTR_APPCMD_BUFF_SIZE bytes). If the data are to be used in the application after the command is processed by the FMSTR_AppCmdAck call, copy the data out to a private buffer.

FMSTR_AppCmdAck

Prototype

```
void FMSTR_AppCmdAck(FMSTR_APPCMD_RESULT resultCode);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultCode* [in] - the result code which is to be returned to FreeMASTER

Description This function is used when the Application Command processing finishes in the application. The resultCode passed to this function is returned back to the host and the driver is re-initialized to expect the next Application Command.

After this function is called and before the next Application Command arrives, the return value of the FMSTR_GetAppCmd function is FMSTR_APPCMDRESULT_NOCMD.

FMSTR_AppCmdSetResponseData

Prototype

```
void FMSTR_AppCmdSetResponseData(FMSTR_ADDR resultDataAddr, FMSTR_SIZE resultDataLen);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *resultDataAddr* [in] - pointer to the data buffer that is to be copied to the Application Command data buffer
- *resultDataLen* [in] - length of the data to be copied. It must not exceed the FMSTR_APPCMD_BUFF_SIZE value.

Description This function can be used before the Application Command processing finishes, when there are data to be returned back to the PC.

The response data buffer is copied into the Application Command data buffer, from where it is accessed when the host requires it. Do not use FMSTR_GetAppCmdData and the data buffer after FMSTR_AppCmdSetResponseData is called.

Note: The current version of FreeMASTER does not support the Application Command response data.

FMSTR_RegisterAppCmdCall

Prototype

```
FMSTR_BOOL FMSTR_RegisterAppCmdCall(FMSTR_APPCMD_CODE appCmdCode, FMSTR_
↪PAPPCMDFUNC callbackFunc);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_appcmd.c*

Arguments

- *appCmdCode* [in] - the Application Command code for which the callback is to be registered
- *callbackFunc* [in] - pointer to the callback function that is to be registered. Use NULL to unregister a callback registered previously with this Application Command.

Return value This function returns a non-zero value when the callback function was successfully registered or unregistered. It can return zero when trying to register a callback function for more than FMSTR_MAX_APPCMD_CALLS different Application Commands.

Description This function can be used to register the given function as a callback handler for the Application Command. The Application Command is identified using single-byte code. The callback function is invoked automatically by the FreeMASTER driver when the protocol decoder obtains a request to get the application command result code.

The prototype of the callback function is

```
FMSTR_APPCMD_RESULT HandlerFunction(FMSTR_APPCMD_CODE nAppcmd,
        FMSTR_APPCMD_PDATA pData, FMSTR_SIZE nDataLen);
```

Where:

- *nAppcmd* -Application Command code
- *pData* —points to the Application Command data received (if any)
- *nDataLen* —information about the Application Command data length

The return value of the callback function is used as the Application Command Result Code and returned to FreeMASTER.

Note: The FMSTR_MAX_APPCMD_CALLS configuration macro defines how many different Application Commands may be handled by a callback function. When FMSTR_MAX_APPCMD_CALLS is undefined or defined as zero, the FMSTR_RegisterAppCmdCall function always fails.

Pipes API

FMSTR_PipeOpen

Prototype

```
FMSTR_HPIPE FMSTR_PipeOpen(FMSTR_PIPE_PORT pipePort, FMSTR_PPIPEFUNC pipeCallback,
↪
        FMSTR_ADDR pipeRxBuff, FMSTR_PIPE_SIZE pipeRxSize,
        FMSTR_ADDR pipeTxBuff, FMSTR_PIPE_SIZE pipeTxSize,
        FMSTR_U8 type, const FMSTR_CHAR *name);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipePort* [in] - port number that identifies the pipe for the client
- *pipeCallback* [in] - pointer to the callback function that is called whenever a pipe data status changes
- *pipeRxBuff* [in] - address of the receive memory buffer
- *pipeRxSize* [in] - size of the receive memory buffer
- *pipeTxBuff* [in] - address of the transmit memory buffer
- *pipeTxSize* [in] - size of the transmit memory buffer
- *type* [in] - a combination of FMSTR_PIPE_MODE_xxx and FMSTR_PIPE_SIZE_xxx constants describing primary pipe data format and usage. This type helps FreeMASTER decide how to access the pipe by default. Optional, use 0 when undetermined.
- *name* [in] - user name of the pipe port. This name is visible to the FreeMASTER user when creating the graphical pipe interface.

Description This function initializes a new pipe and makes it ready to accept or send the data to the PC Host client. The receive memory buffer is used to store the received data before they are read out by the FMSTR_PipeRead call. When this buffer gets full, the PC Host client denies the data transmission into this pipe until there is enough free space again. The transmit memory buffer is used to store the data transmitted by the application to the PC Host client using the FMSTR_PipeWrite call. The transmit buffer can get full when the PC Host is disconnected or when it is slow in receiving and reading out the pipe data.

The function returns the pipe handle which must be stored and used in the subsequent calls to manage the pipe object.

The callback function (if specified) is called whenever new data are received through the pipe and available for reading. This callback is also called when the data waiting in the transmit buffer are successfully pushed to the PC Host and the transmit buffer free space increases. The prototype of the callback function provided by the user application must be as follows. The *PipeHandler* name is only a placeholder and must be defined by the application.

```
void PipeHandler(FMSTR_HPIPE pipeHandle);
```

FMSTR_PipeClose

Prototype

```
void FMSTR_PipeClose(FMSTR_HPIPE pipeHandle);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call

Description This function de-initializes the pipe object. No data can be received or sent on the pipe after this call.

FMSTR_PipeWrite

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeWrite(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
        FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE writeGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data to be written
- *pipeDataLen* [in] - length of the data to be written
- *writeGranularity* [in] - size of the minimum unit of data which is to be written

Description This function puts the user-specified data into the pipe's transmit memory buffer and schedules it for transmission. This function returns the number of bytes that were successfully written into the buffer. This number may be smaller than the number of the requested bytes if there is not enough free space in the transmit buffer.

The *writeGranularity* argument can be used to split the data into smaller chunks, each of the size given by the *writeGranularity* value. The FMSTR_PipeWrite function writes as many data chunks as possible into the transmit buffer and does not attempt to write an incomplete chunk.

This feature can prove to be useful to avoid the intermediate caching when writing an array of integer values or other multi-byte data items. When making the nGranularity value equal to the nLength value, all data are considered as one chunk which is either written successfully as a whole or not at all. The nGranularity value of 0 or 1 disables the data-chunk approach.

FMSTR_PipeRead

Prototype

```
FMSTR_PIPE_SIZE FMSTR_PipeRead(FMSTR_HPIPE pipeHandle, FMSTR_ADDR pipeData,
        FMSTR_PIPE_SIZE pipeDataLen, FMSTR_PIPE_SIZE readGranularity);
```

- Declaration: *freemaster.h*
- Implementation: *freemaster_pipes.c*

Arguments

- *pipeHandle* [in] - pipe handle returned from the FMSTR_PipeOpen function call
- *pipeData* [in] - address of the data buffer to be filled with the received data
- *pipeDataLen* [in] - length of the data to be read
- *readGranularity* [in] - size of the minimum unit of data which is to be read

Description This function copies the data received from the pipe from its receive buffer to the user buffer for further processing. The function returns the number of bytes that were successfully copied to the buffer. This number may be smaller than the number of the requested bytes if there is not enough data bytes available in the receive buffer.

The readGranularity argument can be used to copy the data in larger chunks in the same way as described in the FMSTR_PipeWrite function.

API data types This section describes the data types used in the FreeMASTER driver. The information provided here can be useful when modifying or porting the FreeMASTER Communication Driver to new NXP platforms.

Note: The licensing conditions prohibit use of FreeMASTER and the FreeMASTER Communication Driver with non-NXP MPU or MCU products.

Public common types The table below describes the public data types used in the FreeMASTER driver API calls. The data types are declared in the *freemaster.h* header file.

Type name	Description
FMSTR_ADDR	Data type used to hold the memory address. On most platforms, this is normally a C-pointer, but it may also be a pure integer type.
For example, this type is defined as long integer on the 56F8xxx platform where the 24-bit addresses must be supported, but the C-pointer may be only 16 bits wide in some compiler configurations.	
FMSTR_SIZE	Data type used to hold the memory block size.
It is required that this type is unsigned and at least 16 bits wide integer.	
FMSTR_BOOL	Data type used as a general boolean type.
This type is used only in zero/non-zero conditions in the driver code.	
FMSTR_APPCM	Data type used to hold the Application Command code.
Generally, this is an unsigned 8-bit value.	
FMSTR_APPCM	Data type used to create the Application Command data buffer.
Generally, this is an unsigned 8-bit value.	
FMSTR_APPCM	Data type used to hold the Application Command result code.
Generally, this is an unsigned 8-bit value.	

Public TSA types The table describes the TSA-specific public data types. These types are declared in the *freemaster_tsa.h* header file, which is included in the user application indirectly by the *freemaster.h* file.

FMSTR_TSA_TI	Data type used to hold a descriptor index in the TSA table or a table index in the list of TSA tables.
By default, this is defined as FMSTR_SIZE.	
FMSTR_TSA_TS	Data type used to hold a memory block size, as used in the TSA descriptors.
By default, this is defined as FMSTR_SIZE.	

Public Pipes types The table describes the data types used by the FreeMASTER Pipes API:

FMSTR_HPIPE	Pipe handle that identifies the open-pipe object.
Generally, this is a pointer to a void type.	
FMSTR_PIPE_P	Integer type required to hold at least 7 bits of data.
Generally, this is an unsigned 8-bit or 16-bit type.	
FMSTR_PIPE_SI	Integer type required to hold at least 16 bits of data.
This is used to store the data buffer sizes.	
FMSTR_PPIPEF	Pointer to the pipe handler function.
See *FMSTR_PipeOpen* for more details.	

Internal types The table describes the data types used internally by the FreeMASTER driver. The data types are declared in the platform-specific header file and they are not available in the application code.

FMSTR_U8	The smallest memory entity.
On the vast majority of platforms, this is an unsigned 8-bit integer.	
On the 56F8xx DSP platform, this is defined as an unsigned 16-bit integer.	
FMSTR_U16	Unsigned 16-bit integer.
FMSTR_U32	Unsigned 32-bit integer.
FMSTR_S8	Signed 8-bit integer.
FMSTR_S16	Signed 16-bit integer.
FMSTR_S32	Signed 32-bit integer.
FMSTR_FLOAT	4-byte standard IEEE floating-point type.
FMSTR_FLAGS	Data type forming a union with a structure of flag bit-fields.
FMSTR_SIZE8	Data type holding a general size value, at least 8 bits wide.
FMSTR_INDEX	General for-loop index. Must be signed, at least 16 bits wide.
FMSTR_BCHR	A single character in the communication buffer.
Typically, this is an 8-bit unsigned integer, except for the DSP platforms where it is a 16-bit integer.	
FMSTR_BPTR	A pointer to the communication buffer (an array of FMSTR_BCHR).

Document references

Links

- This document online: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/middleware/freemaster/doc/index.html

- FreeMASTER tool home: www.nxp.com/freemaster
- FreeMASTER community area: community.nxp.com/community/freemaster
- FreeMASTER GitHub code repo: https://github.com/nxp-mcuxpresso/mcux-freemaster
- MCUXpresso SDK home: www.nxp.com/mcuxpresso
- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Documents

- *FreeMASTER Usage Serial Driver Implementation* (document AN4752)
- *Integrating FreeMASTER Time Debugging Tool With CodeWarrior For Microcontrollers v10.X Project* (document AN4771)
- *Flash Driver Library For MC56F847xx And MC56F827xx DSC Family* (document AN4860)

Revision history This Table summarizes the changes done to this document since the initial release.

Revision	Date	Description
1.0	03/2006	Limited initial release
2.0	09/2007	Updated for FreeMASTER version. New Freescale document template used.
2.1	12/2007	Added description of the new Fast Recorder feature and its API.
2.2	04/2010	Added support for MPC56xx platform, Added new API for use CAN interface.
2.3	04/2011	Added support for Kxx Kinetis platform and MQX operating system.
2.4	06/2011	Serial driver update, adds support for USB CDC interface.
2.5	08/2011	Added Packet Driven BDM interface.
2.7	12/2013	Added FLEXCAN32 interface, byte access and isr callback configuration option.
2.8	06/2014	Removed obsolete license text, see the software package content for up-to-date license.
2.9	03/2015	Update for driver version 1.8.2 and 1.9: FreeMASTER Pipes, TSA Active Content, LIN Transport Layer support, DEBUG-TX communication troubleshooting, Kinetis SDK support.
3.0	08/2016	Update for driver version 2.0: Added support for MPC56xx, MPC57xx, KEAxx and S32Kxx platforms. New NXP document template as well as new license agreement used. added MCAN interface. Folders structure at the installation destination was rearranged.
4.0	04/2019	Update for driver released as part of FreeMASTER v3.0 and MCUXpresso SDK 2.6. Updated to match new V4 serial communication protocol and new configuration options. This version of the document removes substantial portion of outdated information related to S08, S12, ColdFire, Power and other legacy platforms.
4.1	04/2020	Minor update for FreeMASTER driver included in MCUXpresso SDK 2.8.
4.2	09/2020	Added example applications description and information about the MCUXpresso Config Tools. Fixed the pipe-related API description.
4.3	10/2024	Added description of Network and Segger J-Link RTT interface configuration. Accompanying the MCUXpresso SDK version 24.12.00.
4.4	04/2025	Added Zephyr-specific information. Accompanying the MCUXpresso SDK version 25.06.00.

1.6 Wireless

1.6.1 NXP Wireless Framework and Stacks

Wireless Framework

Wireless Connectivity Framework Connectivity Framework repository provides both connectivity platform enablement with hardware abstraction layer and a set of Services for NXP connectivity stacks : BLE, Zigbee, OpenThread, Matter.

The connectivity framework repository consists of:

- Common folder to common header files for minimal type definition to be used in the repo
- Platform folder used for platform enablement with Hardware abstraction:
  - platform/include: common API header files used by several platforms
  - platform/common: common code for several platforms
  - specifics platform folders , See below the supported platform list
  - platform/../configs folder: configuration files for framework repository and other middlewares (rpmsg, mbedTls, etc.._)
- Services folder
- Zephyr folder for zephyr modules integrated in mcux SDK
- clang formatting script and script folder to format appropriately the source files of the repo

Supported platforms The following devices/platforms are supported in platform folder for connectivity applications:

- kw45x, k32w1x, mcxw71x, under wireless_mcu, kw45_k32w1_mcxw71 folders.
- kw47x, mcxw72x families under wireless_mcu, kw47_mcxw72, kw47_mcxw72_nbu folders.
- rw61x
- RT1060 and RT1170 for Matter
- Other RT devices such as i.MX RT595s

Supported services The supported services are provided for connectivity stacks and their demo application, and are usually dependent on PLATFORM API implementation:

- DBG: Light Debug Module, currently a stubbed header file
- FSCI: Framework Serial Communication Interface between BLE host stack and upper layer located on an other core/device
- FunctionLib: wrapper to toolchain memory manipulation functions (memcpy, memcmp, etc) or use its own implementation for code size reduction
- HWParameters: Store Factory hardware parameters and Application parameters in Flash or IFR
- LowPower: wrapper of SDK power manager for connectivity applications
- ModuleInfo: Store and handle connectivity component versions
- NVM: NXP proprietary File System used for KW45, KW47 automotive devices and RT1060/RT1170 platform for Matter
- OtaSupport: Handle OTA binary writes into internal or external flash.
- SecLib and RNG: Crypto and Random Number generator functions. It supports several ports:
  - Software algorithms
  - Secure subsystem interface to an HW enclave
  - MbedTls 2.x interface
- Sensors: Provides service for Battery and temperature measurements

- SFC: Smart Frequency Calibration to be run from KW47/MCXW71 from NBU core. Matter related modules:
- OTW: Over The Wire module for External Transceiver firmware update from RT platforms
- FactoryDataProvider to be used for Matter

Supported Zephyr modules integration in mcux SDK Connectivity framework provides integration and port layers to the following Zephyr Modules located into zephyr/subsys:

- NVS: Zephyr File System used by Matter and Zigbee
- Settings: Over layer module that allows to store keys into NVS File System used by Matter Port layer and required libraries for these zephyr modules are located in port and lib folder in zephyr directory

Connectivity framework CHANGELOG

7.0.3 revA mcux SDK 25.06.00

Major Changes

- [wireless_nbu] Enhanced XTAL32M trimming handling: updates are applied when requested by the application core and the NBU enters low-power mode, ensuring no interference from ongoing radio activity. Introduced new APIs to lock (PLATFORM_LockXtal32MTrim()) and unlock XTAL32M (PLATFORM_UnlockXtal32MTrim()) trimming updates using a counter-based mechanism. Also added a reset API (PLATFORM_ResetContext()) for platform-specific variables (currently limited to the trimming lock).
- [wireless_mcu] Introduced a new API, PLATFORM_SetLdoCoreNormalDriveVoltage(), to enable support for NBU clock frequency at 64 MHz, as required by BLE channel sounding applications.
- [wireless_mcu][wireless_nbu] Increased delayLpoCycle default from 2 to 3 to address link layer instabilities in low-power NBU use cases. Adjusted BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0x10 to 0x16 to balance power consumption and stability. □ NBU may malfunction if delayLpoCycle (or BOARD_LL_32MHz_WAKEUP_ADVANCE_HSLOT) is set to 2 while BOARD_RADIO_DOMAIN_WAKE_UP_DELAY is 0x16.

Minor Changes (bug fixes)

- [WorkQ] Increased stack size when RNG use mbedtls port and coverage is enabled.
- [FSCI] Resolved an issue where messages remained unprocessed in the queue by ensuring OSA_EventSet() is triggered when pending messages are detected.
- [OTA] Fixed a bug in in OTA_PullImageChunk() that prevented retrieval of data previously received via OTA_PushImageChunk() when still buffered in RAM during posted operations.
- [OTA] Various MISRA and coverity fixes.
- [mcxw23] Fixed an unused variable warning in PLATFORM_RegisterNbuTemperatureRequestEventCb() API.
- [SFC] Remove obsolete flag gNbuJtagCapability.
- [wireless_mcu] Introduced new API PLATFORM_GetRadioIdleDuration32K(). Deprecated PLATFORM_CheckNextBleConnectivityActivity() API.

- [mcxw23] Aligned platform-specific implementations with the corresponding prototypes defined in wireless_mcu.
- [DBG] Cleaned up fwk_fault_handler.c.

7.0.2 RFP mcux SDK 25.06.00

Major Changes

- [wireless_mcu][wireless_nbu] Introduced PLATFORM_Get32KTimeStamp() API, available on platforms that support it.
- [RNG] Switched to using a workqueue for scheduling seed generation tasks.
- [Sensors] Integrated workqueue to trigger temperature readings on periodic timer expirations.
- [wireless_nbu] Removed outdated configuration files from wireless_nbu/configs.
- [SecLib_RNG][PSA] Added a PSA-compliant implementation for SecLib_RNG. ▯ This is an experimental feature and should be used with caution.
- [wireless_mcu][wireless_nbu] Implemented PLATFORM_SendNBUXtal32MTrim() API to transmit XTAL32M trimming values to the NBU.

Minor Changes (bug fixes)

- [MWS] Migrated the Mobile Wireless Standard (MWS) service to the public repository. This service manages coexistence between connectivity protocols such as BLE, 802.15.4, and GenFSK.
- [HWParameter][NVM][SecLib_RNG][Sensors] Addressed various MISRA compliance issues across multiple modules.
- [Sensors] Applied a filtering mechanism to temperature data measured by the application core before forwarding it to the NBU, improving data reliability.
- [Common] Relocated the GetPowerOfTwoShift() function to a shared module for broader accessibility across components.
- [RNG] Resolved inconsistencies in RNG behavior when using the fsl_adapter_rng HAL by aligning it with other API implementations.
- [SecLib] Updated the AES CMAC block counter in AES_128_CMAC() and AES_128_CMAC_LsbFirstInput() to support data segments larger than 4KB.
- [SecLib] Utilized sss_sscp_key_object_free() with kSSS_keyObjFree_KeysStoreDefragment to avoid key allocation failures.
- [MCXW23] Removed redundant NVIC_SetPriority() call for the ctimer IRQ in the platform file, as it's already handled by the driver.
- [WorkQ] Increased workqueue stack size to accommodate RNG usage with mbedtls.
- [wireless_mcu][ot] Suppressed chip revision transmission when operating with nbu_15_4.
- [platform][mflash] Ensured proper address alignment for external flash reads in PLATFORM_ReadExternalFlash() when required by platform constraints.
- [RNG] Corrected reseed flag behavior in RNG_GetPseudoRandomData() after reaching gRngMaxRequests_d threshold.
- [platform][mflash] Fixed uninitialized variable issue in PLATFORM_ReadExternalFlash().

- [platform][wireless_nbu] Fixed an issue on KW47 where PLATFORM_InitFro192M incorrectly reads IFR1 from a hardcoded flash address (0x48000), leading to unstable FRO192M trimming. The function is now conditionally compiled for KW45 only.

7.0.2 revB mcux SDK 25.06.00

Major Changes

- [RNG][wireless_mcu][wireless_nbu] Rework RNG seeding on NBU request
- [wireless_mcu] [LowPower] Add gPlatformEnableFro6MCalLowpower_d macro to enable FRO6M frequency verification on exit of Low Power
  - add PLATFORM_StartFro6MCalibration() and PLATFORM_EndFro6MCalibration() new function for FRO6M calibration (6MHz or 2Mhz) on wake-up from low power mode.
  - Enabled by default in fwk_config.h
- [wireless_nbu][LowPower] Clear pending interrupt status of the systick before going in low-power - Reduce NBU active time
- [wireless_nbu] Fix impossibility to go to WFI in combo mode (15.4/BLE)
- [wireless_mcu] Implement XTAL32M temperature compensation mechanism. 2 new APIs:
  - PLATFORM_RegisterXtal32MTempCompLut(): register the temperature compensation table for XTAL32M.
  - PLATFORM_CalibrateXtal32M(): apply XTAL32M temperature compensation depending on current temperature.
- [Sensors][wireless_mcu] Add support for periodic temperature measurement. new API:
  - SENSORS_TriggerTemperatureMeasurementUnsafe(): to be called from Interrupt masked critical section, from ISR or when scheduler is stopped
- [SFC] Change default maximal ppm target of the SFC algorithm from 200 to 360ppm. Impact the SFC algorith of kw45 and mcxw71 platforms, 360ppm was already the default setting for kw47 and mcxw72 platforms

Minor Changes (bug fixes)

- [DBG] Fix FWK_DBG_PERF_DWT_CYCLE_CNT_STOP macro
- [wireless_nbu] Add gPlatformIsNbu_d compile Macro set to 1
- [wireless_nbu][ics] gFwkSrvHostChipRevision_c can be processed in the system workqueue
- [kw45_mcxw71][kw47_mcxw72]
  - Remove LTC dependency from platform in kconfig
  - gPlatformShutdownEccRamInLowPower moved from fwk_platform_definition.h to fwk_confg.h as this is a configuration flag.
- [wireless_mcu][sensors] Rework and remove unnecessary ADC APIs
- [wireless_nbu] Add PLATFORM_GetMCUUid() function from Chip UID
- [SecLib] Change AES_MMO_BlockUpdate() function from private to public for zigbee.

7.0.2 revA mcux SDK 25.06.00

Supported platforms:

- Same as 25.03.00 release

Major Changes

- [KW45/MCXW71] HW parameters placement now located in IFR section. Flash storage is not longer used:
  - Compilation: Macro gHwParamsProdDataPlacement_c changed from gHwParamsProdDataMainFlash2IfrMode_c to gHwParamsProdDataIfrMode_c
- [KW47] NBU: Add new fwk_platform_dcdc.[ch] files to allow DCDC stepping by using SPC high power mode. This requires new API in board_dcdc.c files. Please refer to new compilation MACROs gBoardDcdcRampTrim_c and gBoardDcdcEnableHighPowerModeOnNbu_d in board_platform.h files located in kw47evk, kw47loc, frdmmcxw72 board folders.
- [KW45/MCXW71/KW47/MCXW72] Trigger an interrupt each time App core calls PLATFORM_RemoteActiveReq() to access NBU power domain in order to restart NBU core for domain low power process

Minor Changes (bug fixes)

Services

- [SecLib_RNG]
  - Rename mSecLibMutexId mutex to mSecLibSssMutexId in SecLib_sss.c
  - Remove MEM_TRACKING flag from RNG.c
  - Implement port to fsl_adapter_rng.h API using gRngUseRngAdapter_c compil Macro from RNG.c
  - Add support for BLE debug Keys in SecLi and SecLin_sss.c with gSecLibUseBleDebugKeys_d - for Debug only
- [FSCI] Add queue mechanism to prevent corruption of FSCI global variableAllow the application to override the trig sample number parameter when gFsciOverRpmsg_c is set to 1
- [DBG][btsnoop] Add a mechanism to dump raw HCI data via UART using SBTSNOOP_MODE_RAW
- [OTA]
  - OtaInternalFlash.c: Take into account chunks smaller than a flash phrase worth
  - fwk_platform_ot.c: dependencies and include files to gpio, port, pin_mux removed

Platform specific

- [kw45_mcxw71][kw47_mcxw72]
  - fwk_platform_reset.h : add compil Macro gUseResetByLvdForce_c and gUseResetByDeepPowerDown_c to avoid compile the code if not supported on some platforms
  - New compile Flag gPlatformHasNbu_d
  - Rework FRO32K notification service for MISRA fix

7.0.1 RFP mcux SDK 25.03.00 **Supported platforms:**

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170
- MCXW23

Minor Changes (bug fixes)

- [General] Various MISRA/Coverity fixes in framework: NVM, RNG, LowPower, SecLib and platform files

Services

- [SecLib_RNG] fix return status from RNG_GetTrueRandomNumber() function: return correctly gRngSuccess_d when RNG_entropy_func() function is successful
- [SFC] Allow the application to override the trig sample number parameter
- [Settings] Re-define the framework settings API name to avoid double definition when gSettingsRedefineApiName_c flag is defined

Platform specific

- [wireless_mcu] fwk_platform_sensors update :
  - Enable temperature measurement over ADC ISR
  - Enable temperature handling requested by NBU
- [wireless_mcu] fwk_platform_lcl coex config update for KW45
- [kw47_mcxw72] Change the default ppm_target of SFC algorithm from 200 to 360ppm

7.0.1 revB mcux SDK 25.03.00

Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170
- MCXW23

Minor Changes (bug fixes)

General

- [General] Various MISRA/Coverity fixes in framework: NVM, RNG, LowPower, FunctionLib and platform files

Services

- [SecLib_RNG] AES-CBC evolution:
  - added AES_CBC_Decrypt() API for sw, SSS and mbedtls variants.
  - Made AES-CBC SW implementation reentrant avoiding use of static storage of AES block.
  - fixed SSS version to update Initialization Vector within SecLib, simplifying caller's implementation.
  - modified AES_128_CBC_Encrypt_And_Pad() so as to avoid the constraint mandating that 16 byte headroom be available at end of input buffer.
- [SecLib_RNG] RNG modifications:
  - RNG_GetPseudoRandomData() could return 0 in some error cases where caller expected a negative status.

    * Explicited RNG error codes
    * Added argument checks for all APIs and return gRngBadArguments_d (-2) when wrong
    * added checks of RNG initalization and return gRngNotInitialized_d (-3) when not done
    * fixed correcteness of RNG_GetPrngFunc() and RNG_GetPrngContext() relative to API description.
    * Added RNG_DeInit() function mostly for test and coverage purposes.
    * Improved RNG description in README.md
    * Unified the APIs behaviour between mbedtls and non mbedtls variants.
  - RNG/mbedtls : Prevent RNG_Init() from corrupting RNG entropy context if called more than once.
  - RNG/mbedtls: fixed RNG_GetTrueRandomNumber() to return a proper mbedtls_entropy_func() result.
  - [SecLib_RNG] Use defragmetation option when freeing key object in SecLib_sss to avoid leak in S200 memory
  - [SecLib_RNG] Add new API ECP256_IsKeyValid() to check whether a public key is valid
  - [OtaSupport] Update return status to OTA_Flash_Success when success at the end of InternalFlash_WriteData() and InternalFlash_FlushWriteBuffer() APIs
  - [WorQ] Implementing a simple workqueue service to the framework
  - [SFC] Keep using immediate measurement for some measurement before switching to configuration trig to confirm the calibration made
  - [DBG] Adding modules to framework DBG :
    * sbtsnoop
    * SWO
  - [Common] Fix HAL_CTZ and HAL_RBIT IAR versions
  - [LowPower] Fix wrong tick error calculation in case of infinite timeout
  - [Settings] Add new macro gSettingsRedefineApiName_c to avoid multiple definition of settings API when using connectivity framework repo

Platform specific

- [KW47/MCXW72] Change xtal cload default value from 4 to 8 in order to increase the precision of the link layer timebase in NBU
- [wireless_mcu] [wireless_nbu] Use new WorkQ service to process framework intercore messages
- [rw61x] Fix HCI message sending failure in some corner case by releasing controller wakes up after that the host has send its HCI message
- [MCXW23] Adding the initial support of MCXW23 into the framework

7.0.0 mcux SDK 24.12.00

Supported platforms:

- KW45x, KW47x, MCXW71, MCXW72, K32W1x
- RW61x
- RT595, RT1060, RT1170

Minor Changes (bug fixes)

Platform specific

- [RW61X]
  - Add MCUX_COMPONENT_middleware.wireless.framework.platform.rng to the platform to fix a warning at generation
  - Retrieve IEEE 64 bits address from OTP memory
- [KW45x, MCXW71x, KW47x, MCXW72x]
  - Ignore the secure bit from RAM addresses when comparing used ram bank in bank retention mechanism
  - Add gPlatformNbuDebugGpioDAccessEnabled_d Compile Macro (enabled by default). Can be used to disable the NBU debug capability using IOs in case Trustzone is enabled ("PLATFORM_InitNbu()' code executed from unsecure world).
  - Fix in NBU firmware when sending ICS messages gFwkSrvNbuApiRequest_c (from controller_api.h API functions)

Services

- [OTA]
  - Add choice name to OtaSupport flash selection in Kconfig
- [NVM]
  - Add gNvmErasePartitionWhenFlashing_c feature support to gcc toolchain
- [SecLib_RNG]
  - Misra fixes

7.0.0 revB mcux SDK 24.12.00 Supported platforms: KW45x, KW47x, MCXW71, MCXW72, K32W1x, RW61x, RT595, RT1060, RT1170

Major Changes (User Applications may be impacted)

- mcux github support with cmake/Kconfig from sdk3 user shall now use CmakeLists.txt and Kconfig files from root folder. Compilation should be done using west build command. In order to see the Framework Kconfig, use command >west build -t guiconfig
- Board files and linker scripts moved to examples repository

Bugfixes

- [platform lowpower]
  - Entering Deep down power mode will no longer call PLATFORM_EnterPowerDown(). This API is now called only when going to Power down mode

Platform specific

- [KW47/MCXW72]: Early access release only
  - Deep sleep power mode not fully tested. User can experiment deep sleep and deep down modes using low power reference design applications

    - XTAL32K-less support using FRO32K not tested
- [KW45/MCXW71/K32W148]
    - Deep sleep mode is supported. Power down mode is supported in low power reference design applications as experimental only
    - XTAL32K-less support using FRO32K is experimental - FRO32K notifications callback is debug only and should not be used for mass production firmware builds

Minor Changes (no impact on application)

- Overall folder restructuring for SDK3
    - [Platform]:
        * Rename platform_family from connected_mcu/nbu to wireless_mcu/nbu
        * platform family have now a dedicated fwk_config.h, rpmsg_config.h and SecLib_mbedtls_config.h
    - [Services]
        * Move all framework services in a common directory "services/"

7.0.0 revA: KW45/KW47/MCX W71/MCX W72/K32W148

Experimental Features only

- Power down on application power domain: Some tests have shown some failure. Power consumption higher than Deep Sleep. => This feature is not fully supported in this release
- XTAL32K less board with FRO32K support: Some additional stress tests are under progress.
- FRO32K notifications callback is for debug only and shall not be used for production. User shall not execute long processing (such as PRINTF) as it is executed in ISR context.

Main Changes

- Cmake/Kconfig support for SDK3.0
- [Sensors] API renaming:
    - SENSORS_InitAdc() renamed to SENSORS_Init()
    - SENSORS_DeinitAdc() remamed to SENSORS_Deinit()
- [HWparams]
    - Repair PROD_DATA sector in case of ECC error (implies loss of previous contents of sector)
- [NVM] Linker script modification for armgcc whenever gNvTableKeptInRam_d option is used:
    - placement of NVM_TABLE_RW in data initialized section, providing start and end address symbols. For details see NVM_Interface.h comments.
- [OtaSupport]
    - OTA_Initialize(): now transitions the image state from RunCandidate to Permanent if not done by the application. OTA module shall always be initialized on a Permanent image, this change ensures it is the case.
    - OTA_MakeHeadRoomForNextBlock(): now erases the OTA partition up to the image total size (rounded to the sector) if known.

Minor changes

- [Platform]
  - Updated macro values: -kw47: BOARD_32MHZ_XTAL_CDAC_VALUE from 12U to 16U, BOARD_32MHZ_XTAL_ISEL_VALUE from 7U to 11U, BOARD_32KHZ_XTAL_CLOAD_DEFAULT from 8U to 4U, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT from 1U to 3U
    - MCX W72 (low-power reference design applications only): BOARD_32MHZ_XTAL_CDAC_VALUE from 12U to 10U, BOARD_32MHZ_XTAL_ISEL_VALUE from 7U to 11U, BOARD_32KHZ_XTAL_CLOAD_DEFAULT from 8U to 4U, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT from 1U to 3U
  - New PLATFORM_RegisterNbuTemperatureRequestEventCb() API: register a function callback when NBU request new temperature measurement. API provides the interval request for the temperature measurement
  - Update PLATFORM_IsNbuStarted() API to return true only if the NBU firmware has been started.
- [platform lowpower]
  - Move RAM layout values in fwk_platform_definition.h and update RAM retention API for KW47/MCXW72

Bugfixes

- [OtaSupport]
  - OTA_MakeHeadRoomForNextBlock(): fixed a case where the function could try to erase outside the OTA partition range.

6.2.4: KW45/K32W1x/MCXW71/RX61x SDK 2.16.100 This release does not contain the changes from 6.2.3 release.

This release contains changes from 6.2.2 release.

Main Change

- armgcc support for Cmake sdk2 support and VS code integration

Minor changes

- [NBU]
  - Optimize some critical sections on nbu firmware
- [Platform]
  - Optimize PLATFORM_RemoteActiveReq() execution time.

6.2.3: KW47 EAR1.0 Initial Connectivity Framework enablement for KW47 EAR1.0 support.

New features

- OpenNBU feature : nbu_ble project is available for modification and building

Supported features

- Deep sleep mode

Unsuported features

- Power down mode
- FRO32K support (XTAL32K less boards)

Main changes

- [NBU]
  - LPTMR2 available and TimerManager initialization with Compile Macro: gPlatformUseLptmr_d
  - NBU can now have access to GPIOD
  - SW RNG and SW SecLib ported to NBU (Software implementation only)
- [RNG]
  - Obsoleted API removed : FWK_RNG_DEPRECATED_API
  - RNG can be built without SecLib for NBU, using gRngUseSecLib_d in fwk_config.h
  - Some API updates:
    - RNG_IsReseedneeded() renamed to RNG_IsReseedNeeded,
    - RNG_TriggerReseed() renamed to RNG_NotifyReseedNeeded(),
    - RNG_SetSeed() and RNG_SetExternalSeed() return status code.
  - Optimized Linear Congruential modulus computation to reduce cycle count.

Minor changes

- [NVM]
  - Optimize NvIsRecordErased() procedure for faster garbage collection
  - MISRA fix : Remove externs and weaks from NVM module - Make RNG and timer manager dependencies conditional
- [Platform]
  - Allow the debugger to wakeup the KW47/MCXW72 target

6.2.2: KW45/K32W1 MR6 SDK 2.16.000 Experimental Features only:

- Power down on application power domain : Some tests have shown some failure. Power consumption higher than Deep Sleep. => This feature is not fully supported in this release
- XTAL32K less board with FRO32K support : Some additional stress tests are under progress.
- FRO32K notifications callback is for debug only and shall not be used for production. User shall not execute long processing (such as PRINTF) as it is executed in ISR context.

Changes

- [Board] Support for freedom board FRDM-MCX W7X
- [HWparams]
  - Support for location of HWParameters and Application Factory Data IFR in IFR1
  - Default is still to use HWparams in Flash to keep backward compatibility
- [RNG]: API updates:
  - New APIS RNG_IsReseedneeded(), RNG_SetSeed() to provide See to PRNG on NBU/App core - See BluetoothLEHost_ProcessIdleTask() in app_conn.c
  - New APIs RNG_SetExternalSeed() : User can provide external seed. Typically used on NBU firmrware for App core to set a seed to RNG. RNG_TriggerReseed() : Not required on App core. Used on NBU only.
- [NVS] Wear statistics counters added - Fix nvs_file_stat() function
- [NVM] fix Nv_Shutdown() API
- [SecLib] New feature AES MMO supported for Zigbee

6.2.2: RW61x RFP4 SDK 2.16.000

- [Platform] Support Zigbee stack
- [OTA] Add support for RW61x OTA with remap feature.
  - Required modifications to prevent direct access to flash logical addresses when remap is active.
  - Image trailers expected at different offset with remap enabled (see gPlatformMcuBootUseRemap_d in fwk_config.h)
  - fixed image state assessment procedure when in RunCandidate.
- [NVS] Wear statistics counters added
- [SecLib] New feature AES MMO supported for Zigbee
- [Misra] various fixes

6.2.1: KW45/K32W1 MR5 SDK 2.15.000

Experimental Features only:

- Power down on application power domain : Some tests have shown some failure. This feature is not fully supported in this release
- XTAL32K less board with FRO32K support : Some additional stress tests are under progress. Timing variation of the timebase are being analyzed

Major changes

- [RNG]: API updates
  - New compile flag to keep deprecated API: FWK_RNG_DEPRECATED_API
  - change return error code to int type for RNG_Init(), RNG_ReInit()
  - New APIs RNG_GetTrueRandomNumber(), RNG_GetPseudoRandomData()
- [Platform]
  - fwk_platform_sensors
    - Change default temperature value from -1 to 999999 when unknown

  - fwk_platform_genfsk
    * rename from platform_genfsk.c/h to fwk_platform_genfsk.c/h
  - platform family
    * Rename the framework platform folder from kw45_k32w1 to connected_mcu to support other platform from the same family
  - fwk_platform_intflash
    * Moved from fwk_platform files to the new fwk_platform_intflash files the internal flash dependant API
- [NBU]
  - BOARD_LL_32MHz_WAKEUP_ADVANCE_HSLOT changed from 2 to 3 by default
  - BOARD_RADIO_DOMAIN_WAKE_UP_DELAY changed from 0x10 to 0x0F
- [gcc linker]
  - Exclude k32w1_nbu_ble_15_4_dyn.bin from .data section

Minor Changes

- [Platform]
  - PLATFORM_GetTimeStamp(0 has an important fix for reading the Timestamp in TSTMR0
  - New API PLATFORM_TerminateCrypto(), PLATFORM_ResetCrypto() called from SecLib for lowpower exit
  - Fix when enable fro debug callback on nbu
- [DBG]
  - SWO
    * Add new files fwk_debug_swo.c/h to use SWO for debug purpose
    * Two new flags has been added:
      · BOARD_DBG_SWO_CORE_FUNNEL to chose on which core you want to use SWO
      · BOARD_DBG_SWO_PIN_ENABLE to enable SWO on a pin
- [NVS]
  - Add support of NVS and Settings in framework
- [NBU]
  - Fix power down issues and reduce critical section on NBU side:
    * new API PLATFORM_RemoteActiveReqWithoutDelay() called from NBU functions where waiting delay is not required
    * Increase delay needed in power down for OEM part to request the SOC to be active
    * Remove unnecessary code to PLATFORM_RemoteActiveReqWithoutDelay() from PLATFORM_HciRpmsgRxCallback()
    * Improve nbu memory allocation failure debug messages
- [SDK]
  - Multicore: remove critical section in HAL_RpmsgSendTimeout() (only required in FPGA HDI mode)
  - Flash drivers: update for ECC detection

- [Platform]
  - fwk_platform_sensors
    - Fix temperature reporting to NBU
  - fwk_platform_extflash
    - Align .c and .h prototype of PLATFORM_ExternalFlashAreaIsBlank() function
- [NVM]
  - Keep Mutex in NvModuleDeInit(). In Bare metal OS, Mutex can not be destroyed
  - New API NvRegisterEccFaultNotificationCb() to register Notification callback when Ecc error happens in FileSystem
- [MISRA] fixes
  - SecLib_sss.c: ECDH_P256_ComputeDhKey()
  - fwk_platform_extflash.c: PLATFORM_IsExternalFlashPageBlank()
  - fwk_fs_abstraction.c: Various fixes
- [HWparams]
  - Fix on if condition when gHwParamsProdDataPlacementLegacy2IfrMode_c mode is selected
- [OTA]
  - Enable gOtaCheckEccFaults_d by default to avoid bus in case of ECC error during OTA
  - Fix OTA partition overflow during OTA stop and resume transfer
- [BOARD]
  - Place code button or led specific under correct defines in board_comp.c/h
  - Bring back MACROs BOARD_INITRFSWITCHCONTROLPINS in pin_mux header file of the loc board
- [SecLib]
  - Add some undefinition in SecLib_mbedtls_config as new dependency has been added in mbedtls repo:
    - MBEDTLS_SSL_CBC_RECORD_SPLITTING, MBEDTLS_SSL_PROTO_TLS1, MBEDTLS_SSL_PROTO_TLS1_1
- [FRO32K]
  - FRO32K notification callback PLATFORM_FroDebugCallback_t() has new parameter to report he fro_trim value
  - maxCalibrationIntervalMs value can be provided to NBU using PLATFORM_FwkSrvSetRfSfcConfig()
- [Sensors]
  - fix: PLATFORM_GetTemperatureValue() shall have NBU started to send temperature to NBU

6.2.1: RW61x RFP3

- [NVS]
  - Add support of NVS and Settings in framework
- [MISRA] fixes
  - board_lp.c BOARD_UninitDebugConsole() and BOARD_ReinitDebugConsole()

  - fwk_platform_ble.c: Various fixes
- [OTA]
  - Fix OTA partition overflow during OTA stop and resume transfer

6.2.0: RT1060/RT1170 SDK2.15 Major

6.1.8: KW45/K32W1 MR4

- [BOARD PLATFORM]
  - Move gBoardUseFro32k_d to board_platform.h file
  - Offer the possibility to change the source clock accuracy to gain in power consumption
- [BOARD LP]
  - Move PLATFORM_SetRamBanksRetained() at end of BOARD_EnterLowPowerCb() in case a memory allocation is done previously in this function
  - fix low power, increase BOARD_RADIO_DOMAIN_WAKE_UP_DELAY from 0 to 0x10 - Skip this delay when App requesting NBU wakeup
- [PLATFORM]
  - fwk_platform_ble.c/h: New timestamp API that returns the difference between the current value of the LL clock and the argument of the function
  - fwk_platform.c/h:
    - New PLATFORM_EnableEccFaultsAPI_d compile flag: Enable APIs for interception of ECC Fault in bus fault handler
    - New gInterceptEccBusFaults_d compile flag: Provide FaultRecovery() demo code for bus fault handler to Intercept bus fault from Flash Ecc error
- [LOC]
  - Incorrect behavior for set_dtest_page (DqTEST11 overridden)
  - Fix SW1 button wake able on Localization board
  - Fix yellow led not properly initialized
  - Format localization pin_mux.c/h files
- [Inter Core]
  - Affect values to enumeration giving the inter core service message ids
  - Shared memory settings shared between both cores
  - Add callback to register when NBU has unrecoverable Radio issue
- [NVM]
  - Add NV_STORAGE_MAX_SECTORS, NV_STORAGE_SIZE as linker symbol for alignment with other toolchain
  - ECC detection and recovery. New gNvSalvageFromEccFault_d and gNvVerifyReadBackAfterProgram_d compile flags. Please refer to ECC Fault detection section in README.md file located in NVM folder
- [OTA]
  - Prevent bus fault in case of ECC error when reading back OTA_CFR update status (disable by default)
- [SecLib]

  - Shared mutex for RNG and SecLib as they share same hardware resource
- [Key storage]
  - Fix to ignore the garbage at the end of buffers
  - Detect when buffers are too small in KS_AddKey() functions
- [FileCache]
  - Fix deadlock in Filecache FC_Process()
- [SDK]
  - Applications: remove definition of stack location and use default from linker script, fix warmboot stack in freertos at 0x20004000
  - Memory Manager Light:
    * fix Null pointer harfault when MEM_STATISTICS_INTERNAL enable
    * Fix MemReinitBank() on wakeup from lowpower when Ecc banks are turned off

6.1.7: KW45/K32W1 MR3

- [OTA]
  - New API OTA_SetNewImageFlagWithOffset()
  - Fix StorageBitmapSize calculation
  - OTA clean up: Removed OTA_ValidateImage()
- [Low Power]
  - New linker Symbol m_lowpower_flag_start in linker file.
    * Flag is used to indicate NBU that Application domain goes to power down mode. Keep this flag to 0 if only Deep sleep is supported
    * This flag will be set to 1 if Application domain goes to power down mode
  - Re-introduce PWR_AllowDeviceToSleep()/PWR_DisallowDeviceToSleep(), PWR_IsDeviceAllowedToSleep() API
  - Implement tick compensation mechanism for idle hook in a dedicated freertos utils file fwk_freertos_utils.[ch], new functions: FWK_PreIdleHookTickCompensation() and FWK_PostIdleHookTickCompensation
  - Rework timestamping on K4W1
    * PLATFORM_GetMaxTimeStamp() based on TSTMR
    * Rename PLATFORM_GetTimestamp() to PLATFORM_GetTimeStamp()
    * Update PLATFORM_Delay(): Rework to use TSTMR instead of LPTMR for platform_delay
    * Update PLATFORM_WaitTimeout(): Fixed a bug in PLATFORM_WaitTimeout() related to timer wrap
    * Add PLATFORM_IsTimeoutExpired() API
  - Fix race condition in PWR_EnterLowPower(), masking interrupts in case not done at upper layer
  - Low power timer split in new files fwk_platform_lowpower_timer.[ch]
  - New PWR_systicks_bm.c file for bare metal usage: implement SysTick suspend/resume functionality, New weak PWR_SysTicksLowPowerInit()
- [FRO32K]

  - Improve FRO32K calibration in NBU
  - create PLATFORM_InitFro32K() to initialize FRO32K instead of XTAL32K (to be called from hardware_init())
  - update FRO32K README.md file in SFC module
  - Debug:
  - Add Notification callback feature for SFC module FRO32K
  - Linker script update to support m_sfc_log_start in SMU2
- [SecLib]
  - Remove gSecLibSssUseEncryptedKeys_d compile option, split Secure/Unsecure APIs
  - RNG update to use same mutex than SecLib
  - Fix AES_128_CBC_Encrypt_And_Pad length
  - Implement RNG_ReInit() for lowpower
  - Fix issue in ECDH_P256_GenerateKeys() when waking up from power down
  - Call CRYPTO_ELEMU_reset() from SecLib_reInit() for power down support
- [BOARD]
  - Create new board_platform.h file for all Board characteristics settings (32Mhz XTAL, 32KHZ XTAL, etc..)
  - TM_EnterLowpower() TM_EnterLowpower() to be called from LP callbacks
  - Support Localization boards, Only BUTTON0 supported
    - New compile flag BOARD_LOCALIZATION_REVISION_SUPPORT
    - New pin_mux.[ch] files
  - Offer the possibility to override CDAC and ISEL 32MHz settings before the initialization of the crystal in board_platform.h
    - new BOARD_32MHZ_XTAL_CDAC_VALUE, BOARD_32MHZ_XTAL_ISEL_VALUE
    - BOARD_32MHZ_XTAL_TRIM_DEFAULT obsoleted
- [NVM file system]
  - Look ahead in pending save queue - Avoid consuming space to save outdated record
  - Fix NVM gNvDualImageSupport feature in NvIsRecordCopied
- [Inter Core]
  - Change PLATFORM_NbuApiReq() API return parameters granularity from uint32 to uint8
  - MAX_VARIANT_SZ change from 20 to 25
  - Set lp wakeup delay to 0 to reduce time of execution on host side, NBU waits XTAL to be ready before starting execution
  - Update inter core config rpmsg_config.h
  - Add timeout to while loops that relies on hardware in RemoteActiveReq(), Application can register Callbacks when timeout
  - Return non-0 status when calling PLATFORM_FwkSrvSendPacket when NBU non started
  - Let PLATFORM_GetNbuInfo return -10 if response not received on timeout - Doxygen platform_ics APIs
- [HW params]

  - New compile Macro for HW params placement in IFR - Save 8K in FLash: gHwParamsProdDataPlacement_c . 3 modes:
  - Legacy placement, move from legacy to IFR, IFR only placement
  - New compile Macro for Application data to be stored with HW params (in shared flash sector): gHwParamsAppFactoryDataExtension_d, New APIs:
    * Nv_WriteAppFactoryData(), Nv_GetAppFactoryData()
  - See HWParameter.h
- [Platform]
  - Implement PLATFORM_GetIeee802_15_4Addr() API in fwk_platform_ot.c - New gPlatformUseUniqueDeviceIdFor15_4Addr_d compile Macro
  - Wakeup NBU domain when reading RADIO_CTRL UID_LSB register in PLATFORM_GenerateNewBDAddr()
- [Reset]
  - New reset Implementations using Deep power down mode or LVD:
    * new files fwk_platform_reset.[ch]
    * new APIs: PLATFORM_ForceDeepPowerDownReset(), PLATFORM_ForceLvdReset() + reset on ext pins
    * new compile flags: gAppForceDeepPowerDownResetOnResetPinDet_d and gAppForceLvdResetOnResetPinDet_d to reset on external pins
- [FSCI]
  - fix when gFsciRxAck_c enabled
  - integrate new reset APIs

6.1.4: RW610/RW612 RFP1

- [Low Power]
  - Added support of low power for OpenThread stack.
  - Added PWR_AllowDeviceToSleep/PWR_DisallowDeviceToSleep/PWR_IsDeviceAllowedToSleep APIs.
- [platform]
  - Added PLATFORM_GetMaxTimeStamp API.
  - Fixed high impact Coverity.
- [FreeRTOS]
  - Created a new utilities module for FreeRTOS: fwk_freertos_utils.c/h.
  - Implemented a tick compensation mechanism to be used in FreeRTOS idle hook, likely around flash operations. This mechanism aims to estimate the number of ticks missed by FreeRTOS in case the interrupts are masked for a long time.

6.1.4: KW45/K32W1 MR2

- [Low power]
  - Powerdown mode tested and enabled on Low Power Reference Design applications
  - XTAL32K removal functionality using FRO32K, supported from NBU firmwares - limitation: Application domain supports Deep Sleep only (not power down)

  - NBU low power improvement: low power entry sequence improvement and system clock reduction to 16Mhz during WFI
  - Wake up time from cold boot, reset, power switch greatly improved. Device starts on FRO32K, switch to XTAL32K when ready if gBoardUseFro32k_d not set
  - Bug fixes:
    * Move PWR LowPower callback to PLATFORM layers
    * Fix wrong compensation of SysTicks
    * Reinit system clocks when exiting power down mode: BOARD_ExitPowerDownCb(), restore 96MHz clock is set before going to low power
    * Call Timermanager lowpower entry exit callbacks from PLATFORM_EnterLowPower()
    * Update PLATFORM_ShutdownRadio() function to force NBU for Deep power down mode
  - K32W1:
    * Support lowpower mode for 15.4 stacks
- [NVM]
  - New Compilation MACRO gNvDualImageSupport to support multiple firmware image with different register dataset
  - Change default configuration gNvStorageIncluded_d to 1, gNvFragmentation_Enabled_d to 1, gUnmirroredFeatureSet_d to TRUE
  - Some MISRA issues for this new configuration.
  - Remove deprecated functionality gNvUseFlexNVM_d
- [SecLib]
  - New NXP Ultrafast ecp256 security library:
    * New optimized API for ecdh DhKey/ecp256 key pair computation: Ecdh_ComputeDhKeyUltraFast(), ECP256_GenerateKeyPairUltraFast().
    * New macro gSecLibUseDspExtension_d.
    * Improved software version of Seclib with Ultrafast library for ECP256_LePointValid()
  - Bug fixes:
    * Share same mutex between Seclib and RNG to prevent concurrent access to S200
    * Optimized S200 re-initialization, restore ecdh key pair after power down
    * Fixed race condition when power down low power entry is aborted
    * Endianness function updates and clean up
- [OTA]
  - OTASupport improvements:
    * New API OTA_GetImgState(), OTA_UpdateImgState()
    * OTASupport and fwk_platform_extflash API updates for external flash: OTA_SelectExternalStoragePartition(), PLATFORM_IsExternalFlashSectorBlank(), PLATFORM_IsExternalFlashPageBlank(), PLATFORM_OtaGetOtaPartitionConfig()
    * Updated OtaExternalFlash.c, 2 new APIs in fwk_platform_extflash.c

    * Removed unused FLASH_op_type and FLASH_TransactionOpNode_t definitions from public API
    * Removed unused InternalFlash_EraseBlock() from OtaInternalFlash.c

- [NBU firmware]
  - Mechanism to set frequency constraint to controller from the host PLATFORM_SetNbuConstraintFrequency()
  - NbuInfo has one more digit in versionBuildNo field
- [Board]
  - Support Extflash low power mode, add BOARD_UninitExternalFlash(), PLATFORM_UninitExternalFlash(), PLATFORM_ReinitExternalFlash()
  - Support XTAL32K removal functionatity, use FRO32K instead by setting gBoardUseFro32k_d to 1 in board.h file
  - Support localization boards KW45B41Z-LOC Rev C
  - Low power improvement: New BOARD_InitPins() and BOARD_InitPinButtonBootConfig() called from hardware_init.c
  - Removed KW45_A0_SUPPORT support (dcdc)
  - Bug fixes:
    * Fixed glitches on the serial manager RX when exiting from power down
    * Fixed ADC not deinitialized in clock gated modes in BOARD_EnterLowPowerCb()
    * Fixed UART output flush when going to low power: BOARD_UninitAppConsole()
- [platform]
  - PLATFORM_InitBle(), PLATFORM_SendHci() can now block with timeout if NBU does not answer. Application can register callback function to be notified when it occurs: PLATFORM_RegisterBleErrorCallback()
  - Added API to set and get 32Khz XTAL capacitance values: PLATFORM_GetOscCap32KValue() and PLATFORM_SetOscCap32KValue()
  - Added new Service FWK call gFwkSrvNbuMemFullIndication_c to get NBU mem full indication, register with PLATFORM_RegisterNbuMemErrorCallback()
  - Added support negative value in platform intercore service
- [linker script]
  - Realigned gcc linker script with IAR linker script.
  - Added possibility to redefine cstack_start position
  - Added Possibility to change gNvmSectors in gcc linker script
  - Added dedicated reserved Section in shared memory for LL debugging
- [FreeRTOSConfig.h]
  - Removed unused MACRO configFRTOS_MEMORY_SCHEME and configTOTAL_HEAP_SIZE
- [HW Param]
  - Added xtalCap32K field to store XTAL32K triming value
- [fwk_hal_macros.h]
  - Added MACRO for KB, MB and set, clear bits in bit fields
- [Debug]

  - Added MACROs for performance measurement using DWT: DBG_PERF_MEAS

6.1.3 KW45 MR1 QP1

- [Initialization] Delay the switch to XTAL32K source clock until the BLE host stack is initialized
- [lowpower] NBU wakeup from lowpower: configuration can now be programmed with BOARD_NBU_WAKEUP_DELAY_LPO_CYCLE, BOARD_RADIO_DOMAIN_WAKE_UP_DELAY in board.h file
- [NBU firmware] Major fix for NBU system clock accuracy
- [clock_config]
  - Update SRAM margin and flash config when switching system frequency
  - Trim FIRC in HSRUN case
- [XTAL 32K trim] XTAL 32K configuration can be tuned in board.h file with BOARD_32MHZ_XTAL_TRIM_DEFAULT, BOARD_32KHZ_XTAL_CLOAD_DEFAULT, BOARD_32KHZ_XTAL_COARSE_ADJ_DEFAULT
- [MAC address] Add OUI field in PLATFORM_GenerateNewBDAddr() when using Unique Device Id

6.1.2: RW610/RW612 PRC1

- [Low Power]
  - Updates after SDK Power Manager files renaming.
  - Moved PWR LowPower callback to PLATFORM layers.
  - Bug fixes:
    - Fixed wrong compensation of SysTicks during tickless idle.
    - Reinit RTC bus clock after exit from PM3 (power down).
- [OTA]
  - Initial support for OTA using the external flash.
- [platform]
  - Implemented platform specific time stamp APIs over OSTIMER.
  - Implemented platform specific APIs for OTA and external flash support.
  - Removed PLATFORM_GetLowpowerMode API.
  - Added support of CPU2 wake up over Spinel for OpenThread stack.
  - Bug fixes:
    - Fixed issues related to handling CPU2 power state.
- [board]
  - Updated flash_config to support 64MB range.
- [linker script]
  - Fixed wrong assert.

6.1.1: KW45/K32W1 MR1

- [platform] Use new FLib_MemSet32Aligned() to write in ECC RAM bank to force ECC calculation in the MEM_ReinitRamBank() function
- [FunctionLib] Implement new API to set a word aligned
- [platform] Set coarse amplifier gain of the oscilattor 32k to 3
- [platform] Switch back to RNG for MAC Adress generation
- [SecLib] Get rid of the lowpower constraint of deep sleep in ECDH API
- [DCDC] Set DCDC output voltage to 1.35V in case LDO core is set to 1.1V to ensure a drop of 250mV between them
- [NVM] NvIdle() is now returning the number of operations that has been executed
- [documentation] Add markdown of each framework module by default on all package
- [LowPower] Add a delay advised by hardware team on exit of lowpower for SPC
- [SecLib] Rework of SecLib_mbedTLS ECDH functions
- [OTA] Make OTA_IsTransactionPending() public API
- [FunctionLib] Change prototype of FLib_MemCpyWord(), pDst is now a void* to permit more flexibility
- [NVM] Add an API to know if there is a pending operation in the queue
- [FSCI] Fix wrong error case handling in FSCI_Monitor()

6.1.0: KW45/K32W1 RFP

- [LowPower] Do not call PLATFORM_StopWakeUpTimer() in PWR_EnterLowPower() if PLATFORM_StartWakeUpTimer() was not previously called
- [boards] Add the possibility to wakeup on UART 0 even if it is not the default UART
- [boards] Add support for Hardware flow control for UART0, Enable with gBoardUseUart0HwFlowControl, Pin mux update with two additional API for RTS, CTS pins
- [Sensors] Improve ADC wakeup time from deep sleep state: use save and restore API for ADC context before/after deep sleep state.
- [linker script] update SMU2 shared memory region layout with NBU: increase sqram_btblebuf_size to support 24 connections. Shared memory region moved to the end
- [SecLib] SecLib_DeriveBluetoothSKD() API update to support if EdgeLock key shall be regenerated

6.0.11: KW45/K32W1 PRC3.1

FSCI: Framework Serial Communication Interface

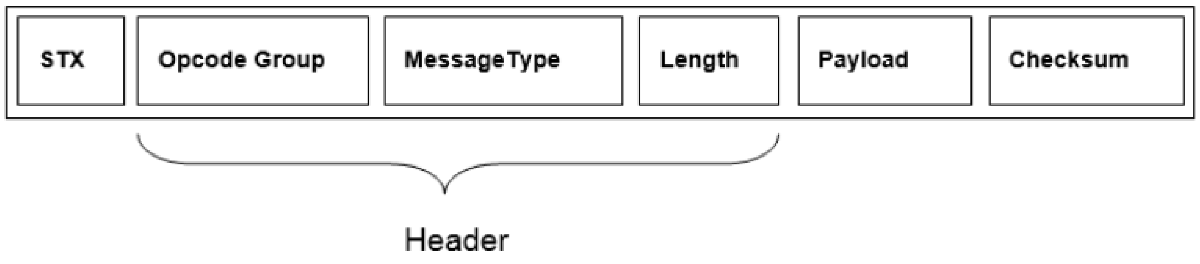
Overview The Framework Serial Communication Interface (FSCI) is both a software module and a protocol that allows monitoring and extensive testing of the protocol layers. It also allows separation of the protocol stack between two protocol layers in a two processing entities setup, the host processor (typically running the upper layers of a protocol stack) and the Black Box application (typically containing the lower layers of the stack, serving as a modem). The Test Tool software is an example of a host processor, which can interact with FSCI Black Boxes at various layers. In this setup, the user can run numerous commands to test the Black Box application services and interfaces.

The FSCI enables common service features for each device enables monitoring of specific interfaces and API calls. Additionally, the FSCI injects or calls specific events and commands into the interfaces between layers.

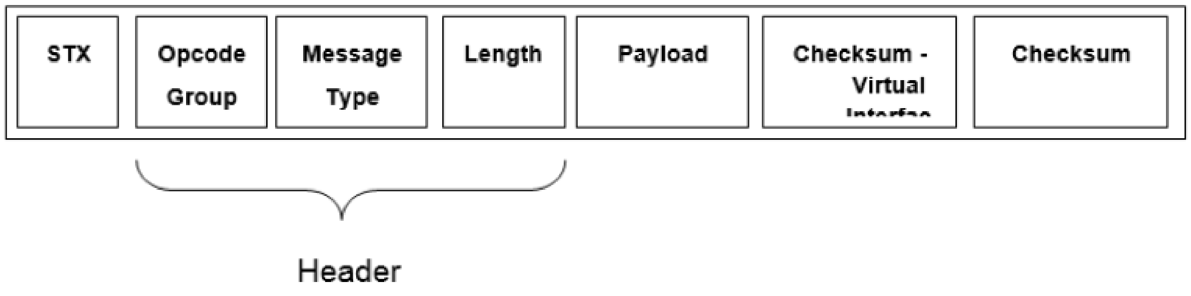
An entity which needs to be interfaced to the FSCI module can use the API to register opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode triggers a callback execution. Two or more entities cannot register the same opcode on the same interface, but they can do so on different interfaces. For example, two MAC instances can register the same opcodes, one over UARTA, and the other over UARTB. This way, Test Tool can communicate with each MAC layer over two UART interfaces.

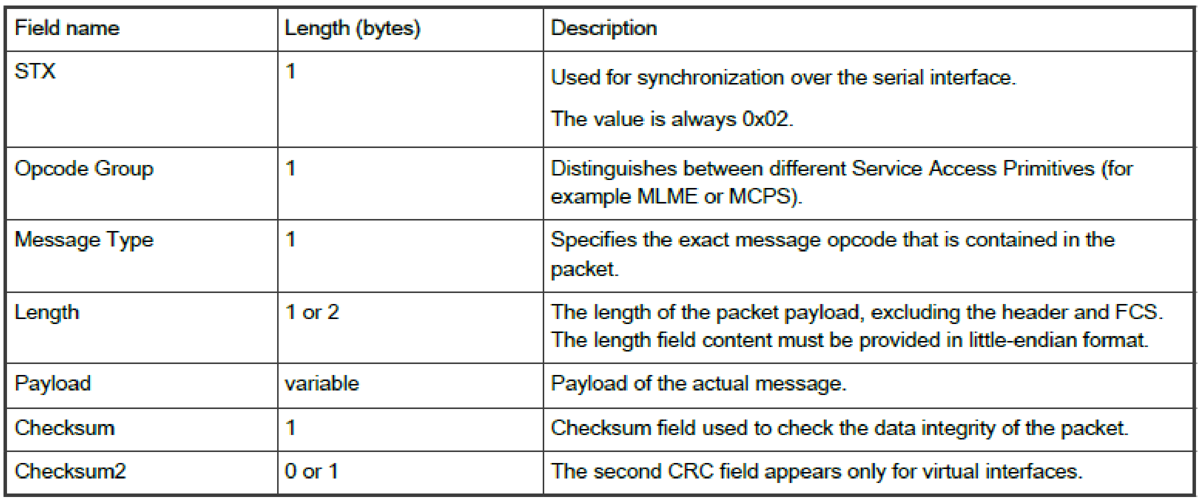
The FSCI module executes either in the context of the Serial Manager task or owns its dedicated task if the compilation Macro *gFsciUseDedicatedTask_c* is set to 1.

FSCI packet structure The FSCI module sends and receives messages as shown in the figure below. This structure is not specific to a serial interface and is designed to offer the best communication reliability. The Black Box device expects messages in little-endian format. It also responds with messages in little-endian format.



Below is an illustration of the FSCI packet structure when a virtual interface is used instead :





Field name	Length (bytes)	Description
STX	1	Used for synchronization over the serial interface. The value is always 0x02.
Opcode Group	1	Distinguishes between different Service Access Primitives (for example MLME or MCPS).
Message Type	1	Specifies the exact message opcode that is contained in the packet.
Length	1 or 2	The length of the packet payload, excluding the header and FCS. The length field content must be provided in little-endian format.
Payload	variable	Payload of the actual message.
Checksum	1	Checksum field used to check the data integrity of the packet.
Checksum2	0 or 1	The second CRC field appears only for virtual interfaces.

> *NOTE* : When virtual interfaces are used, the first checksum is decremented with the ID of the interface. The second checksum is used for error detection.

constant definition The following Macro configurs the FSCI module

```
#define gFsciIncluded_c 0 /* Enable/Disable FSCI module */
#define gFsciUseDedicatedTask_c 1 /* Enable Fsci task to avoid recursivity in Fsci module (Misra␣
↪compliant) */
#define gFsciMaxOpGroups_c 8
#define gFsciMaxInterfaces_c 1
#define gFsciMaxVirtualInterfaces_c 0
#define gFsciMaxPayloadLen_c 245 /* bytes */
#define gFsciTimestampSize_c 0 /* bytes */
#define gFsciLenHas2Bytes_c 0 /* boolean */
#define gFsciUseEscapeSeq_c 0 /* boolean */
#define gFsciUseFmtLog_c 0 /* boolean */
#define gFsciUseFileDataLog_c 0 /* boolean */
#define gFsciLoggingInterface_c 1 /* [0..gFsciMaxInterfaces_c) */
#define gFsciHostMacSupport_c 0 /* Host support at MAC layer */
```

The following provides the OpGroups values reserved by MAC, application, and FSCI.

FSCI Host FSCI Host is a functionality that allows separation at a certain stack layer between two entities, usually two boards running separate layers of a stack.

Support is provided for functionality at the MAC layer, for example, MAC/PHY layers of a stack are running as a Black Box on a board, and MAC higher layers are running on another. The higher layers send and receive serial commands to and from the MAC Black Box using the FSCI set of operation codes and groups.

The protocol of communication between the two is the same. The current level of support is provided for:

- FSCI_MsgResetCPUReqFunc – sends a CPU reset request to black box
- FSCI_MsgWriteExtendedAdrReqFunc – configures MAC extended address to the Black Box
- FSCI_MsgReadExtendedAdrReqFunc – N/A

The approach on the Host interfacing a Black Box using synchronous primitives is by default the polling of the FSCI_receivePacket function, until the response is received from the Black Box. The calling task polls whenever the task is being scheduled. This is required because a stack synchronous primitive requires that the response of that request is available in the context of the caller right after the SAP call has been executed.

The other option, available for RTOS environments, is using an event mechanism. The calling task blocks waiting for the event that is sent from the Serial Manager task when the response is available from the Black Box. This option is disabled by default. The disadvantage of this option is that the primitive cannot be received from another Black Box through a serial interface because the blocked task is the Serial Manager task, which reaches a deadlock as cannot be released again.

FSCI ACK ACK transmission is enabled through the gFsciTxAck_c macro definition. Each FSCI valid packet received triggers an FSCI ACK packet transmission on the same FSCI interface that the packet was received on. The serial write call is performed synchronously to send the ACK packet before any other FSCI packet. Only then the registered handler is called to process the received packet. The ACK is represented by the gFSCI_CnfOpcodeGroup_c and mFsciMsgAck_c Opcode. An additional byte is left empty in the payload so that it can be used optionally as a packet identifier to correlate packets and ACKs. ACK reception is the other component that is enabled through gFsciRxAck_c. The behavior is such that every FSCI packet sent through a serial interface triggers an FSCI ACK packet reception on the same interface after the packet is sent. If an ACK packet is received, the transmission is considered successful. Otherwise, the packet is re-sent a number of times. The ACK wait period is configurable through mFsciRxAckTimeoutMs_c and the number of transmission retries through mFsciTxRetryCnt_c. The ACK mechanism described above can also be coupled with a FSCI packet reception timeout enabled through gFsciRxTimeout_c and configurable through mFsciRxRestartTimeoutMs_c. Whenever there are no more bytes to be read from a serial interface, a timeout is configured at the predefined value if no other bytes are received. If new bytes are received, the timer is stopped and eventually canceled at successful reception. However, if, for any reason, the timeout is triggered, the FSCI module considers that the current packet is invalid, drops it, and searches for a new start marker.

FSCI usage example Detailed data types and APIs are described in ConnFWK API documentation.

Initialization

```
/* Configure the number of interfaces and virtual interfaces used */
#define gFsciMaxInterfaces_c 4
#define gFsciMaxVirtualInterfaces_c 2
....
/* Define the interfaces used */
static const gFsciSerialConfig_t myFsciSerials[] = {
    /* Baudrate, interface type, channel No, virtual interface */ {gUARTBaudRate115200_c, gSerialMgrUart_
↪c, 1, 0}, {gUARTBaudRate115200_c, gSerialMgrUart_c, 1, 1}, {0 , gSerialMgrIICSlave_c, 1, 0}, {0 ,␣
↪gSerialMgrUSB_c, 0, 0},
};
....
/* Call init function to open all interfaces */
FSCI_Init( (void*)mFsciSerials );
```

Registering operation groups

```
myOpGroup = 0x12; // Operation Group used
myParam = NULL; // pointer to a parameter to be passed to the handler function (myHandlerFunc)
myInterface = 1; // index of entry from myFsciSerials
...
FSCI_RegisterOpGroup( myOpGroup, gFsciMonitorMode_c, myHandlerFunc, myParam, myInterface );
```

Implementing handler function

```
void fsciMcpsReqHandler(void *pData, void* param, uint32_t interfaceId)
{
    clientPacket_t *pClientPacket = ((clientPacket_t*)pData);
    fsciLen_t myNewLen;
    switch( pClientPacket->structured.header.opCode )
    {
        case 0x01:
        {
            /* Reuse packet received over the serial interface The OpCode remains the same. The length of the␣
↪response must be <= that the length of the received packet */
            pClientPacket->structured.header.opGroup = myResponseOpGroup;/* Process packet */
            ...
            pClientPacket->structured.header. len = myNewLen;
            FSCI_transmitFormatedPacket(pClientPacket, interfaceId);
            return;
        }
        case 0x02:
        {
```

(continues on next page)

(continued from previous page)

```
            /* Alocate a new message for the response. The received packet is Freed */
            clientPacket_t *pResponsePkt = MEM_BufferAlloc( sizeof(clientPacketHdr_t) + myPayloadSize_d␣
↪+ sizeof(uint8_t) // CRC);
            if(pResponsePkt)
            {
                /* Process received data and fill the response packet */ ...
                pResponsePkt->structured.header. len = myPayloadSize_d;
                FSCI_transmitFormatedPacket(pClientPacket, interfaceId);
            }
            break;
        }
        default:
            MEM_BufferFree( pData );
            FSCI_Error( gFsciUnknownOpcode_c, interfaceId );
            return;
    }
    /* Free message received over the serial interface */
    MEM_BufferFree( pData );
}
```

Helper Functions Library

Overview This framework provides a collection of features commonly used in embedded software centered on memory manipulation.

HWParameter: Hardware parameter

Production Data Storage Hardware parameters provide production data storage

Overview Different platforms/boards need board/network node-specific settings to function according to the design. (Examples of such settings are IEEE® addresses and radio calibration values specific to the node.) For this purpose, the last flash sector is reserved and contains hardware-specific parameters for production data storage. These parameters pertain to the network node as a distinct entity. For example, a silicon mounted on a PCB in a specific configuration, rather than to just the silicon itself. This sector is reserved by the linker file, through the PROD_DATA section and it should be read/written only through the API described below.

> Note : This sector is not erased/written at code download time and it is not updated via over-the-air firmware update procedures to preserve the respective node-specific data, regardless of the firmware running on it.

Constant Definitions Name :

```
extern uint32_t PROD_DATA_BASE_ADDR[];
```

Description :

This symbol is defined in the linker script. It specifies the start address of the PROD_DATA section.

Name :

```
static const uint8_t mProdDataIdentifier[10] = {"PROD_DATA:"};
```

Description :

The value of this constant is copied as identification word (header) at the beginning of the PROD_DATA area and verified by the dedicated read function.

Note: the length of mProdDataIdentifier imposes the definition of PROD_DATA_ID_STRING_SZ as 10. The legacy HW parameters structure provides headroom for future usage. There are currently 63 bytes available.

Data type definitions Name :

```
typedef PACKED_STRUCT HwParameters_tag
{
    uint8_t identificationWord[PROD_DATA_ID_STRING_SZ]; /* internal usage only: valid data present */
    /*@{*/
    uint8_t bluetooth_address[BLE_MAC_ADDR_SZ];      /*!< Bluetooth address */
    uint8_t ieee_802_15_4_address[IEEE_802_15_4_SZ]; /*!< IEEE 802.15.4 MAC address - K32W1 only␣
↪*/
    uint8_t xtalTrim;                                /*!< XTAL 32MHz Trim value */
    uint8_t xtalCap32K;                              /*!< XTAL 32kHz capacitance value */
    /* For forward compatibility additional fields may be added here
       Existing data in flash will not be compatible after modifying the hardwareParameters_t typedef.
       In this case the size of the padding has to be adjusted.
    */
    uint8_t reserved[1];
    /* first byte of padding : actual size if 63 for legacy HwParameters but
      complement to 128 bytes in the new structure */
}
hardwareParameters_t;
```

Description:

Defines the structure of the hardware-dependent information.

> Note : Some members of this structure may be ignored on a specific board/silicon configuration. Also, new members may be added for implementation-specific purposes and the backward compatibility must be maintained.

The CRC calculation starts from the reserved field of the hardwareParameters_t and ends before the hardwareParamsCrc field. Additional members to this structure may be added using the following method :

Add new fields before the reserved field. This method does not cause a CRC fail, but you must keep in mind to subtract the total size of the new fields from the size of the reserved field. For example, if a field of uint8_t size is added using this method, the size of the reserved field shall be changed to 63.

Co-locating application factory data in HW Parameters flash sector. The sector containing the Hardware parameter structure may be located in the internal flash, usually at its last sector. The actual Hardware parameter structure has a size of 128 bytes - including padding reserved for future use. Since there is plenty of room available in a flash sector (4kB or 8kB), co-locating Application Factory Data in the same structure prevents from reserving another flash sector for these data. The application designer may adopt this solution by defining gHwParamsAppFactoryDataExtension_d as 1. A total of 2kB is alloted to this purpose.

If this option was chosen, whenever any of the Hardware parameter fields is modified, its CRC16 will change so the sector will need erasing. The gHwParamsAppFactoryDataPreserveOnHwParamUpdate_d compilation option deals with restoring the contents of the App Factory Data. Nonetheless this requires a temporary allocation a 2kB buffer to preserve the previous content and restore then on completion of the Hw Parameter update.

Special reserved area at start of IFR1 in range [0x02002000..0x02002600[On development boards a 1536 byte area is reserved and the actual Hardware parameter area begins at offset 0x600. Preserving this area on a HW parameter update also requires a temporary 1.5kB dynamic allocation (in addition to the App Factory 2kB allocation), to be able to restore on completion of update operation.

HW Parameters Production Data placement options The placement of production data (PROD_DATA) can be selected based on the definition of gHwParamsProdDataPlacement_c (see fwk_config.h). The productions data seldom need update for final products, once calibration data, MAC addresses or others have been programmed. Two cases exist, plus a transition mode :

1) gHwParamsProdDataMainFlashMode_c (0) :
   - PROD_DATA are located at top of Main Flash. Hardware parameters section is placed in the last sector of internal flash [0xfe000..0x100000[.
   - The linker script must reserve this area explicitly so as to prevent placement of NVM or text sections at that location by setting gUseProdInfoMainFlash_d.
2) gHwParamsProdDataMainFlash2IfrMode_c(1) : - PROD_DATA are located in IFR1, but MainFlash version still exists during interim period. - If the contents of the PROD_DATA section in MainFlash is valid (not blank and correct CRC) but the IFR PROD_DATA is still blank, copy the contents of MainFlash PROD_DATA to IFR location. - When done PROD_DATA in IFR are used. Once the transition is done, an application using (2: gHwParamsProdDataPlacementIfrMode_c) may be programmed.
3) gHwParamsProdDataIfrMode_c (2) :
   - PROD_DATA section dwells in the IFR1 sector [0x02002000..0x02004000[
   - in development phase the area comprised between [0x02002000..0x02002600[must be reserved for internal purposes.
   - This allows to free up the top sector of Main Flash by linking with gUseProdInfoMainFlash_d unset.

LowPower

Low Power reference user guide This Readme file describes the connectivity software architecture and provides the general low power enablement user guide.

1- Connectivity Low Power SW architecture The connectivity low power software architecture is composed of various components. These are described from the lower layer to the application layer:

1. The SDK power manager in component/power_manager. This component provides the basic low power framework. It is not specific to the connectivity but generic across devices. it covers:
   - gather the low power constraints for upper layer and take the decision on the best suitable low power state the device is allowed to go to fullfill the constraints.
   - call the low power entry and exit function callbacks
   - call the appropriate SW routines to switch the device into the suitable low power state
2. Connectivity Low power module in the connectivity framework. This module is composed of:

- The low power service called PWR inside framework/LowPower (this folder), This module is generic to all connectivity devices.
- The platform lowpower: fwk_platform_lowpower.[ch] located in framework\platform\<platform_name>. These files are a collection of low power routines functions for the PWR module and upper layer. These are specific to the device.

  Both PWR and platform lowpower files are detailed in section below.

3. Low power Application modules, it consists of 3 parts:
   - Application initialization file app_services_init.c where the application initializes the low power framework, see next section 'Demo example for typical usage of low power framework'
   - Application Idle task from application to call the main low power entry function PWR_EnterLowPower() to switch the device into lowpower. This function is application specific, one example is given in the section 1.3.3
   - Low power board files : board_lp.[ch] located in board/lowpower. These files implement the low power entry and exit functions related to the application and board. Customers shall modify these files for their own needs. Example code is given for the connectivity applications.

     User guide is provided in section 1.3 below.

   Note : Linker script may also be impacted for power down mode support in order to provide an RAM area for ROM warm boot (depends on the platform) and application warmboot stack

The Low power central and master reference design applications provide an example of Low power implementation for BLE. Customer can also refer to the associated document 'low power connectivity reference design user guide'.

1.1 - SDK power manager This module provides the main low power functionalities such as:

- Decide the best low-power mode dependent on the constraints set by upper layers by using PWR_SetLowPowerModeConstraints() API function.
- Handle the sequences to enter and exit low-power mode.
- Enable and configure wake up sources, call the application callbacks on low power entry/exit sequences.

The SDK power manager provides the capability for application and all components to receive low power constraints to the power. The Application does not set the low-power mode the device shall go into. When going to low power, the SDK power manager selects the best low-power mode that fits all the constraints.

As an example, if the low power constraint set from Application is Power Down mode, and no other constraint is set, the SDK power manager selects Power down mode, the next time the device enters low power. However, if a new constraint is set by another component, such as the SecLib module that operates Hardware encryption, the SecLib module would select WFI as additional low power constraint. Also, the SDK power manager selects this last low-power mode until the constraint is released by the SecLib module. It then reselects Power Down mode for further low power entry modes.

1.2 - PWR Low power module The PWR module in the connectivity framework provides additional services for the connectivity stacks and applications on top of the SDK power manager.

It also provides a simple API for Connectivity Stack and Connectivity applications.

However, more advanced features such as configuring the wake-up sources are only accessible from the SDK Power Manager API.

In addition to the SDK Power Manager, the PWR module uses the software resources from lower level drivers but is independent of the platform used.

1.2.1 - Functional description Initialization of the PWR module should be done through PWR_Init() function. This is mainly to initialize the SDK power manager and the platform for low power. It also registers PWR low power entry/exit callback PWR_LowpowerCb() to the SDK power manager. This function will be called back when entering and exiting low power to perform mandatory save/restore operations for connectivity stacks. The application can perform extra optional save/restore operations in the board_lp file where it can register to the SDK Power Manager its own callback. This is usually used to handle optional peripherals such as serial interfaces, GPIOs, and so on.The main entry function is PWR_EnterLowPower(). It should be called from Idle task when no SW activity is required. The maximum duration for lowpower is given as argument timeoutUs in useconds. This function will check the next Hardware event in the connectivity stack, typically the next Radio activity. A wakeup timer is programmed if the timeoutUs value is shorter than the next radio event timing. Passing a timeout of 0us will be interpreted as no timeout on the application side.

On device wakeup from low power state, the function will return the time duration the device has been in low power state.

Two APi are provided to set and release low power state constraints : PWR_SetLowPowerModeConstraint() and PWR_ReleaseLowPowerModeConstraint(). These are helper functions. User can use directly the SDK power manager if needed.

The PWR module also provides some API to be set as callbacks into other components to prevent from going to low power state. It can be used in following examples :

1. If a DMA is running, the module in charge of the DMA would need to set a constraint to avoid the system from going to a low power state when the RAM and system bus are no longer available.
2. If transfer is going on a peripheral, the drivers shall set a constraint to forbid low power mode.
3. If encryption is on going through an Hardware accelerator, the HW accelerator and the required ressources (clocks, etc), shall be kept active also by setting a constraints.

1.2.2 - Tickless mode support This module also provides some routines functions PWR_SysticksPreProcess() and PWR_SysticksPostProcess() from PWR_systicks.c in order to support the tickless mode when using FreeRTOS. The tickless mode is the capability to suspend the periodic system ticks from FreeRTOS and keep timebase tracking using another low power counter. In this implementation, the Timer Manager and time_stamp component are used for this purpose.

Idle task shall call these functions PWR_SysticksPreProcess() and PWR_SysticksPostProcess() before and after the call to the main low power entry function PWR_EnterLowPower().

Refer to framework/LowPower/PWR_systicks.c file or section 2.1 below for more information.

1.3 - Low power platform submodule Low power platform module file fwk_platform_lowpower.c provides the necessary helper functions to support low power device initialization, device entry, and exit routines. These are platform and device specific. Typically, the PWR module uses the low power platform submodule for all low power specific routines.

The low power platform submodule is documented in the Connectivity Framework Reference Manual document and in the Connectivity Framework API document.

1.4 - Low power board files Low power board files board_lp.[ch] are both application and board specific. Users should update this file to add new functions to include new used peripherals that require low power support. In the current SDK package, only Serial Manager over UART and button (IO toggle wake up source) are supported and demonstrated in the Bluetooth LE demo application.

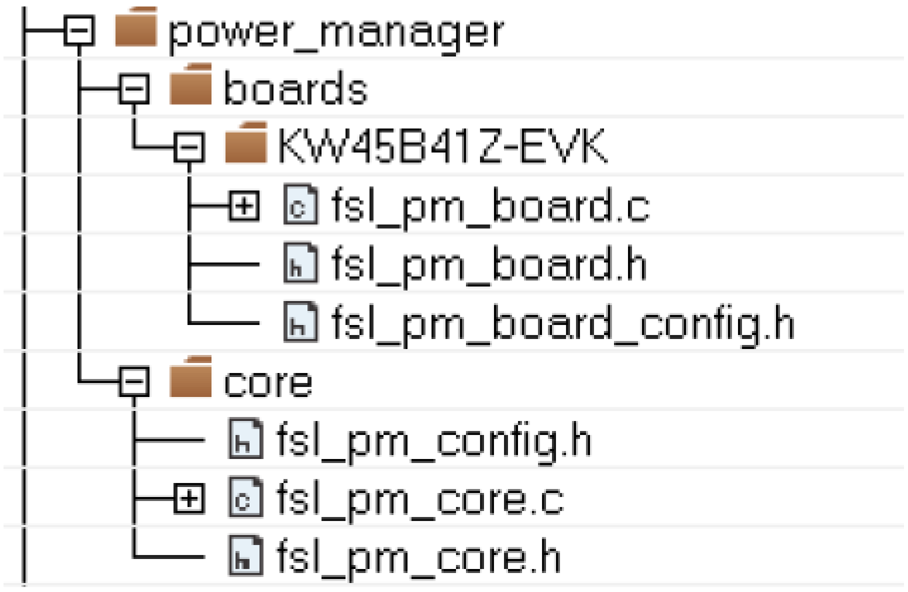
Other peripherals that require specific action on low power entry and restore on low power exit should be added to low power board files. For more details, refer to section Low power board file update

2 - Low power Application user guide This section provides a user guide to enable Low power on a connectivity application, It gives example of typical implementation for the initialization, Idle task function and low power entry/exit functions.

2.1 - Application Project updates It is recommended to reuse the low-power peripheral/central reference design application projects as a start. This ensures that everything is in place for the low-power optimization feature. Then, application files may be added to one of the two projects.

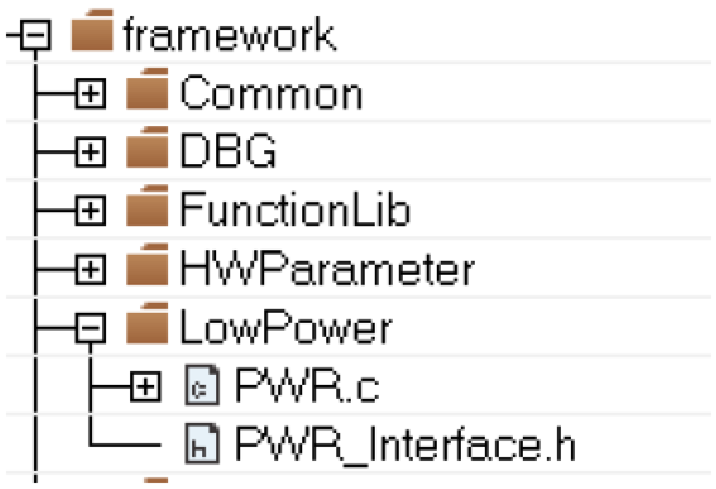
However, users can start directly from the application project and implement low power in it, by performing the steps described in the following sections.

2.1.1 - SDK Power Manager Most of the Low power functionality is implemented in the SDK Power Manager. The files to add into the project SDK power_manager module are listed in the figure below:



You need to use the files located in the folder that match your device.

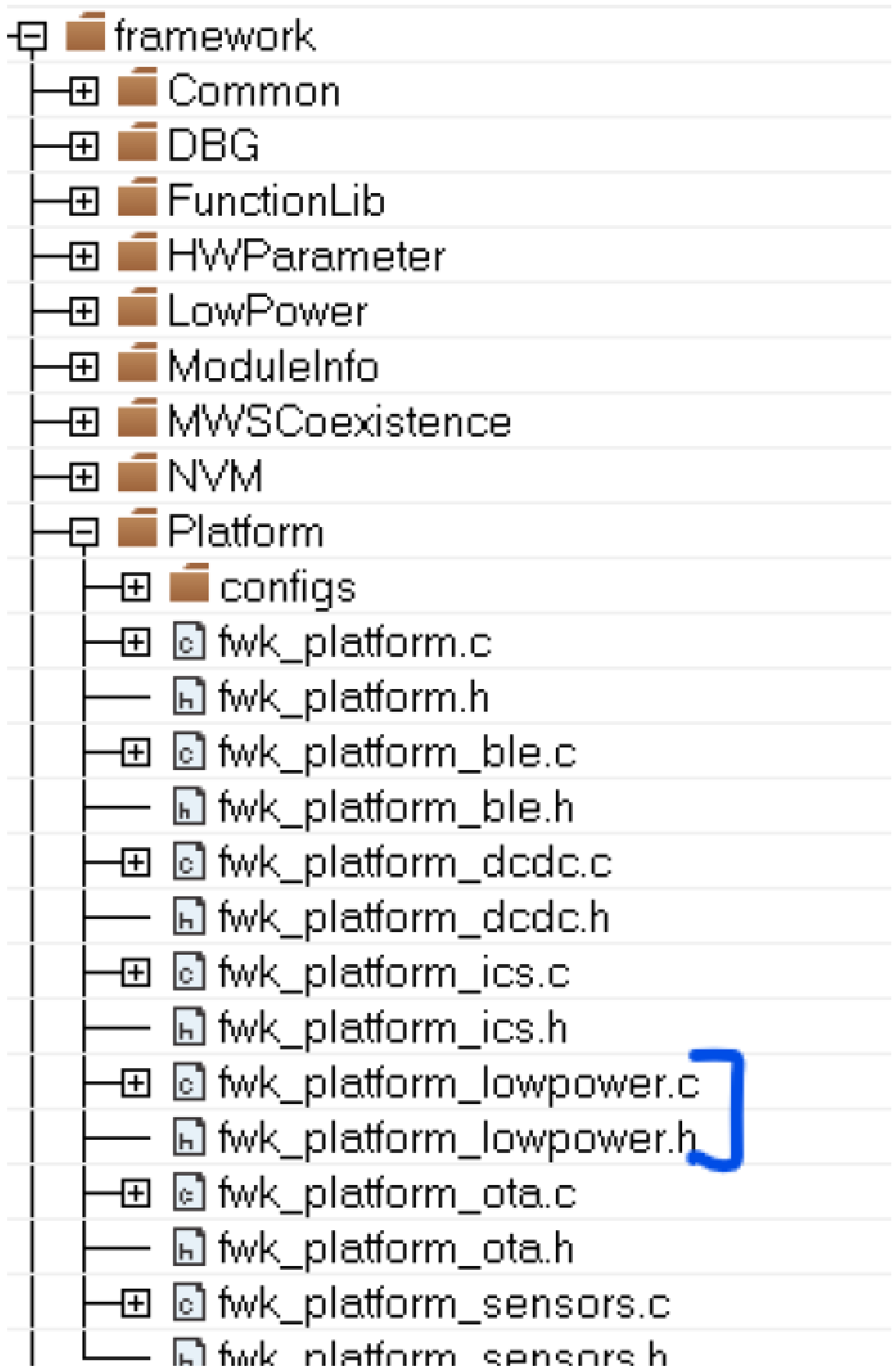
2.1.2 - PWR connectivity framework module PWR.c PWR_Interface.h shall be added to your application projects :



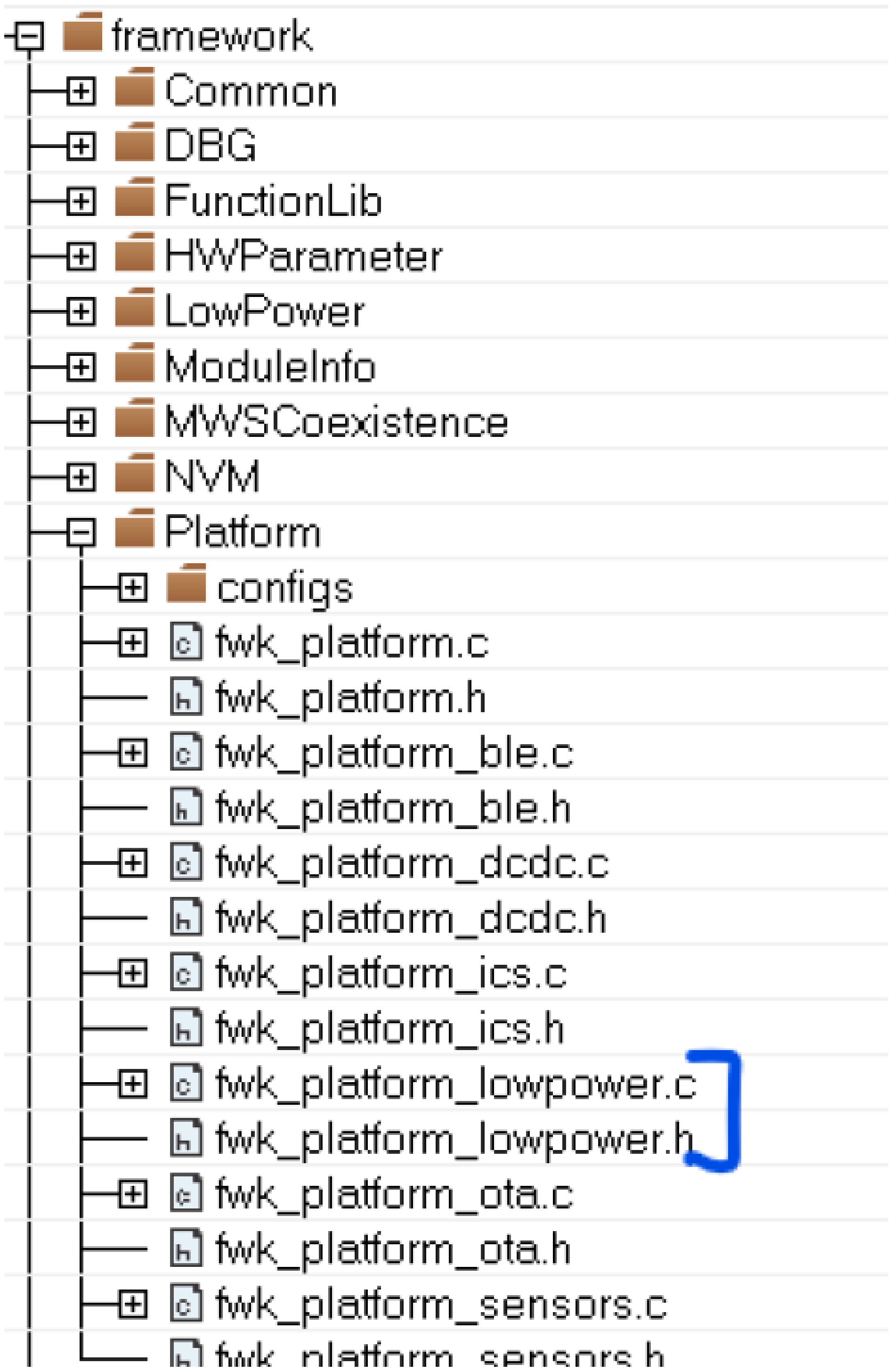
Optionally, in order to support Systick less mode, PWR_systicks.c or PWR_systicks_bm.c could also be added.

The include path to add is: middleware/wireless/framework/LowPower

2.1.3 -Low power platform submodule Low power platform files can be found in the 'Platform' module in the connectivity framework:



2.1.4 - Low power board files These files are located in the same folder that the other board files board.[ch]. Hence, it is not required to add any new include path at compiler command line.



2.1.5 - Application RTOS Idle hook and tickeless hook functions See section 2.4.3 Idle task implementation example

2.2 - Low power and wake up sources Initialization Low power initialization and configuration are performed in APP_ServiceInitLowpower()function. This is called from APP_InitServices() function called from the main() function so all is already set up when calling the main application entry point, typically BluetoothLEHost_AppInit() function in the Bluetooth LE demo applications.

The default Low Power mode configured in APP_InitServices() is Deep Sleep mode. In Bluetooth

LE, (or any other stack technology), Deep Sleep mode fits for all use cases. For instance, for Bluetooth LE states: Advertising, Connected, Scanning states. This mode already performs a very good level of power saving and likely, this is not required to optimize more if the device is powered from external supply.

APP_ServiceInitLowpower() function performs the following initialization and configuration:

- Initialize the Connectivity framework Low power module PWR_Init(), this function initialized the SDK power manager.
- Configure the wakeup sources such as serial manager wake up source for UART, or button for IO wake up configuration. These are typical wakeup sources used in the connectivity application. Developer may want to add additional wake up sources here specific for the application.

  Note : The low power timer wakeup source and wakeup from Radio domain are directly enabled from the Connectivity framework Low power module PWR as it is mandatory for the connectivity stack. If your application supports other peripherals (such as i2c, spi, and others) that require wake sources from low power, developer should add additional wake up sources setting in this function APP_ServiceInitLowpower(). The complete list of wakeup sources are available from the SDK power manager component, see file fsl_pm_board.h in component/boards/<device_name>/.
- Initialize and register the Low power board file used to register and implement low power entry and exit callback function used for peripheral. This is done by calling the BOARD_LowPowerInit() function.
- Register low power Enter and exit critical function to driver component to enable / disable low power when the Hardware is active. Example is given for serial manager that needs to disable low power when the TX ring buffer contains data so the device does not enter low power until the buffer is empty.

Finally, APP_ServiceInitLowpower() function configures the Deep Sleep mode as the default low power constraint for the application. It is recommended to keep this level of low power constraint during all the connectivity stack initialization.

Example of low power framework initialization can be found in app_services_init.c file. Below is some code example for initializing the low power framework and wake up sources:

```
static void APP_ServiceInitLowpower(void)
{
    PWR_ReturnStatus_t status = PWR_Success;

    /* It is required to initialize PWR module so the application
     * can call PWR API during its init (wake up sources...) */
    PWR_Init();

    /* Initialize board_lp module, likely to register the enter/exit
     * low power callback to Power Manager */
    BOARD_LowPowerInit();

    /* Set Deep Sleep constraint by default (works for All application)
     *   Application will be allowed to release the Deep Sleep constraint
     *   and set a deepest lowpower mode constraint such as Power down if it needs
     *   more optimization */
    status = PWR_SetLowPowerModeConstraint(PWR_DeepSleep);
    assert(status == PWR_Success);

#if (defined(gAppButtonCnt_c) && (gAppButtonCnt_c > 0))

    /* Init and enable button0 as wake up source
     * BOARD_WAKEUP_SOURCE_BUTTON0 can be customized based on board configuration
```

(continues on next page)

(continued from previous page)

```
     * On EVK we use the SW2 mapped to GPIOD */
    PM_InitWakeupSource(&button0WakeUpSource, BOARD_WAKEUP_SOURCE_BUTTON0, NULL,␣
↪true);
#endif

#if (gAppButtonCnt_c > 1)
    /* Init and enable button1 as wake up source
     * BOARD_WAKEUP_SOURCE_BUTTON1 can be customized based on board configuration
     * On EVK we use the SW3 mapped to PTC6 */
    PM_InitWakeupSource(&button1WakeUpSource, BOARD_WAKEUP_SOURCE_BUTTON1, NULL,␣
↪true);
#endif

#if (defined(gAppUseSerialManager_c) && (gAppUseSerialManager_c > 0))

#if defined(gAppLpuart0WakeUpSourceEnable_d) && (gAppLpuart0WakeUpSourceEnable_d > 0)
    /* To be able to wake up from LPUART0, we need to keep the FRO6M running
     * also, we need to keep the WAKE domain is SLEEP.
     * We can't put the WAKE domain in DEEP SLEEP because the LPUART0 is not mapped
     * to the WUU as wake up source */
    (void)PM_SetConstraints(PM_LP_STATE_NO_CONSTRAINT, APP_LPUART0_WAKEUP_
↪CONSTRAINTS);
#endif

    /* Register PWR functions into SerialManager module in order to disable device lowpower
        during SerialManager processing. Typically, allow only WFI instruction when
        uart data are processed by serail manager  */
    SerialManager_SetLowpowerCriticalCb(&gSerMgr_LowpowerCriticalCBs);
#endif

#if defined(gAppUseSensors_d) && (gAppUseSensors_d > 0)
    Sensors_SetLowpowerCriticalCb(&app_LowpowerSensorsCriticalCBs);
#endif

    (void)status;
}
```

2.3 - low power entry/exit sequences : board files updates Board Files that handles low-power are board_lp.[ch] files.

Low power board files implement the low-power callbacks of the peripherals to be notified when entering or exiting Low Power mode. This module also registers these low-power callbacks to the SDK Power Manager component to get the notifications when the device is about to enter low-power or exit Low Power mode. The Low-power callbacks are registered from BOARD_LowPowerInit() function. This function is called from app_services_init.c file after PWR module initialization.

The low power callback functions can be categorized in two groups:

- Entry Low power call back functions: These are usually used to prepare the peripherals to enter low-power. For example, they can be used for flushing FIFOs, switching off some clocks, and reconfiguring pin mux to avoid leakage on pins. In case of Power Down mode, these functions could be used to save the Hardware peripheral context.
- Exit Low power call back functions: These are typically used to restore the peripherals to functionality. Therefore, they perform the reverse of what is done by the entry call-back functions: restoring the pin mux, re-enabling the clock, in case of Power Down mode, restoring the Hardware peripheral context, and so on.

Note that distinction can be done between clock gating mode (Deep Sleep mode), and power gated mode (Power down mode) when entering and exiting Low Power mode. The

BOARD_EnterLowPowerCb() and BOARD_ExitLowPowerCb() functions provide the code to call the various peripheral entry and exit functions to go and exit Deep Sleep mode: serial manager, button, debug console, and others.

However, the processing to save and restore the Hardware peripheral is implemented in different functions BOARD_EnterPowerDownCb() and BOARD_ExitPowerDownCb(). These two functions should be called when exiting power gated modes of the power domain. These two should implement specific code for such case (likely the complete reinitialization of each peripheral). In order to know the Low Power mode that the wake up domain, or main domain has been entered, the low-power platform API PLATFORM_GetLowpowerMode() can be called.

> **Note :** BOARD_ExitPowerDownCb() is called before BOARD_ExitLowPowerCb() as it is generally required to restore the Hardware peripheral contexts before reconfiguring the pin mux to avoid any signal glitches on the pads

Also, It is important to know whether the location of the Hardware peripheral is in the main domain or wake up domain. The two power domains can go into different power modes with the limitation that the wakeup domain cannot go to a deepest Low Power mode than the main domain. Depending on the constraint set on SDK power manager, the wake up domain could remain in active while the main domain can go to deep sleep or power down modes. In this case, the peripherals in the wake up domain does not required to be restored, as explained in the section Power Down. Likely, only pin mux reconfiguration is required in this case.

example Low power entry and exit functions shall be registered to the SDK power manager so these functions will be called when the device will enter and exit low power mode. This is done by BOARD_LowPowerInit() typically called from application source code in app_services_init.c file

```
static pm_notify_element_t boardLpNotifyGroup = {
    .notifyCallback = BOARD_LowpowerCb,
    .data           = NULL,
};

void BOARD_LowPowerInit(void)
{
    status_t status;

    status = PM_RegisterNotify(kPM_NotifyGroup2, &boardLpNotifyGroup);
    assert(status == kStatus_Success);
    (void)status;
}
```

BOARD_LowpowerCb() callback function will handle both the entry and exit sequences. An argument is passed to the function to indicate the lowpower state the device enter/exit. Typical implementation is given below. Customer shall make sure to differentiate low power entry and exit, and the various low power states.

Typically, nothing is expected to be done if low power state is WFI or Sleep mode. These modes are some light low power states and the system can be woken up by interrupt trigger.

In Deep sleep mode, the clock tree and source clocks are off, the system needs to be woken up from an event from the WUU module.

In Power down mode, some peripherals are likely to be powered off, context save and restore may need to be done in these functions.

```
static status_t BOARD_LowpowerCb(pm_event_type_t eventType, uint8_t powerState, void *data)
{
    status_t ret = kStatus_Success;
    if (powerState < PLATFORM_DEEP_SLEEP_STATE)
    {
        /* Nothing to do when entering WFI or Sleep low power state
            NVIC fully functionnal to trigger upcoming interrupts */
```

(continues on next page)

(continued from previous page)

```
    }
    else
    {
        if (eventType == kPM_EventEnteringSleep)
        {
            BOARD_EnterLowPowerCb();

            if (powerState >= PLATFORM_POWER_DOWN_STATE)
            {
                /* Power gated low power modes often require extra specific
                 * entry/exit low power procedures, those should be implemented
                 * in the following BOARD API */
                BOARD_EnterPowerDownCb();
            }
        }
        else
        {
            /* Check if Main power domain domain really went to Power down,
             *   powerState variable is just an indication, Lowpower mode could have been skipped by an␣
↪immediate wakeup
             */
            PLATFORM_PowerDomainState_t main_pd_state = PLATFORM_NO_LOWPOWER;
            PLATFORM_status_t           status;

            status = PLATFORM_GetLowpowerMode(PLATFORM_MainDomain, &main_pd_state);
            assert(status == PLATFORM_Successful);
            (void)status;

            if (main_pd_state == PLATFORM_POWER_DOWN_MODE)
            {
                /* Process wake up from power down mode on Main domain
                 *  Note that Wake up domain has not been in power down mode */
                BOARD_ExitPowerDownCb();
            }

            BOARD_ExitLowPowerCb();
        }
    }
    return ret;
}
```

2.4 - Low power constraint updates and optimization Except for the board file update as seen in previous section, the application does not need any other changes for low-power support in Deep Sleep mode. It shall work as if no low-power is supported. However, If more aggressive power saving is required, this constraint can be changed in your application in order to further reduce the power consumption in Low Power mode.

2.4.1 - Changing the Default Application low power constraint after firmware initialization The Low power reference design applications (central or peripheral) provides demonstration on how to change the Application low power constraint. In the Application main entry point BluetoothLEHost_AppInit(), Deep Sleep mode is configured by default from APP_ServiceInitLowpower() function.

> **Note :** It is recommended to keep Deep Sleep mode as default during all the stack initialization phase until BluetoothLEHost_Initialized() and BleApp_StartInit() functions are called. In case of Bonded device with privacy, it is recommended to wait for gControllerPrivacyStateChanged_c event to be called.

BleApp_LowpowerInit() function provides an example of code on how to release the default Deep sleep low-power constraint and set a new constraint such as Power down mode for the application. This deeper low-power mode is used when no Bluetooth LE activity is on going, and if no other higher Low-power constraint is set by another components or layer. For instance, if some serial transmission is on going by the serial manager, or if the SecLib module has on going activity on the HW crypto accelerator, the low-power mode could less deep.

```
static void BleApp_LowpowerInit(void)
{
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
    PWR_ReturnStatus_t status;

    /*
     * Optionally, Allow now Deepest lowpower mode constraint given by gAPP_
↪LowPowerConstraintInNoBleActivity_c
     *    rather than DeepSleep mode.
     * Deep Sleep mode constraint has been set in APP_InitServices(), this is fine
     *    to keep this constraint for typical lowpower application but we want the
     *     lowpower reference design application to be more agressive in term of power saving.

     *   To apply a lower lowpower mode than Deep Sleep mode, we need to
     *     - 1) First, release the Deep sleep mode constraint previously set  by default in app_services_init()
     *     - 2) Apply new lowpower constraint when No BLE activity
     *   In the various BLE states (advertising, scanning, connected mode), a new Lowpower
     *     mode constraint will be applied depending of Application Compilation macro set in app_preinclude.
↪h :
     *     gAppPowerDownInAdvertising, gAppPowerDownInConnected, gAppPowerDownInScanning
     */

    /*  1) Release the Deep sleep mode constraint previously set  by default in app_services_init() */
    status = PWR_ReleaseLowPowerModeConstraint(PWR_DeepSleep);
    assert(status == PWR_Success);
    (void)status;

    /*  2) Apply new Lowpower mode constraint gAppLowPowerConstraintInNoBleActivity_c *
     *       The BleAppStart() call above has already set up the new lowpower constraint
     *         when Advertising request has been sent to controller          */
    BleApp_SetLowPowerModeConstraint(gAppLowPowerConstraintInNoBleActivity_c);
#endif
}
```

2.4.2 - Changing the Application lowest low power constraint during application execution

In the various application use cases, (in the various Bluetooth LE activity states, advertising, connected, scanning), some lower low-power constraint can be set, as Power down for advertising, Deep Sleep for connected, or Scanning. Customer can change the level of Low Power mode in the various use case mainly depending of the time duration the device is supposed to remain in low-power. The longer the time that the device remains in low power, the higher the benefit for a deeper Low Power mode such as Power down mode. However, please note that the wake up from power down mode takes significantly more time than deep sleep as ROM code is re executed and the hardware logic needs to be restored. Sections Deep Sleep and Power Down provide some guidance on when to use Deep Sleep mode or Power Down modes respectively.

In the low power reference design applications, four application compilations macros are defined to adjust the low-power mode into advertising, scanning, connected, or no Bluetooth LE activity. Other use cases can be added as desired. For instance, If application needs to run a DMA transfer, or if application needs to wakeup regularly to process data from external device, it may be useful to set WFI constraint (in case of DMA transfer), or Deep Sleep constraint (in case of regular wake up to process external data), rather than power down or a even lower low-power mode.

The 4 application compilation macros can be found in app_preinclude.h file of the project. See app_preinclude.h for low power reference design peripheral application :

```
/*! Lowpower Constraint setting for various BLE states (Advertising, Scanning, connected mode)
    The value shall map with the type defintion PWR_LowpowerMode_t in PWR_Interface.h
      0 : no LowPower, WFI only
      1 : Reserved
      2 : Deep Sleep
      3 : Power Down
      4 : Deep Power Down
    Note that if a Ble State is configured to Power Down mode, please make sure
       gLowpowerPowerDownEnable_d variable is set to 1 in Linker Script
    The PowerDown mode will allow lowest power consumption but the wakeup time is longer
       and the first 16K in SRAM is reserved to ROM code (this section will be corrupted on
       each power down wakeup so only temporary data could be stored there.)
           Power down feature not supported. */

#define gAppLowPowerConstraintInAdvertising_c          3
/* Scanning not supported on peripheral */
//#define gAppLowPowerConstraintInScanning_c              2
#define gAppLowPowerConstraintInConnected_c            2
#define gAppLowPowerConstraintInNoBleActivity_c        4
```

In lowpower_central.c lowpower_preripheral.c files, the application sets and releases the low power constraint from BleApp_SetLowPowerModeConstraint() and BleApp_ReleaseLowPowerModeConstraint() functions. These functions are called with the macro value passed as argument.

> **Important Note :** Setting the application low power constraint shall be done on new Bluetooth LE state request so the new constraint is applied immediately, while the application low-power mode constraint shall be released when the Bluetooth LE state is exited. For example, setting the new low power constraint for Advertising shall be done when the application requests advertising to start. Releasing the low power constraint shall be done in the advertising stop callback (advertising has been stopped).

After releasing the low power constraint, the previous low power constraint, (likely the one that has been set during firmware initialization in APP_ServiceInitLowpower() function, or the updated low power constraint in BleApp_StartInit() function) applies again.

2.4.3 - Idle task implementation example

2.4.3.1 Tickless mode support and Low power entry function

Idle task configuration from FreeRTOS shall be enabled by configUSE_TICKLESS_IDLE in FreeRTOSConfig.h. This will have the effect to have vPortSuppressTicksAndSleep() called from Idle task created by FreeRTOS. Here is a typical implementation of this function:

```
void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime)
{

    bool abortIdle = false;
    uint64_t actualIdleTimeUs, expectedIdleTimeUs;

    /* The OSA_InterruptDisable() API will prevent us to wakeup so we use
     * OSA_DisableIRQGlobal() */
    OSA_DisableIRQGlobal();

    /* Disable and prepare systicks for low power */
    abortIdle = PWR_SysticksPreProcess((uint32_t)xExpectedIdleTime, &expectedIdleTimeUs);

    if (abortIdle == false)
    {
```

(continues on next page)

(continued from previous page)

```
            /* Enter low power with a maximal timeout */
            actualIdleTimeUs = PWR_EnterLowPower(expectedIdleTimeUs);

            /* Re enable systicks and compensate systick timebase */
            PWR_SysticksPostProcess(expectedIdleTimeUs, actualIdleTimeUs);
    }

    /* Exit from critical section */
    OSA_EnableIRQGlobal();
}
```

2.4.3.2 Connectivity background tasks and Idle hook function example

Some process needs to be run in background before going into low power. This is the case for writing in NVM, or firmware update OTA to be writen in Flash. If so, configUSE_IDLE_HOOK shall be enabled in FreeRTOSCOnfig.h so vApplicationIdleHook() will be called prior to vPortSuppressTicksAndSleep(). Typical implementation of vApplicationIdleHook() function can be found here :

```
void vApplicationIdleHook(void)
{
    /* call some background tasks required by connectivity */
#if ((gAppUseNvm_d) || \
    (defined gAppOtaASyncFlashTransactions_c && (gAppOtaASyncFlashTransactions_c > 0)))

    if (PLATFORM_CheckNextBleConnectivityActivity() == true)
    {
        BluetoothLEHost_ProcessIdleTask();
    }
#endif
}
```

PLATFORM_CheckNextBleConnectivityActivity() function implemented in low power platform file fwk_platform_lowpower.c typically checks the next connectivity event and returns true if there's enough time to perform time consuming tasks such as flash erase/write operations (can be defined by the compile macro depending on the platform).

2. Low power features

2.1 - FreeRTOS systicks

Low power module in framework supports the systick generation for FreeRTOS. Systicks in FreeRTOS are most of the time not required in the Bluetooth LE demos applications because the framework already supports timers by the timer manager component, so the application can use the timers from this module. The systicks in FreeRTOS are useful for all internal timer service provided by FreeRTOS (through OSA) like OSA_TimeDelay(), OSA_TimeGetMsec(), OSA_EventWait(). When systicks are enabled, an interrupt (systick interrupt) is triggered and executed on a periodic basis. In order to save power, periodic systick interrupts are undesirable and thus disabled when going to low-power mode. This feature is called low power FreeRTOS tickless mode. When entering the low power state, the system ticks shall be disabled and switch to a low power timer. On wake-up, the module retrieves the time passed in low power and compensate the ticks count accordingly. This feature does not apply on bare metal scheduler.

On FreeRTOS, the vPortSuppressTicksAndSleep() function implemented in the app_low_power.c file will be called when going to idle. FreeRTOS will give to this function the xExpectedIdleTime, time in tick periods before a task is due to be moved into the Ready state. This function will manage the systicks (disable/enable) through PWR_SysticksPreProcess() and PWR_SysticksPostProcess() calls. Then, when calling PWR_EnterLowPower(), a time out duration in micro seconds will be given and the function will set a timer before entering low power.

In addition, this function will return the low power period duration, used to compensate the ticks count.

In our example low power reference design peripheral application, an OSA_EventWait() has been added to demonstrate the tickless mode feature. You can adjust the timeout with the gAppTaskWaitTimeout_ms_c flag in the app_preinclude.h file, its value in our demo is 8000ms. So 8 seconds after stopping any activity we will wake up from low power. If the flag is not defined in the application its value will be osaWaitForever_c and there will be no OS wake up.

2.2 - Selective RAM bank retention To optimize the consumption in low power, the linker script specific function PLATFORM_GetDefaultRamBanksRetained() is implemented. This function obtains the RAM banks that need to be retained when the device goes in low power, in order to set them with PLATFORM_SetRamBanksRetained() function. The RAM banks that are not needed are set in power off state, when the device goes in low power mode.

The function PLATFORM_GetDefaultRamBanksRetained() is linker script specific. Hence, it cannot be adapted for a different application. If these functions are called from board_lp.c, it is possible to give to PLATFORM_SetRamBanksRetained() a different bank_mask adapted to your specific application.

In deep power down, this feature does not have any impact because in this power mode, all RAM banks are already powered off.

3 - Low power modes overview PWR module API provides the capability to set low power mode constraints from various components or from the application. These constraints are provided to the SDK power manager. Upper layer (all Application code, connectivity stacks, etc.) can call directly the SDK Power Manger if it requires more advanced tuning. The PWR API can be found in PWR_Interface.h.

> **Note :** 'Upper layer' signifies all layers, applications, components, or modules that are above the connectivity framework in the Software architecture.

> **Note :** Each power domain has its own Low Power mode capability. The Low Power modes described below are for the main domain and it is supposed that the wake up domain goes to the same Low Power mode. This is not always true as the wake up domain that contains some wake up peripheral can go a lower Low Power mode state than the main domain so the peripherals in the wake up domain can remain operational when the main domain is in Low Power mode (deep sleep or power down modes). In this case, the context of the Hardware peripheral located in the wake up domain does not need to be saved and restored as for the peripherals located in the main domain

3.1 Wait for Interrupt (WFI) **Definition**

In the Wait for Interrupt (WFI) state, the CPU core is powered on, but is in an idle mode with the clock turned OFF.

Wake up time and typical use case

The wakeup time from this Low Power mode is insignificant because the Fast clock from FRO is still running.

This Low Power mode is mainly used when there is an hardware activity while the Software runs the Idle task. This allows the code execution to be temporarily suspende, thus reducing a bit the power consumption of the device by switching off the processor clock. When an interrupt fires, the processor clock is instantaneously restored to process the Interrupt Service Routine (ISR).

Usage

In order to prevent the software from programming the device to go to a lower Low Power mode (such as Deep Sleep, Power Down mode or Deep Power Down mode), the component responsible for the hardware drivers shall call PWR_SetLowPowerModeConstraint(PWR_WFI) function. When the Hardware activity is completed, the component shall release the constraint by calling PWR_ReleaseLowPowerModeConstraint(PWR_WFI).

Alternatively, the component can call PWR_LowPowerEnterCritical() and then PWR_LowPowerExitCritical() functions.

For fine tuning of the Low Power mode allowing more power saving, the component can call directly the SDK power manager API with PM_SetConstraints() function using the appropriate Low Power mode and low power constraint. However, this is reserved for more advanced user that knows the device very well. It is not recommended to do so.

The PWR module has no external dependencies, so the low-power entry and exit callback functions must be defined by the user for each peripheral that has specific low power constraints It is consequently convenient to register to the component the low power callbacks structure that is used for entering and exit low power critical sections. In Bluetooth LE, you can take the example in the app_conn.c file as shown here :

```
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
static const Seclib_LowpowerCriticalCBs_t app_LowpowerCriticalCBs =
{
   .SeclibEnterLowpowerCriticalFunc = &PWR_LowPowerEnterCritical,
   .SeclibExitLowpowerCriticalFunc  = &PWR_LowPowerExitCritical,
};
#endif

void BluetoothLEHost_Init(..)
{
  ...
        /* Cryptographic hardware initialization */
        SecLib_Init();
#if defined(gAppLowpowerEnabled_d) && (gAppLowpowerEnabled_d>0)
        /* Register PWR functions into SecLib module in order to disable device lowpower
            during Seclib processing. Typically, allow only WFI instruction when
            commands (key generation, encryption) are processed by Seclib  */
        SecLib_SetLowpowerCriticalCb(&app_LowpowerCriticalCBs);
#endif
  ...
}
```

Limitations

No limitation when using the WFI mode.

3.2 Sleep mode Sleep mode is similar to WFI low power mode but with some additional clock gating. The Sleep mode is device specific, please consult the Hardware reference manuel of the device for more information.

3.2 Deep Sleep mode **Definition**

In Deep Sleep mode, the fast clock is turned off, and the CPU along with the main power domain are placed into a retention state, with the voltage being scaled down to support state retention only. Because no high frequency clock is running, the voltage applied on the power domain can be reduced to reduce leakage on the hardware logic. This reduces the overall power consumption in the Deep Sleep mode. When waking up from Deep sleep mode, the core voltage is increased back to nominal voltage and the fast clock (FRO) is turned back on, the peripheral in this domain can be reused as normal.

To same more additional power, Some unused RAM banks can be powered off. this prevents from having current leakage and consequently, allow to reduce even more the power consumption in Deep SLeep mode. This is achieved by calling PLATFORM_SetRamBanksRetained() from low power entry function from board_lp.c file.

Usage

All firmware is able to implement Deep Sleep mode transparently to the application thanks to the PWR module, low power platform submodule and low power board file. This is described in the section Low-power implementation.

When entering this mode, it is recommended to turn the output pins into input mode, or high impedance to reduce leakage on the pads. This is typically done in pin_mux.c file, called from board.c file and executed from the low power callback in board_lp.c file. As an example, the TX line of the UART peripheral can be turned to disabled so it prevents the current from being drawn by the pad in Low Power mode.

Wake up time and typical use case

The wake up time is very fast, it takes mostly the time for the Fast FRO to start up again (couple of hundreds of microseconds) so this mode is a very good balance between power consumption in low-power mode and wake up latency and shall be used extensively in most of the use cases of the application.

Limitations

In Deep Sleep mode, the clock is disabled to the CPU and the main peripheral domain, so peripheral activity (for example, an on-going DMA transfer) is not possible in Deep Sleep mode.

3.3 Power Down mode

Definition

In Power Down mode, both the clock, and power are shut off to the CPU and the main peripheral domain. SRAM is retained, but register values are lost. The SDK power manager handles the restore of the processor registers and dependencies such as interrupt controller and similar ones transparently from the application.

Usage

The application, with the help of the low power board files, saves and restores the peripherals that were located in the power domain during the entry and exit of the power down mode. This is done from low power board_lp files in the entry/exit low power callbacks. Example is given for the serial manager and debug console in *board_lp.c* file in function *BOARD_ExitPowerDownCb()*.

If the device contains a dedicated wake up power domain where some wake up peripherals are located, if this wake up domain is not turned into power down mode but only Deep sleep mode or active mode, this peripheral does not need for a save and restore on low power entry/exit. For instance, on KW45, This is basically achieved when enabling the wakeup source of the peripheral *PWR_EnableWakeUpSource()* from *APP_ServiceInitLowpower()* function. Alternatively, this can be directly achieved by setting the constraint to the SDK power manager by calling *PM_SetConstraints()*, (use APP_LPUART0_WAKEUP_CONSTRAINTS for wakeup from UART constraint).

On exit from low power, The low power state of power domain can be retrieved by Platform API PLATFORM_GetLowpowerMode(). This API shall be called from low power exit callback function only.

As for Deep Sleep mode, software shall configure the output pins into input or high impedance during the Low Power mode to avoid leakage on the pads.

Wake up time and typical use case

The wake up time is significantly longer than wake up time from Deep Sleep (from several hundreds of micro-seconds to a couple of milliseconds depending on the platform). On some platform, it can takes longer, for instance, if ROM code is implemented and perform authentication checks for security and hardware logic in power domain needs to be restored (case for KW45).

However, After ROM code execution, the SDK power manager resumes the Idle task execution from where it left before entering low-power mode. Hence, the wakeup time from this mode is still significantly lower that the initialization time from a power on reset or any other reset.

Depending on the wakeup time of the platform and the low power time duration, This mode is recommended when no Software activity is expected to happen for the next several seconds. In Bluetooth LE, this mode is preferred in advertising or without Bluetooth LE activity. However, in scanning or connected mode, Regular wakes up happens regularly for instance to retrieve HCI message responses from the Link layer, the Deep Sleep mode is rather recommended.

Limitations

In addition to the Deep Sleep limitation (no Hardware processing on going when going to Power down mode) and the significant increase of the wake time, the Power Down mode requires the ROM code to execute and this last uses significant amount of memory in SRAM.

Typically, The first SRAM bank (16 KBytes) is used by the ROM code during execution so the Application firmware can use this section of SRAM for storing bss, rw data, or stacks. Only temporary data could be stored here and this location is overwritten on every Power Down exit sequence.

In order to avoid placing firmware data section (bss, rw, etc.) in the first SRAM bank, the linker script variable *gLowpowerPowerDownEnable_d* should be set to 1. Setting the linker script variable to avoid placing firmware data section in the first SRAM bank, The effect of setting this flag is to prevent the firmware from using the first 16 KB in SRAM.

> *Note* : This setting is ONLY required if the application implements Power Down mode. If Application uses other low-power mode, this is not required.

3.4 Deep Power-down mode **Definition**

In Deep Power Down mode, the SRAM is not retained. This power mode is the lowest disponible, it is exited through reset sequence.

Usage

In addition to the Power Down limitation, the Deep Power Down mode shut down all memory in SRAM. Because it is exited through reset sequence the wake time is also longer.

Wake up time and typical use case

As this low-power mode is exited through the reset sequence, the wake up time is longer than any other mode. In Bluetooth LE, this mode is possible in no Bluetooth LE activity, and is preferred if we know that there will be no Bluetooth LE activity before a several amount of time.

Limitations

All memory in SRAM will be shut down in deep power down, the main limitation in going in this low-power mode is that the context will not be saved.

ModuleInfo

Overview

The ModuleInfo is a small Connectivity Framework module that provides a mechanism that allows stack components to register information about themselves.

The information comprises :

- Component or module name (for example: Bootloader, IEEE 802.15.4 MAC, and Bluetooth LE Host) and associated version string
- Component or module ID
- Version number
- Build number

The information can be retrieved using shell commands or FSCI commands.

Detailed data types and APIs used in ConnFWK_APIs_documentation.pdf.

NVM: Non-volatile memory module

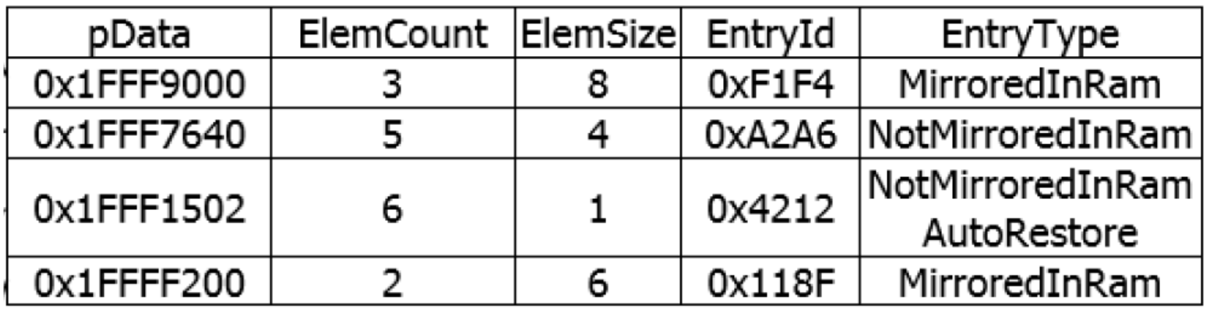
Overview In a standard Harvard-architecture-based MCU, the flash memory is used to store the program code and program constant data. Modern processors have a built-in flash memory controller that can be used under user program execution to store non-volatile data. The flash memories have individually erasable segments (sectors) and each segment has a limited number of erase cycles. If the same segments are used to store various kinds of data all the time, those segments quickly become unreliable. Therefore, a wear-leveling mechanism is necessary to prolong the service life of the memory. The NVM module in the connectivity framework provides a file system with a wear-leveling mechanism, described in the subsequent sections. The *NvIdle()* function handles the program and erase memory operations. Before resetting the MCU, *NvShutdown()* must be called to ensure that all save operations have been processed.

NVM boundaries and linker script requirement Most of the MCUs have only a standard flash memory that the non-volatile (NV) storage system uses. The amount of memory that the NV system uses for permanent storage and its boundaries are defined in the linker configuration file though the following linker symbols :

- NV_STORAGE_START_ADDRESS
- NV_STORAGE_END_ADDRESS
- NV_STORAGE_MAX_SECTORS
- NV_STORAGE_SECTOR_SIZE

The reserved memory consists of two virtual pages. The virtual pages are equally sized and each page is using one or more physical flash sectors. Therefore, the smallest configuration is using two physical sectors, one sector per virtual page.

NVM Table The Flash Management and Non-Volatile Storage Module holds a pointer to a RAM table. The upper layers of this table register information about data that the storage system should save and restore. An example of NVM table entry list is given below.



pData	ElemCount	ElemSize	EntryId	EntryType
0x1FFF9000	3	8	0xF1F4	MirroredInRam
0x1FFF7640	5	4	0xA2A6	NotMirroredInRam
0x1FFF1502	6	1	0x4212	NotMirroredInRam AutoRestore
0x1FFFF200	2	6	0x118F	MirroredInRam

NVM Table entry As show above, A NVM table entry contains a generic pointer to a contiguous RAM data structure, the number of elements the structure contains, the size of a single element, a table entry ID, and an entry type.

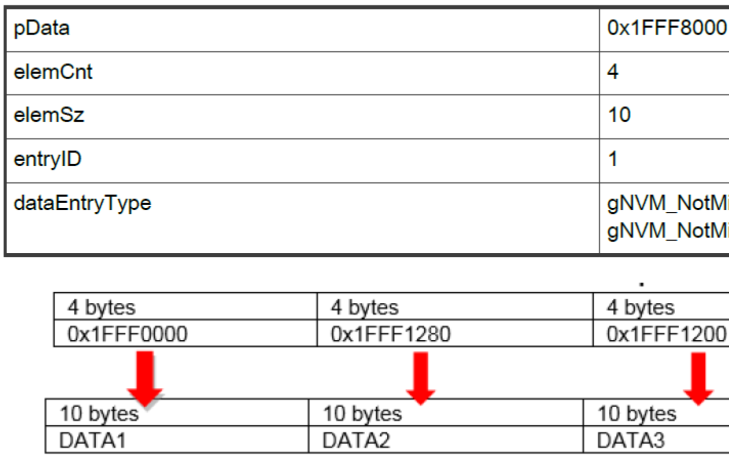
A RAM table entry has the following structure:

- pData (4 bytes) is a pointer to the RAM memory location where the dataset elements are stored.

- elemCnt (2 bytes) represents how many elements the dataset has.
- elemSz (2 bytes) is the size of a single element.
- entryID is a 16-bit unique ID of the dataset.
- dataEntryType is a 16-bit value representing the type of entry (mirrored/unmirrored/unmirrored auto restore).

For mirrored datasets, pData must point directly to the RAM data. For unmirrored datasets, it must be a double pointer to a vector of pointers. Each pointer in this table points to a RAM/FLASH area. Mirrored datasets require the data to be permanently kept in RAM, while unmirrored datasets have dataset entries either in flash or in RAM. If the unmirrored entries must be restored at the initialization, NotMirroredInRamAutoRestore should be used. The entryID gUnmirroredFeatureSet_d should be set to 1 for enabling unmirrored entries in the application. The last entry in the RAM table must have the entryID set to gNvEndOfTableId_c.

The figure below provides an example of table entry :



pData	0x1FFF8000
elemCnt	4
elemSz	10
entryID	1
dataEntryType	gNVM_NotM gNVM_NotM

When the data pointed to by the table entry pointer (pData) has changed (entirely or just a single element), the upper layers call the appropriate API function that requests the storage system to save the modified data. All the save operations (except for the synchronous save and atomic save) and the page erase and page copy operations are performed on system idle task. The application must create a task that calls NvIdle in an infinite loop. It should be created with OSA_PRIORITY_IDLE. However, the application may choose another priority. The save operations are done in one virtual page, which is the active page. After a save operation is performed on an unmirrored dataset, pData points to a flash location and the RAM pointer is freed. As a result, the effective data should always be allocated using the memory management module.

Active page The active page contains information about the records and the records. The storage system can save individual elements of a table entry or the entire table entry. Unmirrored datasets can only have individual saves. On mirrored datasets, the save/restore functions must receive the pointer to RAM data. For example, if the application must save the third element in the above vector, it should send 0x1FFF8000 + 2 * elemSz. For unmirrored datasets, the application must send the pointer that points to the area where the data is located. For example, if the application must save the third element in the above vector, it should send 0x1FFF8000 + 2 * sizeof(void*).

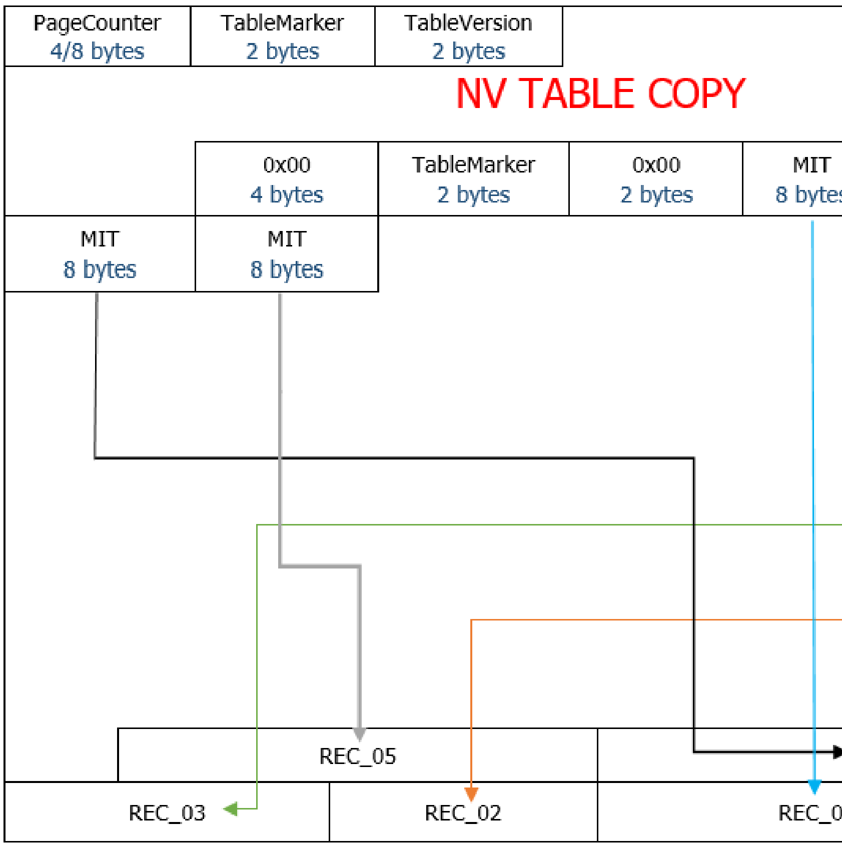
The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page erase operation is performed when the system is formatted. It is also performed when the page is full and a new record cannot be written into that page. Before being erased, the full page is first copied (only the most recent saves) and erased afterward.

The validity of the Meta Information Tag (MIT), and, therefore, of a record, is guaranteed by the MIT start and stop validation bytes. These two bytes must be equal to consider the record referred by the MIT valid. Furthermore, the value of these bytes indicates the type of the record, whether it is a single element or an entire table entry. The nonvolatile storage system allows dynamic changes of the table within the RAM memory, as follows:

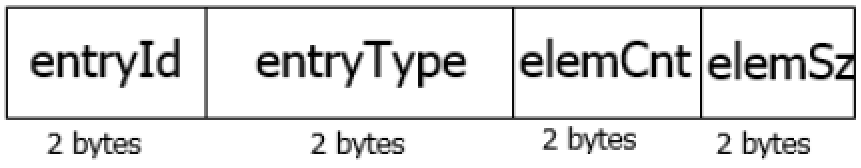
- Remove table entry
- Register table entry

A new table entry can be successfully registered if there is at least one entry previously removed or if the NV table contains uninitialized table entries, declared explicitly to register new table entries at run time. A new table entry can also replace an existing one if the register table entry is called with an overwrite set to true. This functionality is disabled by default and must be enabled by the application by setting gNvUseExtendedFeatureSet_d to 1.

The layout of an active page is shown below:



As shown above, the table stored in the RAM memory is copied into the flash active page, just after the table version. The "table start" and "table end" are marked by the table markers. The data pointers from RAM are not copied. A flash copy of a RAM table entry has the following structure:

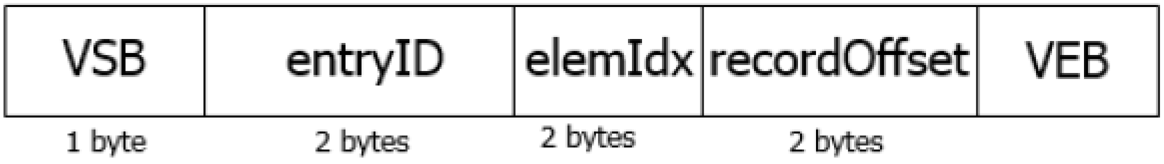


Where:

- entryID is the ID of the table entry
- entryType represents the type of the entry (mirrored/unmirrored/unmirrored auto restore)
- elemCnt is the elements count of that entry
- elemSz is the size of a single element

This copy of the RAM table in flash is used to determine whether the RAM table has changed. The table marker has a value of 0x4254 ("TB" if read as ASCII codes) and marks the beginning and end of the NV table copy.

After the end of the RAM table copy, the Meta Information Tags (MITs) follow. Each MIT is used to store information related to one record. An MIT has the following structure:

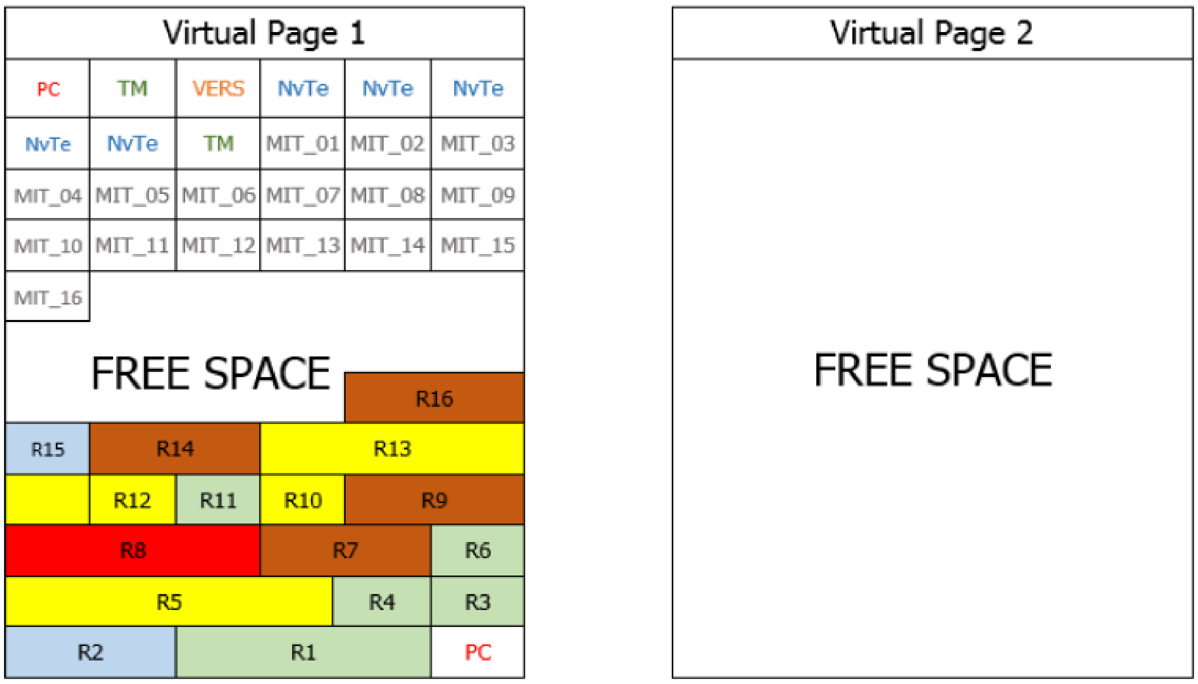


Where:

- VSB is the validation start byte.
- entryID is the ID of the NV table entry.
- elemIdx is the element index.
- recordOffset is the offset of the record related to the start address of the virtual page.
- VEB is the validation end byte.

A valid MIT has a VSB equal to a VEB. If the MIT refers to a single-element record type, VSB=VEB=0xAA. If the MIT refers to a full table entry record type (all elements from a table entry), VSB=VEB=0x55. Because the records are written to the flash page, the available page space decreases. As a result, the page becomes full and a new record does not have enough free space to be copied into that page.

In the example given below, the virtual page 1 is considered to be full if a new save request is pending and the page free space is not sufficient to copy the new record and the additional MIT. In this case, the latest saved datasets (table entries) are copied to virtual page 2.

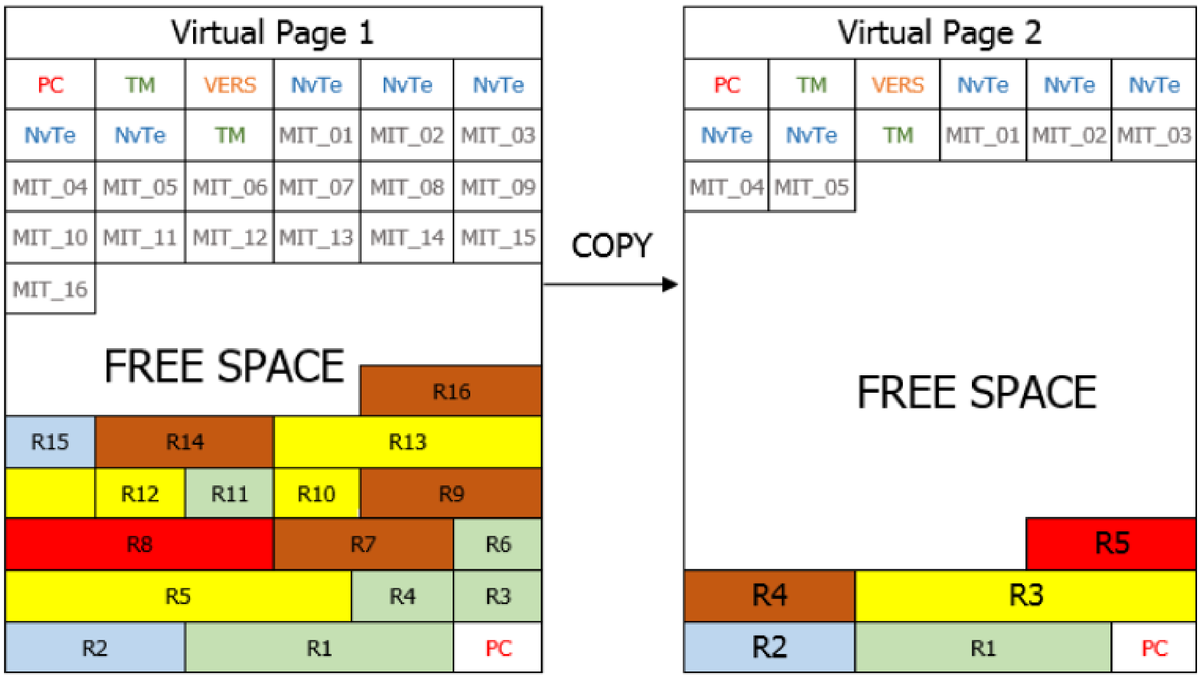


In this example, there are five datasets (one color for each dataset) with both 'full' and 'single' record types.

- R1 is a 'full' record type (contains all the NV table entry elements), whereas R3, R4, R6 and R11 are 'single' record types.
- R2 – full record type; R15 – single record type
- R5, R13 – full record type; R10, R12 – single record type

- R8 – full record type
- R7, R9, R14, R16 – full record type

As shown above, the R3, R4, R6, and R11 are 'single' record types, while R1 is a 'full' record type of the same dataset. When copied to virtual page 2, a defragmentation process takes place. As a result, the record copied to virtual page 2 has as much elements as R1, but individual elements are taken from R3, R4, R6, and R11. After the copy process completes, the virtual page 2 has five 'full' record types, one for each dataset. |This is illustrated below:



Finally, the virtual page 2 is validated by writing the PC value and a request to erase virtual page 1 is performed. The page is erased on an idle task, sector by sector where only one sector is erased at a time when idle task is executed.

If there is any difference between the RAM and flash tables, the application must call RecoverNvEntry for each entry that is different from its RAM copy to recover the entry data (ID, Type, ElemSz, ElemCnt) from flash before calling NvInit. The application must allocate the pData and change the RAM entry. It can choose to ignore the flash entry if the entry is not desired. If any entry from RAM differs from its flash equivalent at initialization, a page copy is triggered that ignores the entries that are different. In other words, data stored in those entries is lost.

The application can check if the RAM table was updated. In other words, if the MCU program was changed and the RAM table was updated, using the function GetFlashTableVersion and compare the result with the constant gNvFlashTableVersion_c. If the versions are different, NvInit detects the update and automatically upgrades the flash table. The upgrade process triggers a page copy that moves the flash data from the active page to the other one. It keeps the entries that were not modified intact and it moves the entries that had their elements count changed as follows:

- If the RAM element count is smaller than the flash element count, the upgrade only copies as many elements as are in RAM.
- If the RAM element count is larger than the flash element count, the upgrade copies all data from flash and fills the remaining space with data from RAM. If the entry size is changed, the entry is not copied. Any entryIds that are present in flash and not present in RAM are also not copied. This functionality is not supported if gNvUseExtendedFeatureSet_d is not set to 1.

ECC Fault detection The KW45/K32W1 internal flash is organized in 16 byte phrases and 8kB sectors (minimal erase unit). Its flash controller is synthesized so that it generates ECC information and an ECC generator / checker. During the programming of internal flash, errors may accidentally happen and cause ECC errors as a flash phrase is being written. These may happen due to multiple reasons:

- programmatic errors such as overwriting an already programmed phrase (transitioning bits from 0b to 1b). These are evitable by performing a blank check verification over phrase to be programmed, at the expense of processing power.
- occurrence of power drop or glitches during a programming operation.
- excessive wear of flash sector. The flash controller is capable of correcting one single ECC error but raises a bud fault whenever reading a phrase containing more than one ECC fault. Once an ECC error has 'infected' a flash phrase, the fault will remain and raise again at each read operation over the same phrase including blank check and prefetch. It can only be rid of by erasing the whole flash sector that contained the faulty phrase. In order to recover from situations where an ECC fault has occurred a gNvSalvageFromEccFault_d option has been added, which forces gNvVerifyReadBackAfterProgram_d to be defined to TRUE. If defined, the gNvVerifyReadBackAfterProgram_d option of the NVM module, causes the program to read back the programmed area after every flash programming operation. The verification is performed in safe mode if gNvSalvageFromEccFault_d is also defined. This is so as to detect ECC faults as early as possible as they appear, indeed when verifying a programming operation, one cannot be certain of the absence of ECC fault and avoid the bus fault. The safe API is thence used to perform the read back operation is performed using this safe API, so that we can tread in the flash and detect potential errors. The defects are detected on the fly whereas in the absence of safe read back, the error would cause a fault, potentially much later. During normal operation, assuming that no chip reset was provoked, this will consist in a single ECC fault either in the last record data or its meta information. Detecting such a fault calls for an immediate page copy to the other virtual page, so that the currently active page gets erased and the error gets cleared. Should the ECC fault occurs in the middle of a page copy operation, the switch of active page is postponed so that the fault page can be erased again and the copy can be restarted.

If the system underwent a power drop during a flash programming operation, sufficient to provoke a reset, at the ensuing reboot, ECC fault(s) may be present in the NVM area at the location that was being written. The detection is performed by an NVM sweeping mechanism, using the safe read API. That marks the faulty virtual page so that all subsequent reads within this virtual page are done with the safe API. If this case arises, a copy of the valid contents of the faulty page is attempted to the other virtual page. At NVM initialization, faults should be detected, either at the top of the meta data or at the bottom of the record area within the previous active page. This should guarantee that only the latest record write operation may be impaired. When the page copy has taken place, the faulty page is erased and the execution may resume. During NvCopyPage, when 'garbage collecting' occurs or whenever the current virtual active page needs to be transferred to the other virtual page, ECC errors are intercepted so that the operation can be attempted again in case of error. In case of NVM contents clobbering by programming errors, the salvage operation does its best to rescue as many records as possible but data will inevitably be lost.

An additional option -namely *gInterceptEccBusFaults_d* - was introduced in order to catch and correct ECC faults at Bus Fault handler level. Indeed, should an ECC bus fault fire, in spite of the precautions taken with NVM's gNvSalvageFromEccFault_d, we verify if the fault belongs to the NV storage. If so, a drastic policy can be adopted consisting in an erasure of the faulty sector. The corresponding Bus Fault handling is not part of the NVM, but dwells in the framework platform specific sources. Alternative handling could be implemented by the customer.

Save policy: Execution of program and erase operations on a flash an MCU core fetches code from cause perturbations of the core activity or requires to place critical code in RAM so that real-time ISR can still be served. The penalty of a sector erase is much higher than a simple program operation. The NVM is designed so as to limit the erase operations at 'garbage collecting' time, so that flash wear is limited and no time is wasted. Several write policies are implemented to cope with the application constraints, one synchronous mode API and several posted write APIs. Among the posted write policies, the gNvmSaveOnIdleTimerPolicy_d compilation option selects a mode where flash write operations occur at time interval within the Idle task. Another option exists to 'randomize' the time interval with some jitter.

1) NvSyncSave performs a write synchronously with the disadvantage of stalling processor activity until comp
2) NvSaveOnCount posts a pending write operation and postpones the actual flash operation until number of record updates has reached a maximum. The actual write happens during Idle Task execution.see NvSetCountsBetweenSaves related API.
3) NvSaveOnInterval: posts a pending write operation and postpones the actual flash operation until the predefined number of ticks has elapsed. Optional mode - Active if (gNvmSaveOnIdleTimerPolicy_d & gNvmUseSaveOnTimerOn_c). see NvSetMinimumTicksBetweenSaves related API. Note that gNvmUseSaveIntervalJitter_c policy is a sub-option of gNvmSaveOnIdleTimerPolicy_d used to randomize slightly the time at which the write operation will happen.

Constant macro definition

- *gNvStorageIncluded_d* : If set to TRUE, it enables the whole functionality of the nonvolatile storage system. By default, it is set to FALSE (no code or data is generated for this module).
- *gNvUseFlexNVM_d* : If set to TRUE, it enables the FlexNVM functionality of the nonvolatile storage system. By default, it is set to FALSE. If FlexNVM is used, the standard nonvolatile storage system is disabled.
- *gNvFragmentation_Enabled_d* : Macro used to enable/disable the fragmented saves/restores (a particular element from a table entry can be saved or restored). It is set to FALSE by default.
- *gNvUseExtendedFeatureSet_d* : Macro used to enable/disable the extended feature set of the module:
  - Remove existing NV table entries
  - Register new NV table entries
  - Table upgrade

  It is set to FALSE by default.
- *gUnmirroredFeatureSet_d* : Macro used to enable unmirrored datasets. It is set to 0 by default.
- *gNvTableEntriesCountMax_c* : This constant defines the maximum count of the table entries (datasets) that the application is going to use. It is set to 32 by default.
- *gNvRecordsCopiedBufferSize_c* : This constant defines the size of the buffer used by the page copy function, when the copy operation performs defragmentation. The chosen value must be bigger than the maximum number of elements stored in any of the table entries. It is set by default to 64.
- *gNvCacheBufferSize_c* : This constant defines the size of the cache buffer used by the page copy function, when the copy operation does not perform defragmentation. The chosen value must be a multiple of 8. It is set by default to 64.
- *gNvMinimumTicksBetweenSaves_c* : This constant defines the minimum timer ticks between dataset saves (in seconds). It is set to 4 by default.
- *gNvCountsBetweenSaves_c* : This constant defines the number of calls to 'NvSaveOnCount' between dataset saves. It is set to 256 by default.

- *gNvInvalidDataEntry_c* : Macro used to mark a table entry as invalid in the NV table. The default value is 0xFFFFU.
- *gNvFormatRetryCount_c* : Macro used to define the maximum retries count value for the format operation. It is set to 3 by default.
- *gNvPendingSavesQueueSize_c* : Macro used to define the size of the pending saves queue. It is set to 32 by default.
- *gFifoOverwriteEnabled_c* : Macro used to enable overwriting older entries in the pending saves queue (if it is full). If it is FALSE and the queue is full, the module tries to process the oldest save in the queue to free a position. It is set to FALSE by default.
- *gNvMinimumFreeBytesCountStart_c* : Macro used to define the minimum free space at initialization. If the free space is smaller than this value, a page copy is triggered. It is set by default to 128.
- *gNvEndOfTableId_c* : Macro used to define the ID of the end-of-table entry. It is set to 0xFFFEU by default. No valid entry should use this ID.
- *gNvTableMarker_c* : Macro used to define the table marker value. The table marker is used to indicate the start and the end of the flash copy of the NV table. It is set to 0x4254U by default.
- *gNvFlashTableVersion_c* : Macro used to define the flash table version. It is used to determine if the NVM table was updated. It is set to 1 by default. The application should modify this every time the NVM table is updated and the data from NVM is still required.
- *gNvTableKeptInRam_d* : Set gNvTableKeptInRam_d to FALSE, if the NVM table is stored in FLASH memory (default). If the NVM table is stored in RAM memory, set the macro to TRUE.
- *gNvVerifyReadBackAfterProgram_d* : set by default force verification of NVM programming operations. Is forced implicitly when gNvSalvageFromEccFault_d is defined.
- *gNvSalvageFromEccFault_d* : use safe flash API to read from flash, and provide corrective action when ECC fault is met.

OtaSupport: Over-the-Air Programming Support

Overview This module includes APIs for the over-the-air image upgrade process. A Server device receives an image over the serial interface from a PC or other device thorough FSCI commands. If the Server has an image storage, the image is saved locally. If not, the image is requested chunk by chunk: With image storage

- OTA_RegisterToFsci()
- OTA_InitExternalMemory()
- OTA_WriteExternalMemory()
- …
- OTA_WriteExternalMemory()

Without image storage:

- OTA_RegisterToFsci()
- OTA_QueryImageReq()
- OTA_ImageChunkReq()
- …
- OTA_ImageChunkReq()

A Client device processes the received image by computing the CRC and filter unused data and stores the received image into a non-volatile storage. After the entire image has been transferred and verified, the Client device informs the Bootloader application that a new image is available, and that the MCU must be reset to start the upgrade process. See the following command sequence:

- OTA_StartImage()
- OTA_PushImageChunk() and OTA_CrcCompute ()
- ...
- OTA_PushImageChunk() and OTA_CrcCompute ()
- OTA_CommitImage()
- OTA_SetNewImageFlag()
- ResetMCU()

SecLib_RNG: Security library and random number generator

Random number generator

Overview The RNG module is part of the framework used for random number generation. It uses hardware RNG peripherals as entropy sources (TRNG, Secure Subsystem, …) to provide a true random number generator interface. A Pseudo-Random number generator (PRNG) implementation is available. The PRNG may depend of SecLib services (thus requiring a common mutex) to perform HMAC-SHA256, SHA256, AES-CTR, or alternateively a Lehmer Linear Congruential generator. A prerequisite for the PRNG to function with desired randomness is to be seeded using a proper source of entropy. If no hardware acceleration is present, the RNG may fallback to lesser quality ad-hoc source e.g if present SIM_UID registers, the UIDL is used as the initial seed for the random number generator.

Initialization The RNG module requires an initialization via a call to RNG_Init. The RNG initialization involves a call to RNG_SetSeed.

In the case of a dual core system consisting of a Host core and an NBU core, the Secure Subsystem is owned by the Host core. The Host core then has a direct access to its TRNG embedded in its secure subssystem. On the NBU code side, a request is emitted via RPMSG to the Host to provide a seed. On receipt of this request, the Host sets a 'reseed needed' flag (from the ISR context) If the core running the RNG service owns the TRNG entropy hardware (if any), it can get the seed directly form this hardware synchronously. In the case of an NBU that does not control the devices entropy source, that is owned by the Host, it request a seed from the Host processor via RPMSG exchange. On receipt of this request the Host sets a flag notifying of this request from the RPMSG ISR context. From the Idle thread, this flag is polled via the RNG_IsReseedNeeded API. If set the seed is regenerated and forwarded to the NBU via RPMSG.

RNG_ReInit API is to be used at wake up time in the context of LowPower. RNG_DeInit is used for unit tests and coverage purposes but has no useful role in a real application.

Seed handling RNG_SetSeed: RNG_SetExternalSeed may be used to inject application entropy to RNG context seed using a supplied array of bytes. RNG_IsReseedNeeded used from task in Host core to check whether seed must be sent to NBU core.

RNG_GetTrueRandomNumber is the API used to generate a Random 32 bit number from a HW source of entropy. It is essential if only to seed the pseudo random number generator.

RNG_GetPseudoRandomData is used to generate arrays of random bytes.

Security Library

Overview The framework provides support for cryptography in the security module. It supports both software and hardware encryption. Depending on the device, the hardware encryption uses either the S200, MMCAU, LTC, or CAU3 module instruction set or dedicated AES and SHA hardware blocks.

Software implementation is provided in a library format.

Support for security algorithms

	SW Seclib : SecLib.c	EdgeLock SecLib_sss.c	Seclib_e	Mbedtls SecLib_mbec	nccl (part of SecLib.c)	Usage example
AES_128	SecLib_aes.c	x		x		
AES_128_ECB		x		x		
AES_128_CBC	x	x		x		
AES_128_CTR encryption	x	x				
AES_128_OFB Encryption	x					
AES_128_CMAC	x	x		x		BLE connection, ieee 15.4
AES_128_EAX	x					
AES_128_CCM	x	x		x		BLE, ieee 15.4
SHA1	SecLib_sha.c	x		x		
SHA256	x	x		x		
HMAC_SHA256	x	x		x		PRNG, Digest for Matter
ECDH_P256 shared secret generation	x (by 15 incremental steps) -> SecLib_ecdh.c	x with MACRO SecLibECDHUseSSS	x	x	x	BLE pairing,
EC_P256 key pair generation	x	x	x	x	x	
EC_P256 public key generation from private key			x	x	x	Matter (ECDSA)
ECDSA_P256 hash and msg signature generation / verification		only if owner of the key pair		x	x	Matter
SPAKE2+ P256 arithmetics				x	x	Matter

BLE advanced secure mode

New elements in existing structures: computeDhKeyParam_t::keepInternalBlob - boolean telling if the shared blob is kept in this structure(in .outpoint) after ECDH_P256_ComputeDhKey() or ECDH_P256_ComputeDhKeySeg() call.

New arguments in existing functions: ECDH_P256_ComputeDhKey keepBlobDhKey - boolean telling ECDH_P256_ComputeDhKey() or ECDH_P256_ComputeDhKeySeg() to keep the shared object after computation for later use (it is required by the SecLib_GenerateBluetoothF5KeysSecure).

New macros: gSecLibSssUseEncryptedKeys_d - Enable or disable S200 blobs SecLib support. 0 - the Bluetooth Keys are available in plaintext, 1 - the Bluetooth Keys are not available in plaintext, but in secured blobs. Default is disabled.

New functions:

LE Secure connections pairing:

void ECDH_P256_FreeDhKeyDataSecure This is a function used to free the shared object stored in computeDhKeyParam_t. When user calls ECDH_P256_ComputeDhKeySeg() with keepBlobDhKey set to 1, it should also call **ECDH_P256_FreeDhKeyDataSecure** .

SecLib_GenerateBluetoothF5Keys This function is extracted from the Bluetooth LE Host Stack implementation. This corresponds to the legacy implementation without key blobs.

SecLib_GenerateBluetoothF5KeysSecure Similar to **SecLib_GenerateBluetoothF5Keys** this function is modified to work with key blobs, the reason is to not use SSS inside the Bluetooth LE Host Stack.

SecLib_DeriveBluetoothSKD This is a helper function used by the Bluetooth LE Host Stack in the pairing procedure, when receiving the vendor HCI command specifying that the ESK needs to be provided to LL.

ELKE_BLE_SM_F5_DeriveKeys This is a private function, helper for **SecLib_GenerateBluetoothF5KeysSecure**. It was provided by the STEC team.

Privacy:

SecLib_ObfuscateKeySecure This is a function used by the Bluetooth LE Host Stack to obfuscate the IRK before setting it to Bluetooth LE Controller or before saving it to NVM

SecLib_DeobfuscateKeySecure This is a function used by the Bluetooth LE Host Stack to extract the plaintext IRK key from the saved NVM blob.

SecLib_VerifyBluetoothAh This function is extracted from the legacy Bluetooth LE Host Stack implementation using plaintext keys.

SecLib_VerifyBluetoothAhSecure Similar to **SecLib_VerifyBluetoothAh** with modification to work with S200 key blob.

SecLib_GenerateSymmetricKey This is a function used by the application to generate the local IRK and local CSRK.

SecLib_GenerateBluetoothEIRKBlobSecure This is a function used by the application to generate the EIRK needed by Bluetooth LE Controller from the IRK blob.

A2B feature

ECDH_P256_ComputeA2BKey This function is used to compute the EdgeLock to EdgeLock key. pInPeerPublicKey points to the peer public key, pOutE2EKey is the pointer to where the E2E key object will be stored, this will be freed by the application when it is no longer required by calling ECDH_P256_FreeE2EKeyData().

ECDH_P256_FreeE2EKeyData This function is used to free the key object given as a parameter. It is used by the application to free the E2E key when is no longer needed.

SecLib_ExportA2BBlobSecure This function is used to import an ELKE blob or plain text symmetric key in s200 and export an E2E key blob. The input type is identified by the keyType parameter.

SecLib_ImportA2BBlobSecure This function is used to import an E2E key blob in s200 and export an ELKE blob or plain text symmetric key. The output type is identified by the keyType parameter.

LE Secure connections Pairing flow and SecLib usage:

1. Each device needs to generate locally the public+private keypair. This is done using **ECDH_P256_GenerateKeys**.
2. Devices exchange their public keys.
3. Upon receiving the peer device's public key, local device is computing DH key using **ECDH_P256_ComputeDhKey**.
4. Each device sends DHKeyCheck packet
5. Upon receiving DhKeyCheck each device computes LTK blob using **SecLib_GenerateBluetoothF5Keys**
6. After computing the each device sends HCI_LeStartEnc (on initiator), HCI_Le_Provide_Long_Term_Key (on responder)
7. Bluetooth LE Controller sends back SKD report custom event
8. Bluetooth LE Host Stack computes ESKD based on LTK blob using **SecLib_DeriveBluetoothSKD** and sends it to Bluetooth LE Controller
9. Bluetooth LE Controller encrypts the link

IRK flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received:

- the local IRK is generated using **SecLib_GenerateSymmetricKey**
- the local EIRK is generated using **SecLib_GenerateBluetoothEIRKBlobSecure**
- local CSRK is generated using **SecLib_GenerateSymmetricKey**

2. During legacy pairing when receiving bonding keys, IRK is obfuscated using **SecLib_ObfuscateKeySecure** and stored
3. When app wants to set the OOB keys using Gap_SaveKeys the IRK is obfuscated using **SecLib_ObfuscateKeySecure**
4. When application calls API Gap_VerifyPrivateResolvableAddress IRK is obfuscated using **SecLib_ObfuscateKeySecure** and verified using **SecLib_VerifyBluetoothAhSecure**
5. When a new connection is received in Host with RPA address not resolved by the Bluetooth LE Controller, the Host tries to resolve it by obfuscating it using **SecLib_ObfuscateKeySecure** and verifying it using **SecLib_VerifyBluetoothAhSecure**
6. When adding a peer in Bluetooth LE Controller resolving list, the peer's IRK is obfuscated using **SecLib_ObfuscateKeySecure** before setting it using **HCI_Le_Add_Device_To_Resolving_List**.
7. When an IRK plaintext is requested by the application using Gap_LoadKeys it is obtained using **SecLib_DeobfuscateKeySecure**
8. When legacy pairing completes and LTK needs to be send in the pairing complete event (gConnEvtPairingComplete_c) the **SecLib_DeobfuscateKey** is used to extract the plaintext.

A2B flow and SecLib usage:

1. At startup, when gInitializationComplete_c event is received, the application will call **ECDH_P256_GenerateKeys** to generate the public/private key pair required for the E2E key derivation and send the public key to the peer device.
2. When the public key is received from the peer device, the application will call **ECDH_P256_ComputeA2BKeySecure** to generate the EdgeLock to EdgeLock key.
3. The application will obtain an E2E IRK blob by calling **SecLib_ExportA2BBlobSecure** with key type gSecElkeBlob_c. The obtained blob is sent to the peer anchor. The peer anchor will call **SecLib_ImportA2BBlob** with keyType gSecElkeBlob_c and save the resulting ELKE blob in NVM, for Digital Key both anchors must have the same IRK.
4. After pairing, in order to send the LTK and IRK contained in the bonding data securely, the application will call **SecLib_ExportA2BBlobSecure** with keyType gSecLtkElkeBlob_c for the LTK, and **SecLib_ExportA2BBlobSecure** with keyType gSecPlainText_c for the IRK. The E2E blobs obtained are sent along with the rest of the bonding data to the peer anchor device.
5. After the bonding data is trasfered the E2E key is no longer needed and **ECDH_P256_FreeE2EKeyData** is called with the key object obtained at step 2 when **ECDH_P256_ComputeA2BKeySecure** was called.

Sensors

Overview The Sensors module provides an API to communicate with the ADC. Two values can be obtained by this module :

- Temperature value

- Battery level

The temperature is given in tenths of degrees Celsius and the battery in percentage.

This module is multi-caller, the ADC is protected by a mutex on the resource and by prevententing lowpower (only WFI) during its processing. Platform specific code can be find in fwk_platform_sensors.c/h.

Constant macro definitions Name :

```
#define VALUE_NOT_AVAILABLE_8 0xFFu
#define VALUE_NOT_AVAILABLE_32 0xFFFFFFFFu
```

Description :

Defines the error value that can be compared to the value obtain on the ADC.

SFC : Smart Frequency Calibration

Overview The Smart Frequency Calibration module provides operations and calibration for the FRO32K source clock. This module is split between main core and Radio core:

- fwk_rf_sfc.[ch]: RF_SFC module on Radio core that provides Main FRO32K measurement/calibration and state machine in synchornization with Radio domain activities. See details below.
- fwk_sfc.h: SFC module on host core that provides type definition for usage with fwk_platform_ics.[ch] with PLATFORM_FwkSrvSetRfSfcConfig() API and fwk_platform_ble.c for received callback from the NBU core

Host SFC Module

Algorithm parametrization This module provides ability to configure the RF_SFC module by sending message to Radio core through fwk_platform_ics.c PLATFORM_FwkSrvSetRfSfcConfig():

- Filter size
- Maximum ppm threshold
- Maximum calibration interval
- Number of sample in filter to swicth from convergence to monitor mode

Ppm target The ppm target is the deviation from the target clock accepted by the algorithm. When the deviation is larger than the ppm target. The algorithm will update the trimming value and reset the filter. The ppm target cannot be more aggressive RF_SFC_MAXIMAL_PPM_TARGET in order to avoid having to update trimming value at each measurement.

Filter size Filter size must be included between RF_SFC_MINIMAL_FILTER_SIZE and RF_SFC_MAXIMAL_FILTER_SIZE. See *Filtering and Frequency estimation* section for more details on the parameter.

Maximum calibration interval In monitor mode, new measurement are triggered by low-power entry/exit. If the NBU core has a lot of radio activity it could never enter lowpower. The maximum calibration interval is here to ensure a measurement is done regularly. When executing idle the SFC module checks when the last measurement has been done, if it has been too long, it reset the filter and forces a new measurement

Trig sample number The trig sample number is the number of samples needed by the algorithm in its filter to switch from convergence to monitor mode. Having more than one sample in convergence mode allows to confirm the trimming value that we have set.

SFC debug information On the other way, the RF_SFC from Radio core sends back notifications to SFC module on main core using RX callback PLATFORM_RegisterFroNotificationCallback() from fwk_platform_ics.h and such information:

- last measured frequency
- average ppm from 32768Khz frequency
- last ppm measured from 32768Khz frequency
- FRO trimming value

RF_SFC module The RF_SFC module provides the functionality to calibrate the FRO32K source clock during Initialization and radio activity.

The RF_SFC is mostly used on XTAL32K less solution when no 32Khz crystal is soldered on the board. It allows to calibrate the FRO32K source clock to the desired frequency to keep Radio time base within the allowed tolerance given by the connectivity standards. However, even on a XTAL32K solution, the RF_SFC is also used during Initialization until the XTAL32K is up and running in the system. The system firstly runs on the FRO32K clock source then switch to the XTAL32K clock source when it is ready with enough accuracy. This allows to save significant boot time as the FRO32K start up (including calibration) is much faster compared to XTAL32K .

This module will handle:

- FRO32K clock frequency measurement against 32Mhz crystal. It schedules appropriately the start of the measurement and gets the result when completed,
- Filter and estimate the 32Khz frequency value and error by averaging from the last measurements,
- FRO32K calibration in order to update the trimming value to reduce the frequency error on the clock.

The targeted frequency offset shall be within 200ppm. The RF_SFC will handle two modes of operation:

- Convergence mode: when frequency estimation is above 200pm,
- Monitor mode: when frequency estimation is below 200pm.

The RF_SFC module works in active and all low power modes on NBU domain, or on host application domain except power down mode. Power down mode on host application domain is not supported with the FRO32K configuration as clock source.

Feature enablement Enabling the FRO32K is done by calling the PLATFORM_InitFro32K() function during application initialization in hardware_init.c file, in BOARD_InitHardware() function. If FRO32K is not enabled, Oscillator XTAL32K shall be called instead by calling PLATFORM_InitOsc32K() function. The call to PLATFORM_InitFro32K() from BOARD_InitHardware() can be done by setting the Compilation flag gBoardUseFro32k_d to 1 in hardware_init.c or any header files included from this file.

```
#define gBoardUseFro32k_d       1
```

Detailed description

Frequency measurements When NBU low power is enabled, the frequency measurements are triggered on Low power wake-up by HW signal. The SFC process called from Idle task will check regularly the completion of the frequency measurement. When the measurement is done, it goes to filtering and frequency estimation process. The frequency measurement duration depends on monitor mode or convergence mode: In convergence mode, the frequency measurement duration is 0.5ms while it is 2ms in monitor mode. In monitor mode, the duration value remains less than the minimal radio activity duration so it does not impact the low power consumption in monitoring mode.

Filtering and Frequency estimation The FRO32KHz frequency measurement values are noisy because of thermal noise on the FRO32K itself. Also, the frequency measurement can introduce some error. In monitoring mode, it is required to filter the measurements by applying an exponential filter. new_estimation = (new_measurement + ((1 « n) -1) * last_estimation) » n

Default value for n is 7 (meaning 128 samples in the averaging window).

Frequency calibration When the frequency estimation gets higher than the targeted 200ppm target, the RF_SFC updates the trimming value for one positive or negative increment. For this purpose, it requires to:

- wake up the host application domain and keep the domain active,
- update the trim register of the FRO32K , this register is used to trim the capacitance value of the FRO32K,
- re-allow the host application domain to enter low power.

A slight power impact is expected during a calibration update due to host domain wake-up.

Operational modes When the low power mode is enabled on NBU power domain, RF SFC handles two modes of operation: convergence and monitor modes. However, when low power is disabled on NBU power domain, only convergence mode is supported.

Convergence mode Convergence mode is used when the estimated FRO32K frequency is above 200ppm or when the filter has been reset. Typically this occurs :

- During Power ON reset or other reset when NBU is switched OFF
- When temperature varies and FRO32K frequency deviates outside 200ppm threshold target
- When no calibration has been done during some time as we discard old values that could influence the algorithm

The convergence mode process typically starts with a FRO32K trim register update, performs a frequency measurement and the FRO32K trim register is updated until the measured frequency gets below 200ppm. These operations are repeated in a loop until the estimated frequency value gets below 200ppm. When below 200ppm during multiple measurements, the RC SFC switches to Monitoring mode. The convergence mode is only a transition mode to monitoring mode. In convergence mode, the NBU power domain does not go to low power. The convergence mode time duration depends on the initial frequency error of the FR032K. Default frequency measurement duration is 0.5ms so 20 measurements (given as example only) will require less than 10 ms to converge.

Monitoring mode Monitoring mode is used when the estimated FRO32K frequency is below 200ppm. In this mode, the measurement is triggered on NBU domain wake up from low power mode using an internal hardware signal. The exponential filter is applied to compute the frequency estimation. If the frequency estimation value is still within 200ppm, the NBU power domain is allowed to go to low power. If the estimated value gets above the 200ppm threshold, the RF SFC switch back to convergence mode. The trim register is updated by one increment (positive or negative) and because the frequency has been adjusted and changed, the estimated filtered frequency is reset to discard all previous measurements. Going back to convergence mode typically happens during a temperature gradient. If the temperature is constant, it is not expected to have the estimated value to go beyond 200ppm so no calibration should be required.

Initialization and configuration During initialization, the RF SFC module will block the Radio Software until monitoring mode is reached. This is to prevent the radio from running with an inaccurate time base due to an important 32k clock frequency error.

Initialization and configuration is done by the NBU core. The configuration parameters can set up:

- The 200ppm target threshold. This value shall be 200ppm or higher.
- The filtering number n (see section above), It shall be between 0 and 8. Default is 7 which is similar to an averaging filter of 128 samples. A higher value will be more robust against noise. A lower value will track temperature variation more faster.

In order to prevent the host application domain from going into power down mode (power down mode not supported with FRO32K as clock source), the fwkSrvLowPowerConstraintCallbacks functions structure is registered to the Framework service on host application core from fwk_platform_lowpower.c file, PLATFORM_LowPowerInit() function. The NBU code applies a low power Deep Sleep constraint to the application core. This constraint is released when the NBU firmware has no activity to do and re-applied when a new activity starts.

Lowpower impact

Power impact during active mode: In monitoring mode (this should be 99.9% of the time if temperature does not vary), the FRO32KHz frequency measurements are performed during a Radio activity so it does not increase the active current as the sources clocks are already active. Also, it does not increase the active time as the measurement takes less time than an advertising event or connection event so no impact on power consumption.

The main power impact will be in convergence mode. In this case, measurements/calibrations are done in loop until the monitoring mode is reached (frequency error less than 200ppm). This could happen:

- During power ON reset,
- When temperature varies: The frequency will deviate from 32768Hz and FRO32K trimming register correction will need to be updated for that,
- When no measurement has been done during some time as we cannot predict if the FRO has drifted, so we discard older values and start convergence mode.

When FRO32K frequency needs to be adjusted, the NBU core will wake-up the main power domain and will update the FRO32K trimming register.

Power impact during low power mode: The power consumption in low power mode will increase slightly due to running FRO32K compared to XTAL32K. The power consumption of FRO32K typically consumes 350nA while it is only 100nA with XTAL32K. Refer to the product datasheet for the exact numbers.

1.6.2 Wi-Fi

This manual provides developer reference documentation for Wi-Fi driver and Wi-Fi Connection Manager. Refer to the source code for additional information.

Wi-Fi API Guide for RW61x

Main Page

Introduction NXP wireless SoCs require a combination of firmware binary image streamed into the radio subsystem, and driver source code compiled onto the application MCU. The radio driver source code provides APIs that enable a developer to send and receive packets over the radio interfaces by communicating with the firmware images that are streamed into the radio subsystems on start-up.

Developer Documentation This manual provides developer reference documentation for Wi-Fi driver and Wi-Fi Connection Manager. Refer to the source code for additional information.

Note

> The File Documentation provides documentation for all the APIs that are available in Wi-Fi driver and connection manager.

Abbreviations and acronyms

File Index

File List Here is a list of all documented files with brief descriptions:

wlan.h (This file provides Wi-Fi APIs for the application)

Data Structure Documentation

ipv4_config Struct Reference

Data Fields enum address_types addr_type

unsigned address

unsigned gw

unsigned netmask

unsigned dns1

unsigned dns2

Detailed Description This data structure represents an IPv4 address

Field Documentation

enum address_types ipv4_config::addr_type

Set to ADDR_TYPE_DHCP to use DHCP to obtain the IP address or set to ADDR_TYPE_STATIC to use a static IP. In case of static IP address ip, gw, netmask and dns members should be specified. When using DHCP, the ip, gw, netmask and dns are overwritten by the values obtained from the DHCP server. They should be zeroed out if not used.

unsigned ipv4_config::address

The system's IP address in network order.

unsigned ipv4_config::gw

The system's default gateway in network order.

unsigned ipv4_config::netmask

The system's subnet mask in network order.

unsigned ipv4_config::dns1

The system's primary dns server in network order.

unsigned ipv4_config::dns2

The system's secondary dns server in network order.

The documentation for this struct was generated from the following file: wlan.h

ipv6_config Struct Reference

Data Fields unsigned address [4]

unsigned char addr_type

uint8_t addr_state

Detailed Description This data structure represents an IPv6 address

Field Documentation

unsigned ipv6_config::address[4]

The system's IPv6 address in network order.

unsigned char ipv6_config::addr_type

The address type: linklocal, site-local or global.

uint8_t ipv6_config::addr_state

The state of IPv6 address (Tentative, Preferred, etc.).

The documentation for this struct was generated from the following file: wlan.h

rx_pkt_he_rate_info Struct Reference

Data Fields t_u32 hemcs_rxcnt [12]

t_u32 hestbcrate_rxcnt [12]

Detailed Description Sum of RX packets for HE (802.11ax high efficiency) rate.

Field Documentation

t_u32 rx_pkt_he_rate_info::hemcs_rxcnt[12]

Sum of RX packets for HE rate. The array index represents MSC0~MCS11, the following array indexes have the same effect.

t_u32 rx_pkt_he_rate_info::hestbcrate_rxcnt[12]

Sum of RX STBC (space time block code) packets for HE rate.

The documentation for this struct was generated from the following file: wlan.h

rx_pkt_ht_rate_info Struct Reference

Data Fields t_u32 htmcs_rxcnt [16]

t_u32 htsgi_rxcnt [16]

t_u32 htstbcrate_rxcnt [16]

Detailed Description Sum of RX packets for HT (802.11n high throughput) rate.

Field Documentation

t_u32 rx_pkt_ht_rate_info::htmcs_rxcnt[16]

Sum of RX packets for HT rate. The array index represents MSC0~MCS15, the following array indexes have the same effect.

t_u32 rx_pkt_ht_rate_info::htsgi_rxcnt[16]

Sum of TX short GI (guard interval) packets for HT rate.

t_u32 rx_pkt_ht_rate_info::htstbcrate_rxcnt[16]

Sum of TX STBC (space time block code) packets for HT rate.

The documentation for this struct was generated from the following file: wlan.h

rx_pkt_rate_info Struct Reference

Data Fields t_u32 nss_rxcnt [2]

t_u32 nsts_rxcnt

t_u32 bandwidth_rxcnt [3]

t_u32 preamble_rxcnt [6]

t_u32 ldpc_txbfcnt [2]

t_s32 rssi_value [2]

t_s32 rssi_chain0 [4]

t_s32 rssi_chain1 [4]

Detailed Description Sum of RX packets.

Field Documentation

t_u32 rx_pkt_rate_info::nss_rxcnt[2]

Sum of RX NSS (N*N MIMO spatial stream) packets. nss_txcnt[0] is for NSS 1, nss_txcnt[1] is for NSS 2.

t_u32 rx_pkt_rate_info::nsts_rxcnt

Sum of received packets for all STBC rates.

t_u32 rx_pkt_rate_info::bandwidth_rxcnt[3]

Sum of received packets for three bandwidth types. bandwidth_rxcnt[0] is for 20MHz, bandwidth_rxcnt[1] is for 40MHz, bandwidth_rxcnt[2] is for 80MHz.

t_u32 rx_pkt_rate_info::preamble_rxcnt[6]

Sum of received packets for four preamble format types. preamble_txcnt[0] is for preamble format 0, preamble_txcnt[1] is for preamble format 1, preamble_txcnt[2] is for preamble format 2, preamble_txcnt[3] is for preamble format 3, preamble_txcnt[4] and preamble_txcnt[5] are as reserved.

t_u32 rx_pkt_rate_info::ldpc_txbfcnt[2]

Sum of packets for TX LDPC packets.

t_s32 rx_pkt_rate_info::rssi_value[2]

Average RSSI

t_s32 rx_pkt_rate_info::rssi_chain0[4]

RSSI value of path A

t_s32 rx_pkt_rate_info::rssi_chain1[4]

RSSI value of path B

The documentation for this struct was generated from the following file: wlan.h

rx_pkt_vht_rate_info Struct Reference

Data Fields t_u32 vhtmcs_rxcnt [10]

t_u32 vhtsgi_rxcnt [10]

t_u32 vhtstbcrate_rxcnt [10]

Detailed Description Sum of RX packets for VHT (802.11ac very high throughput) rate.

Field Documentation

t_u32 rx_pkt_vht_rate_info::vhtmcs_rxcnt[10]

Sum of RX packets for VHT rate. The array index represents MSC0~MCS9, the following array indexes have the same effect.

t_u32 rx_pkt_vht_rate_info::vhtsgi_rxcnt[10]

Sum of RX short GI (guard interval) packets for VHT rate.

t_u32 rx_pkt_vht_rate_info::vhtstbcrate_rxcnt[10]

Sum of RX STBC (space time block code) packets for VHT rate.

The documentation for this struct was generated from the following file: wlan.h

tx_ampdu_prot_mode_para Struct Reference

Data Fields int mode

Detailed Description Set protection mode for the transmit AMPDU packet

Field Documentation

int tx_ampdu_prot_mode_para::mode

mode, 0: set RTS/CTS mode, 1: set CTS to self mode, 2: disable protection mode, 3: set dynamic RTS/CTS mode.

The documentation for this struct was generated from the following file: wlan.h

tx_pkt_he_rate_info Struct Reference

Data Fields t_u32 hemcs_txcnt [12]

t_u32 hestbcrate_txcnt [12]

Detailed Description Sum of TX packets for HE (802.11ax high efficiency) rate.

Field Documentation

t_u32 tx_pkt_he_rate_info::hemcs_txcnt[12]

Sum of TX packets for HE rate. The array index represents MSC0~MCS11, the following array indexes have the same effect.

t_u32 tx_pkt_he_rate_info::hestbcrate_txcnt[12]

Sum of TX STBC (space time block code) packets for HE rate.

The documentation for this struct was generated from the following file: wlan.h

tx_pkt_ht_rate_info Struct Reference

Data Fields t_u32 htmcs_txcnt [16]

t_u32 htsgi_txcnt [16]

t_u32 htstbcrate_txcnt [16]

Detailed Description Sum of TX packets for HT (802.11n high throughput) rate.

Field Documentation

t_u32 tx_pkt_ht_rate_info::htmcs_txcnt[16]

Sum of TX packets for HT rate. The array index represents MSC0~MCS15, the following array indexes have the same effect.

t_u32 tx_pkt_ht_rate_info::htsgi_txcnt[16]

Sum of TX short GI (guard interval) packets for HT rate.

t_u32 tx_pkt_ht_rate_info::htstbcrate_txcnt[16]

Sum of TX STBC (space time block code) packets for HT rate.

The documentation for this struct was generated from the following file: wlan.h

tx_pkt_rate_info Struct Reference

Data Fields t_u32 nss_txcnt [2]

t_u32 bandwidth_txcnt [3]

t_u32 preamble_txcnt [4]

t_u32 ldpc_txcnt

t_u32 rts_txcnt

t_s32 ack_RSSI

Detailed Description Sum of TX packets.

Field Documentation

t_u32 tx_pkt_rate_info::nss_txcnt[2]

Sum of TX NSS (N*N MIMO spatial stream) packets. nss_txcnt[0] is for NSS 1, nss_txcnt[1] is for NSS 2.

t_u32 tx_pkt_rate_info::bandwidth_txcnt[3]

Sum of TX packets for three bandwidths. bandwidth_txcnt[0] is for 20MHz, bandwidth_txcnt[1] is for 40MHz, bandwidth_txcnt[2] is for 80MHz.

t_u32 tx_pkt_rate_info::preamble_txcnt[4]

Sum of RX packets for four preamble format types. preamble_txcnt[0] is for preamble format 0, preamble_txcnt[1] is for preamble format 1, preamble_txcnt[2] is for preamble format 2, preamble_txcnt[3] is for preamble format 3,

t_u32 tx_pkt_rate_info::ldpc_txcnt

Sum of TX LDPC (low density parity check) packets.

t_u32 tx_pkt_rate_info::rts_txcnt

Sum of TX RTS (request to send) packets

t_s32 tx_pkt_rate_info::ack_RSSI

RSSI of ACK packet

The documentation for this struct was generated from the following file: wlan.h

tx_pkt_vht_rate_info Struct Reference

Data Fields t_u32 vhtmcs_txcnt [10]

t_u32 vhtsgi_txcnt [10]

t_u32 vhtstbcrate_txcnt [10]

Detailed Description Sum of TX packets for VHT (802.11ac very high throughput) rate.

Field Documentation

t_u32 tx_pkt_vht_rate_info::vhtmcs_txcnt[10]

Sum of TX packets for VHT rate. The array index represents MSC0~MCS9, the following array indexes have the same effect.

t_u32 tx_pkt_vht_rate_info::vhtsgi_txcnt[10]

Sum of TX short GI packets for HT mode.

t_u32 tx_pkt_vht_rate_info::vhtstbcrate_txcnt[10]

Sum of TX STBC (space time block code) packets for VHT mode.

The documentation for this struct was generated from the following file: wlan.h

wifi_scan_params_t Struct Reference

Data Fields uint8_t * bssid

char * ssid

int channel [MAX_CHANNEL_LIST]

IEEEtypes_Bss_t bss_type

int scan_duration

int split_scan_delay

Detailed Description This structure is used to configure Wi-Fi scan parameters

Field Documentation

uint8_t* wifi_scan_params_t::bssid

BSSID (basic service set ID)

char* wifi_scan_params_t::ssid

SSID (service set ID)

int wifi_scan_params_t::channel[MAX_CHANNEL_LIST]

Channel list

IEEEtypes_Bss_t wifi_scan_params_t::bss_type

BSS (basic service set) type. 1: Infrastructure BSS, 2: Indenpent BSS.

int wifi_scan_params_t::scan_duration

Time for scan duration

int wifi_scan_params_t::split_scan_delay

split scan delay

The documentation for this struct was generated from the following file: wlan.h

wlan_cipher Struct Reference

Data Fields uint16_t none: 1

uint16_t wep40: 1

uint16_t wep104: 1

uint16_t tkip: 1

uint16_t ccmp: 1

uint16_t aes_128_cmac: 1

uint16_t gcmp: 1

uint16_t sms4: 1

uint16_t gcmp_256: 1

uint16_t ccmp_256: 1

uint16_t rsvd: 1

uint16_t bip_gmac_128: 1

uint16_t bip_gmac_256: 1

uint16_t bip_cmac_256: 1

uint16_t gtk_not_used: 1

uint16_t rsvd2: 2

Detailed Description Wi-Fi cipher structure

Field Documentation

uint16_t wlan_cipher::none

1 bit value can be set for none

uint16_t wlan_cipher::wep40

1 bit value can be set for wep40

uint16_t wlan_cipher::wep104

1 bit value can be set for wep104

uint16_t wlan_cipher::tkip

1 bit value can be set for tkip

uint16_t wlan_cipher::ccmp

1 bit value can be set for ccmp

uint16_t wlan_cipher::aes_128_cmac

1 bit value can be set for aes 128 cmac

uint16_t wlan_cipher::gcmp

1 bit value can be set for gcmp

uint16_t wlan_cipher::sms4

1 bit value can be set for sms4

uint16_t wlan_cipher::gcmp_256

1 bit value can be set for gcmp 256

uint16_t wlan_cipher::ccmp_256

1 bit value can be set for ccmp 256

uint16_t wlan_cipher::rsvd

1 bit is reserved

uint16_t wlan_cipher::bip_gmac_128

1 bit value can be set for bip gmac 128

uint16_t wlan_cipher::bip_gmac_256

1 bit value can be set for bip gmac 256

uint16_t wlan_cipher::bip_cmac_256

1 bit value can be set for bip cmac 256

uint16_t wlan_cipher::gtk_not_used

1 bit value can be set for gtk not used

uint16_t wlan_cipher::rsvd2

4 bits are reserved

The documentation for this struct was generated from the following file: wlan.h

wlan_ieeeps_config Struct Reference

Data Fields t_u32 ps_null_interval

t_u32 multiple_dtim_interval

t_u32 listen_interval

t_u32 adhoc_awake_period

t_u32 bcn_miss_timeout

t_s32 delay_to_ps

t_u32 ps_mode

Detailed Description This structure is for IEEE PS (power save) configuration

Field Documentation

t_u32 wlan_ieeeps_config::ps_null_interval

The interval that STA sends null packet

t_u32 wlan_ieeeps_config::multiple_dtim_interval

The count of listen interval

t_u32 wlan_ieeeps_config::listen_interval

Periodic interval that STA listens to AP beacons

t_u32 wlan_ieeeps_config::adhoc_awake_period

Periodic awake period for adhoc networks

t_u32 wlan_ieeeps_config::bcn_miss_timeout

Beacon miss timeout in milliseconds

t_s32 wlan_ieeeps_config::delay_to_ps

The delay of enabling IEEE-PS in milliseconds

t_u32 wlan_ieeeps_config::ps_mode

PS mode, 1: PS-auto mode, 2: PS-poll mode, 3: PS-null mode.

The documentation for this struct was generated from the following file: wlan.h

wlan_ip_config Struct Reference

Data Fields struct ipv6_config ipv6 [CONFIG_MAX_IPV6_ADDRESSES]

size_t ipv6_count

struct ipv4_config ipv4

Detailed Description Network IP configuration.

This data structure represents the network IP configuration for IPv4 as well as IPv6 addresses

Field Documentation

struct ipv6_config wlan_ip_config::ipv6[CONFIG_MAX_IPV6_ADDRESSES]

The network IPv6 address configuration that should be associated with this interface.

size_t wlan_ip_config::ipv6_count

The network IPv6 valid addresses count

struct ipv4_config wlan_ip_config::ipv4

The network IPv4 address configuration that should be associated with this interface.

The documentation for this struct was generated from the following file: wlan.h

wlan_network Struct Reference

Data Fields int id

int wps_network

char name [WLAN_NETWORK_NAME_MAX_LENGTH+1]

char ssid [IEEEtypes_SSID_SIZE+1]

char bssid [IEEEtypes_ADDRESS_SIZE]

unsigned int channel

uint8_t sec_channel_offset

uint16_t acs_band

int rssi

short rssi_threshold

unsigned short ht_capab

unsigned int vht_capab

unsigned char vht_oper_chwidth

unsigned char he_oper_chwidth

enum wlan_bss_type type

enum wlan_bss_role role

struct wlan_network_security security

struct wlan_ip_config ip

unsigned ssid_specific: 1

unsigned trans_ssid_specific: 1

unsigned bssid_specific: 1

unsigned channel_specific: 1

unsigned security_specific: 1

unsigned dot11n: 1

unsigned dot11ac: 1

unsigned dot11ax: 1

uint16_t mdid

unsigned ft_1x: 1

unsigned ft_psk: 1

unsigned ft_sae: 1

unsigned int owe_trans_mode

char trans_ssid [IEEEtypes_SSID_SIZE+1]

unsigned int trans_ssid_len

uint16_t beacon_period

uint8_t dtim_period

uint8_t wlan_capa

uint8_t btm_mode

bool bss_transition_supported

bool neighbor_report_supported

bool **twt_capab**

Detailed Description Wi-Fi network profile

This data structure represents a Wi-Fi network profile. It consists of an arbitrary name, Wi-Fi configuration, and IP address configuration.

Every network profile is associated with one of the two interfaces. The network profile can be used for the station interface (i.e. to connect to an Access Point) by setting the role field to WLAN_BSS_ROLE_STA. The network profile can be used for the uAP interface (i.e. to start a network of our own.) by setting the mode field to WLAN_BSS_ROLE_UAP.

If the mode field is WLAN_BSS_ROLE_STA, either of the SSID or BSSID fields are used to identify the network, while the other members like channel and security settings characterize the network.

If the mode field is WLAN_BSS_ROLE_UAP, the SSID, channel and security fields are used to define the network to be started.

In both the above cases, the address field is used to determine the type of address assignment to be used for this interface.

Field Documentation

int wlan_network::id

Identifier for network profile

int wlan_network::wps_network

WPS network flag.

char wlan_network::name[WLAN_NETWORK_NAME_MAX_LENGTH+1]

The name of this network profile. Each network profile that is added to the Wi-Fi connection manager should have a unique name.

char wlan_network::ssid[IEEEtypes_SSID_SIZE+1]

The network SSID, represented as a C string of up to 32 characters in length. If this profile is used in the uAP mode, this field is used as the SSID of the network. If this profile is used in the station mode, this field is used to identify the network. Set the first byte of the SSID to NULL (a 0-length string) to use only the BSSID to find the network.

char wlan_network::bssid[IEEEtypes_ADDRESS_SIZE]

The network BSSID, represented as a 6-byte array. If this profile is used in the uAP mode, this field is ignored. If this profile is used in the station mode, this field is used to identify the network. Set all 6 bytes to 0 to use any BSSID, in which case only the SSID is used to find the network.

unsigned int wlan_network::channel

The channel for this network.

If this profile is used in uAP mode, this field specifies the channel to start the uAP interface on. Set this to 0 for auto channel selection.

If this profile is used in the station mode, this constrains the channel on which the network to connect should be present. Set this to 0 to allow the network to be found on any channel.

uint8_t wlan_network::sec_channel_offset

The secondary channel offset

uint16_t wlan_network::acs_band

The ACS (auto channel selection) band if set channel to 0.

int wlan_network::rssi

RSSI (received signal strength indicator) value.

short wlan_network::rssi_threshold

Specify RSSI threshold (dBm) for scan

unsigned short wlan_network::ht_capab

HT capabilities info field within HT capabilities information element

unsigned int wlan_network::vht_capab

VHT capabilities info field within VHT capabilities information element

unsigned char wlan_network::vht_oper_chwidth

VHT bandwidth

unsigned char wlan_network::he_oper_chwidth

HE bandwidth

enum wlan_bss_type wlan_network::type

BSS type

enum wlan_bss_role wlan_network::role

The network Wi-Fi mode enum wlan_bss_role. Set this to specify what type of Wi-Fi network mode to use. This can either be WLAN_BSS_ROLE_STA for use in the station mode, or it can be WLAN_BSS_ROLE_UAP for use in the uAP mode.

struct wlan_network_security wlan_network::security

The network security configuration specified by struct wlan_network_security for the network.

struct wlan_ip_config wlan_network::ip

The network IP address configuration specified by struct wlan_ip_config that should be associated with this interface.

unsigned wlan_network::ssid_specific

If set to 1, the ssid field contains the specific SSID for this network. the Wi-Fi connection manager can only connect to networks with matching SSID matches. If set to 0, the ssid field contents are not used when deciding whether to connect to a network or not. The BSSID field is used instead and any network with matching BSSID matches is accepted.

This field can be set to 1 if the network is added with the SSID specified (not an empty string), otherwise it is set to 0.

unsigned wlan_network::trans_ssid_specific

If set to 1, the ssid field contains the transitional SSID for this network.

unsigned wlan_network::bssid_specific

If set to 1, the bssid field contains the specific BSSID for this network. The Wi-Fi connection manager cannot connect to any other network with the same SSID unless the BSSID matches. If set to 0, the Wi-Fi connection manager can connect to any network whose SSID matches.

This field set to 1 if the network is added with the BSSID specified (not set to all zeroes), otherwise it is set to 0.

unsigned wlan_network::channel_specific

If set to 1, the channel field contains the specific channel for this network. The Wi-Fi connection manager cannot look for this network on any other channel. If set to 0, the Wi-Fi connection manager can look for this network on any available channel.

This field is set to 1 if the network is added with the channel specified (not set to 0), otherwise it is set to 0.

unsigned wlan_network::security_specific

If set to 0, any security that matches is used. This field is internally set when the security type parameter above is set to WLAN_SECURITY_WILDCARD.

unsigned wlan_network::dot11n

The network supports 802.11N.

unsigned wlan_network::dot11ac

The network supports 802.11AC.

unsigned wlan_network::dot11ax

The network supports 802.11AX.

uint16_t wlan_network::mdid

Mobility Domain ID

unsigned wlan_network::ft_1x

The network uses FT 802.1x security

unsigned wlan_network::ft_psk

The network uses FT PSK security

unsigned wlan_network::ft_sae

The network uses FT SAE security

unsigned int wlan_network::owe_trans_mode

OWE (opportunistic wireless encryption) Transition mode

char wlan_network::trans_ssid[IEEEtypes_SSID_SIZE+1]

The network transitional SSID, represented as a C string of up to 32 characters in length.

unsigned int wlan_network::trans_ssid_len

Transitional SSID length

uint16_t wlan_network::beacon_period

Beacon period of associated BSS

uint8_t wlan_network::dtim_period

DTIM period of associated BSS

uint8_t wlan_network::wlan_capa

Wi-Fi capabilities of the uAP network 802.11n, 802.11ac or/and 802.11ax

uint8_t wlan_network::btm_mode

BTM mode

bool wlan_network::bss_transition_supported

BSS transition support

bool wlan_network::neighbor_report_supported

Neighbor report support

The documentation for this struct was generated from the following file: wlan.h

wlan_network_security Struct Reference

Data Fields enum wlan_security_type type

int key_mgmt

struct wlan_cipher mcstCipher

struct wlan_cipher ucstCipher

unsigned pkc: 1

int group_cipher

int pairwise_cipher

int group_mgmt_cipher

bool is_pmf_required

char psk [WLAN_PSK_MAX_LENGTH]

uint8_t psk_len

char password [WLAN_PASSWORD_MAX_LENGTH+1]

size_t password_len

char * sae_groups

uint8_t pwe_derivation

uint8_t transition_disable

char * owe_groups

char pmk [WLAN_PMK_LENGTH]

bool pmk_valid

int8_t mfpc

int8_t mfpr

unsigned wpa3_ent: 1

unsigned wpa3_sb: 1

unsigned wpa3_sb_192: 1

unsigned eap_ver: 1

unsigned peap_label: 1

uint8_t eap_crypto_binding

unsigned eap_result_ind: 1

unsigned char tls_cipher

char identity [IDENTITY_MAX_LENGTH]

char anonymous_identity [IDENTITY_MAX_LENGTH]

char eap_password [PASSWORD_MAX_LENGTH]

bool verify_peer_cert

unsigned char * ca_cert_data

size_t ca_cert_len

unsigned char * client_cert_data

size_t client_cert_len

unsigned char * client_key_data

size_t client_key_len

char client_key_passwd [PASSWORD_MAX_LENGTH]

char ca_cert_hash [HASH_MAX_LENGTH]

char domain_match [DOMAIN_MATCH_MAX_LENGTH]

char domain_suffix_match [DOMAIN_MATCH_MAX_LENGTH]

unsigned char * ca_cert2_data

size_t ca_cert2_len

unsigned char * client_cert2_data

size_t client_cert2_len

unsigned char * client_key2_data

size_t client_key2_len

char client_key2_passwd [PASSWORD_MAX_LENGTH]

unsigned char * dh_data

size_t dh_len

unsigned char * server_cert_data

size_t server_cert_len

unsigned char * server_key_data

size_t server_key_len

char server_key_passwd [PASSWORD_MAX_LENGTH]

size_t nusers

char identities [MAX_USERS][IDENTITY_MAX_LENGTH]

char passwords [MAX_USERS][PASSWORD_MAX_LENGTH]

char pac_opaque_encr_key [PAC_OPAQUE_ENCR_KEY_MAX_LENGTH]

char a_id [A_ID_MAX_LENGTH]

uint8_t fast_prov

unsigned char * **dpp_connector**

unsigned char * **dpp_c_sign_key**

unsigned char * **dpp_net_access_key**

Detailed Description Network security configuration

Field Documentation

enum wlan_security_type wlan_network_security::type

Type of network security to use. Specified by enum wlan_security_type.

int wlan_network_security::key_mgmt

Key management type

struct wlan_cipher wlan_network_security::mcstCipher

Type of network security Group Cipher suite

struct wlan_cipher wlan_network_security::ucstCipher

Type of network security Pairwise Cipher suite

unsigned wlan_network_security::pkc

Proactive key caching

int wlan_network_security::group_cipher

Type of network security Group Cipher suite

int wlan_network_security::pairwise_cipher

Type of network security Pairwise Cipher suite

int wlan_network_security::group_mgmt_cipher

Type of network security Pairwise Cipher suite

bool wlan_network_security::is_pmf_required

Is PMF (protected management frame) required

char wlan_network_security::psk[WLAN_PSK_MAX_LENGTH]

Pre-shared key (network password). For WEP networks this is a hex byte sequence of length psk_len, for WPA and WPA2 networks this is an ASCII pass-phrase of length psk_len. This field is ignored for networks with no security.

uint8_t wlan_network_security::psk_len

Length of the WEP key or WPA/WPA2 pass phrase, WLAN_PSK_MIN_LENGTH to WLAN_PSK_MAX_LENGTH. Ignored for networks with no security.

char wlan_network_security::password[WLAN_PASSWORD_MAX_LENGTH+1]

WPA3 SAE password, for WPA3 SAE networks this is an ASCII password of length password_len. This field is ignored for networks with no security.

size_t wlan_network_security::password_len

Length of the WPA3 SAE Password, WLAN_PASSWORD_MIN_LENGTH to WLAN_PASSWORD_MAX_LENGTH. Ignored for networks with no security.

char* wlan_network_security::sae_groups

Preference list of enabled groups for SAE. By default (if this parameter is not set), the mandatory group 19 (ECC group defined over a 256-bit prime order field) is preferred, but other groups are also enabled. If this parameter is set, the groups is tried in the indicated order.

uint8_t wlan_network_security::pwe_derivation

SAE (Simultaneous Authentication of Equals) mechanism for PWE (Password Element) derivation

uint8_t wlan_network_security::transition_disable

Transition Disable indication

char* wlan_network_security::owe_groups

OWE Groups

char wlan_network_security::pmk[WLAN_PMK_LENGTH]

PMK (pairwise master key). When pmk_valid is set, this is the PMK calculated from the PSK for WPA/PSK networks. If pmk_valid is not set, this field is ignored. When adding networks with wlan_add_network, users can initialize PMK and set pmk_valid in lieu of setting the psk. After successfully connecting to a WPA/PSK network, users can call wlan_get_current_network to inspect pmk_valid and pmk. Thus, the pmk value can be populated in subsequent calls to wlan_add_network. This saves the CPU time required to otherwise calculate the PMK.

bool wlan_network_security::pmk_valid

Flag reporting whether PMK is valid or not.

int8_t wlan_network_security::mfpc

Management frame protection capable (MFPC)

int8_t wlan_network_security::mfpr

Management frame protection required (MFPR)

unsigned wlan_network_security::wpa3_ent

WPA3 Enterprise mode

unsigned wlan_network_security::wpa3_sb

WPA3 Enterprise Suite B mode

unsigned wlan_network_security::wpa3_sb_192

WPA3 Enterprise Suite B 192 mode

unsigned wlan_network_security::eap_ver

EAP (Extensible Authentication Protocol) version

unsigned wlan_network_security::peap_label

PEAP (Protected Extensible Authentication Protocol) label

uint8_t wlan_network_security::eap_crypto_binding

crypto_binding option can be used to control WLAN_SECURITY_EAP_PEAP_MSCHAPV2, WLAN_SECURITY_EAP_PEAP_TLS and WLAN_SECURITY_EAP_PEAP_GTC version 0 cryptobinding behavior: 0 = do not use cryptobinding (default) 1 = use cryptobinding if server supports it 2 = require cryptobinding

unsigned wlan_network_security::eap_result_ind

eap_result_ind=1 can be used to enable WLAN_SECURITY_EAP_SIM, WLAN_SECURITY_EAP_AKA and WLAN_SECURITY_EAP_AKA_PRIME to use protected result indication.

unsigned char wlan_network_security::tls_cipher

Cipher for EAP TLS (Extensible Authentication Protocol Transport Layer Security)

char wlan_network_security::identity[IDENTITY_MAX_LENGTH]

Identity string for EAP

char wlan_network_security::anonymous_identity[IDENTITY_MAX_LENGTH]

Anonymous identity string for EAP

char wlan_network_security::eap_password[PASSWORD_MAX_LENGTH]

Password string for EAP.

bool wlan_network_security::verify_peer_cert

whether verify peer with CA or not false: not verify, true: verify.

unsigned char* wlan_network_security::ca_cert_data

CA (Certificate Authority) certification blob (Binary Large Object) in PEM (Base64 ASCII)/DER (binary) format

size_t wlan_network_security::ca_cert_len

CA (Certificate Authority) certification blob (Binary Large Object) length

unsigned char* wlan_network_security::client_cert_data

Client certification blob (Binary Large Object) in PEM (Base64 ASCII)/DER (binary) format

size_t wlan_network_security::client_cert_len

Client certification blob (Binary Large Object) length

unsigned char* wlan_network_security::client_key_data

Client key blob (Binary Large Object)

size_t wlan_network_security::client_key_len

Client key blob (Binary Large Object) length

char wlan_network_security::client_key_passwd[PASSWORD_MAX_LENGTH]

Client key password

char wlan_network_security::ca_cert_hash[HASH_MAX_LENGTH]

CA certification HASH

char wlan_network_security::domain_match[DOMAIN_MATCH_MAX_LENGTH]

Domain

char wlan_network_security::domain_suffix_match[DOMAIN_MATCH_MAX_LENGTH]

Domain Suffix

unsigned char* wlan_network_security::ca_cert2_data

CA (Certificate Authority) certification blob (Binary Large Object) in PEM (Base64 ASCII)/DER (binary) format for phase two

size_t wlan_network_security::ca_cert2_len

CA (Certificate Authority) certification blob (Binary Large Object) length for phase two

unsigned char* wlan_network_security::client_cert2_data

Client certification blob (Binary Large Object) in PEM (Base64 ASCII)/DER (binary) format for phase two

size_t wlan_network_security::client_cert2_len

Client certification blob (Binary Large Object) length for phase two

unsigned char* wlan_network_security::client_key2_data

Client key blob (Binary Large Object) for phase two

size_t wlan_network_security::client_key2_len

Client key blob (Binary Large Object) length for phase two

char wlan_network_security::client_key2_passwd[PASSWORD_MAX_LENGTH]

Client key password for phase two

unsigned char* wlan_network_security::dh_data

DH (Diffie Hellman) parameters blob (Binary Large Object)

size_t wlan_network_security::dh_len

DH (Diffie Hellman) parameters blob (Binary Large Object) length

unsigned char* wlan_network_security::server_cert_data

Server certification blob (Binary Large Object) in PEM (Base64 ASCII)/DER (binary) format

size_t wlan_network_security::server_cert_len

Server certification blob (Binary Large Object) length

unsigned char* wlan_network_security::server_key_data

Server key blob (Binary Large Object)

size_t wlan_network_security::server_key_len

Server key blob (Binary Large Object) length

char wlan_network_security::server_key_passwd[PASSWORD_MAX_LENGTH]

Server key password

size_t wlan_network_security::nusers

Number of EAP users

char wlan_network_security::identities[MAX_USERS][IDENTITY_MAX_LENGTH]

User Identities

char wlan_network_security::passwords[MAX_USERS][PASSWORD_MAX_LENGTH]

User Passwords

char wlan_network_security::pac_opaque_encr_key[PAC_OPAQUE_ENCR_KEY_MAX_LENGTH]

Encryption key for EAP-FAST PAC-Opaque values

char wlan_network_security::a_id[A_ID_MAX_LENGTH]

EAP-FAST authority identity (A-ID)

uint8_t wlan_network_security::fast_prov

EAP-FAST provisioning modes: 0 = provisioning disabled 1 = only anonymous provisioning allowed 2 = only authenticated provisioning allowed 3 = both provisioning modes allowed (default)

The documentation for this struct was generated from the following file: wlan.h

wlan_scan_result Struct Reference

Data Fields char ssid [WLAN_NETWORK_NAME_MAX_LENGTH+1]

unsigned int ssid_len

char bssid [IEEEtypes_ADDRESS_SIZE]

unsigned int channel

enum wlan_bss_type type

enum wlan_bss_role role

unsigned dot11n: 1

unsigned dot11ac: 1

unsigned dot11ax: 1

unsigned wmm: 1

unsigned wps: 1

unsigned int wps_session

unsigned wep: 1

unsigned wpa: 1

unsigned wpa2: 1

unsigned wpa2_sha256: 1

unsigned owe: 1

unsigned wpa3_sae: 1

unsigned wpa2_entp: 1

unsigned wpa3_entp: 1

unsigned wpa3_1x_sha256: 1

unsigned wpa3_1x_sha384: 1

unsigned ft_1x: 1

unsigned ft_1x_sha384: 1

unsigned ft_psk: 1

unsigned ft_sae: 1

unsigned char rssi

char trans_ssid [WLAN_NETWORK_NAME_MAX_LENGTH+1]

unsigned int trans_ssid_len

char trans_bssid [IEEEtypes_ADDRESS_SIZE]

uint16_t beacon_period

uint8_t dtim_period

t_u8 ap_mfpc

t_u8 ap_mfpr

t_u8 ap_pwe

bool neighbor_report_supported

bool bss_transition_supported

Detailed Description Scan result

Field Documentation

char wlan_scan_result::ssid[WLAN_NETWORK_NAME_MAX_LENGTH+1]

The network SSID, represented as a NULL-terminated C string of 0 to 32 characters. If the network has a hidden SSID, this can be the empty string.

unsigned int wlan_scan_result::ssid_len

SSID length

char wlan_scan_result::bssid[IEEEtypes_ADDRESS_SIZE]

The network BSSID, represented as a 6-byte array.

unsigned int wlan_scan_result::channel

The network channel.

enum wlan_bss_type wlan_scan_result::type

The Wi-Fi network type.

enum wlan_bss_role wlan_scan_result::role

The Wi-Fi network mode.

unsigned wlan_scan_result::dot11n

The network supports 802.11N. This is set to 0 if the network does not support 802.11N or if the system does not have 802.11N support enabled.

unsigned wlan_scan_result::dot11ac

The network supports 802.11AC. This is set to 0 if the network does not support 802.11AC or if the system does not have 802.11AC support enabled.

unsigned wlan_scan_result::dot11ax

The network supports 802.11AX. This is set to 0 if the network does not support 802.11AX or if the system does not have 802.11AX support enabled.

unsigned wlan_scan_result::wmm

The network supports WMM. This is set to 0 if the network does not support WMM or if the system does not have WMM support enabled.

unsigned wlan_scan_result::wps

The network supports WPS. This is set to 0 if the network does not support WPS or if the system does not have WPS support enabled.

unsigned int wlan_scan_result::wps_session

WPS Type WPS_SESSION_PBC/ WPS_SESSION_PIN

unsigned wlan_scan_result::wep

The network uses WEP security.

unsigned wlan_scan_result::wpa

The network uses WPA security.

unsigned wlan_scan_result::wpa2

The network uses WPA2 security

unsigned wlan_scan_result::wpa2_sha256

The network uses WPA2 SHA256 security

unsigned wlan_scan_result::owe

The network uses OWE security

unsigned wlan_scan_result::wpa3_sae

The network uses WPA3 SAE security

unsigned wlan_scan_result::wpa2_entp

The network uses WPA2 Enterprise security

unsigned wlan_scan_result::wpa3_entp

The network uses WPA3 Enterprise security

unsigned wlan_scan_result::wpa3_1x_sha256

The network uses WPA3 Enterprise SHA256 security

unsigned wlan_scan_result::wpa3_1x_sha384

The network uses WPA3 Enterprise SHA384 security

unsigned wlan_scan_result::ft_1x

The network uses FT 802.1x security

unsigned wlan_scan_result::ft_1x_sha384

The network uses FT 892.1x SHA384 security

unsigned wlan_scan_result::ft_psk

The network uses FT PSK security

unsigned wlan_scan_result::ft_sae

The network uses FT SAE security

unsigned char wlan_scan_result::rssi

The signal strength of the beacon

char wlan_scan_result::trans_ssid[WLAN_NETWORK_NAME_MAX_LENGTH+1]

The network SSID, represented as a NULL-terminated C string of 0 to 32 characters. If the network has a hidden SSID, this should be the empty string.

unsigned int wlan_scan_result::trans_ssid_len

SSID length

char wlan_scan_result::trans_bssid[IEEEtypes_ADDRESS_SIZE]

The network BSSID, represented as a 6-byte array.

uint16_t wlan_scan_result::beacon_period

Beacon period

uint8_t wlan_scan_result::dtim_period

DTIM (delivery traffic indication map) period

t_u8 wlan_scan_result::ap_mfpc

MFPC (Management Frame Protection Capable) bit of AP (Access Point)

t_u8 wlan_scan_result::ap_mfpr

MFPR (Management Frame Protection Required) bit of AP (Access Point)

t_u8 wlan_scan_result::ap_pwe

PWE (Password Element) bit of AP (Access Point)

bool wlan_scan_result::neighbor_report_supported

Neighbor report support

bool wlan_scan_result::bss_transition_supported

bss transition support

The documentation for this struct was generated from the following file: wlan.h

File Documentation

wlan.h File Reference This file provides Wi-Fi APIs for the application.

Function Documentation

int verify_scan_duration_value (int *scan_duration*)

Check whether the scan duration is valid or not.

Parameters

Returns

0 if the time is valid, else return -1.

int verify_scan_channel_value (int *channel*)

Check whether the scan channel is valid or not.

Parameters

Returns

0 if the channel is valid, else return -1.

int verify_split_scan_delay (int *delay*)

Check whether the scan delay time is valid or not.

Parameters

Returns

0 if the time is valid, else return -1.

int set_scan_params (struct wifi_scan_params_t * *wifi_scan_params*)

Set the scan parameters.

Parameters

Returns

0 if Wi-Fi scan parameters are set successfully, else return -1.

int get_scan_params (struct wifi_scan_params_t * *wifi_scan_params*)

Get the scan parameters.

Parameters

Returns

WM_SUCCESS.

int wlan_get_current_rssi (short * *rssi*)

Get the current RSSI value.

Parameters

Returns

WM_SUCCESS.

int wlan_get_current_nf (void)

Get the current noise floor.

Returns

The noise floor value

int wlan_init (const uint8_t * *fw_start_addr*, const size_t *size*)

Initialize the Wi-Fi driver and create the Wi-Fi driver thread.

Parameters

Returns

WM_SUCCESS if the Wi-Fi connection manager service has initialized successfully.

Negative value if initialization failed.

int wlan_start (int(*)(enum wlan_event_reason reason, void *data) *cb*)

Start the Wi-Fi connection manager service.

This function starts the Wi-Fi connection manager.

Note

The status of the Wi-Fi connection manager is notified asynchronously through the callback, *cb* , with a WLAN_REASON_INITIALIZED event (if initialization succeeded) or WLAN_REASON_INITIALIZATION_FAILED (if initialization failed). If the Wi-Fi connection manager fails to initialize, the caller should stop Wi-Fi connection manager via wlan_stop() and try wlan_start() again.

Parameters

Returns

WM_SUCCESS if the Wi-Fi connection manager service has started successfully.

-WM_E_INVAL if the *cb* pointer is NULL.

-WM_FAIL if an internal error occurred.

WLAN_ERROR_STATE if the Wi-Fi connection manager is already running.

int wlan_stop (void)

Stop the Wi-Fi connection manager service.

This function stops the Wi-Fi connection manager, causing the station interface to disconnect from the currently connected network and stop the uAP interface.

Returns

WM_SUCCESS if the Wi-Fi connection manager service has been stopped successfully.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running.

void wlan_deinit (int *action*)

Deinitialize the Wi-Fi driver, send a shutdown command to the Wi-Fi firmware and delete the Wi-Fi driver thread.

Parameters

int wlan_remove_all_network_profiles (void)

Stop and remove all Wi-Fi network profiles.

Returns

WM_SUCCESS if successful otherwise return -WM_E_INVAL.

void wlan_reset (cli_reset_option *ResetOption*)

Reset the driver.

Parameters

int wlan_remove_all_networks (void)

Stop and remove all Wi-Fi network (access point).

Returns

WM_SUCCESS if successful.

void wlan_destroy_all_tasks (void)

This API destroys all tasks.

int wlan_is_started (void)

Retrieve the status information of if Wi-Fi started.

Returns

TRUE if Wi-Fi network is started.

FALSE if not started.

int wlan_set_get_rx_abort_cfg (struct wlan_rx_abort_cfg * *cfg*, t_u16 *action*)

Set/Get RX abort configuration to/from firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_rx_abort_cfg_ext (const struct wlan_rx_abort_cfg_ext * *cfg*)

Set the dynamic RX abort configuration to firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_rx_abort_cfg_ext (struct wlan_rx_abort_cfg_ext * *cfg*)

Get the dynamic RX abort configuration from firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_get_cck_desense_cfg (struct wlan_cck_desense_cfg * *cfg*, t_u16 *action*)

Set/Get CCK (complementary code keying) desense configuration to/from firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

void wlan_initialize_uap_network (struct wlan_network * *net*)

Initialize the uAP network information.

This API initializes a uAP network with default configurations. The network ssid, passphrase is initialized to NULL. Channel is set to auto. The IP Address of the uAP interface is 192.168.10.1/255.255.255.0. The network name is set to 'uap-network'.

Parameters

void wlan_initialize_sta_network (struct wlan_network * *net*)

Initialize the station network information.

This API initializes a station network with default configurations. The network ssid, passphrase is initialized to NULL. Channel is set to auto.

Parameters

int wlan_add_network (struct wlan_network * *network*)

Add a network profile to the list of known networks.

This function copies the contents of *network* to the list of known networks in the Wi-Fi connection manager. The network's 'name' field is unique and between WLAN_NETWORK_NAME_MIN_LENGTH and WLAN_NETWORK_NAME_MAX_LENGTH characters. The network must specify at least an SSID or BSSID. the Wi-Fi connection manager can store up to WLAN_MAX_KNOWN_NETWORKS networks.

Note

Profiles for the station interface may be added only when the station interface is in the WLAN_DISCONNECTED or WLAN_CONNECTED state.

This API can be used to add profiles for station or uAP interfaces.

Set mfpc and mfpr to -1 for default configurations.

Parameters

Returns

WM_SUCCESS if the contents pointed to by *network* have been added to the Wi-Fi connection manager.

-WM_E_INVAL if *network* is NULL or the network name is not unique or the network name length is not valid or network security is WLAN_SECURITY_WPA3_SAE but Management Frame Protection Capable is not enabled. in wlan_network_security field. if network security type is WLAN_SECURITY_WPA or WLAN_SECURITY_WPA2 or WLAN_SECURITY_WPA_WPA2_MIXED, but the passphrase length is less than 8 or greater than 63, or the psk length equal to 64 but not hexadecimal digits. if network security type is WLAN_SECURITY_WPA3_SAE, but the password length is less than

8 or greater than 255. if network security type is WLAN_SECURITY_WEP_OPEN or WLAN_SECURITY_WEP_SHARED.

-WM_E_NOMEM if there was no room to add the network.

WLAN_ERROR_STATE if the Wi-Fi connection manager was running and not in the WLAN_DISCONNECTED, WLAN_ASSOCIATED or WLAN_CONNECTED state.

int wlan_remove_network (const char * *name*)

Remove a network profile from the list of known networks.

This function removes a network (identified by its name) from the WLAN Connection Manager, disconnecting from that network if connected.

Note

This function is asynchronous if it is called while the WLAN Connection Manager is running and connected to the network to be removed. In that case, the Wi-Fi connection manager can disconnect from the network and generate an event with reason WLAN_REASON_USER_DISCONNECT. This function is synchronous otherwise.

This API can be used to remove profiles for station or uAP interfaces. Station network can not be removed if it is in WLAN_CONNECTED state and uAP network can not be removed if it is in WLAN_UAP_STARTED state.

Parameters

Returns

WM_SUCCESS if the network named *name* was removed from the Wi-Fi connection manager successfully. Otherwise, the network is not removed.

WLAN_ERROR_STATE if the Wi-Fi connection manager was running and the station interface was not in the WLAN_DISCONNECTED state.

-WM_E_INVAL if *name* is NULL or the network was not found in the list of known networks.

-WM_FAIL if an internal error occurred while trying to disconnect from the network specified for removal.

int wlan_connect (char * *name*)

Connect to a Wi-Fi network (access point).

When this function is called, Wi-Fi connection manager starts connection attempts to the network specified by *name* . The connection result can be notified asynchronously to the WLCMGR callback when the connection process has completed.

When connecting to a network, the event refers to the connection attempt to that network.

Calling this function when the station interface is in the WLAN_DISCONNECTED state should, if successful, cause the interface to transition into the WLAN_CONNECTING state. If the connection attempt succeeds, the station interface should transition to the WLAN_CONNECTED state, otherwise it should return to the WLAN_DISCONNECTED state. If this function is called while the station interface is in the WLAN_CONNECTING or WLAN_CONNECTED state, the Wi-Fi connection manager should first cancel its connection attempt or disconnect from the network, respectively, and generate an event with reason WLAN_REASON_USER_DISCONNECT. This should be followed by a second event that reports the result of the new connection attempt.

If the connection attempt was successful the WLCMGR callback is notified with the event WLAN_REASON_SUCCESS, while if the connection attempt fails then either of the events, WLAN_REASON_NETWORK_NOT_FOUND, WLAN_REASON_NETWORK_AUTH_FAILED, WLAN_REASON_CONNECT_FAILED or WLAN_REASON_ADDRESS_FAILED are reported as appropriate.

Parameters

Returns

WM_SUCCESS if a connection attempt was started successfully

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running.

-WM_E_INVAL if there are no known networks to connect to or the network specified by *name* is not in the list of known networks or network *name* is NULL.

-WM_FAIL if an internal error has occurred.

int wlan_connect_opt (char * *name*, bool *skip_dfs*)

Connect to a Wi-Fi network (access point) with options.

When this function is called, the Wi-Fi connection manager starts connection attempts to the network specified by *name* . The connection result should be notified asynchronously to the WLCMGR callback when the connection process has completed.

When connecting to a network, the event refers to the connection attempt to that network.

Calling this function when the station interface is in the WLAN_DISCONNECTED state should, if successful, cause the interface to transition into the WLAN_CONNECTING state. If the connection attempt succeeds, the station interface should transition to the WLAN_CONNECTED state, otherwise it should return to the WLAN_DISCONNECTED state. If this function is called while the station interface is in the WLAN_CONNECTING or WLAN_CONNECTED state, the Wi-Fi connection manager should first cancel its connection attempt or disconnect from the network, respectively, and generate an event with reason WLAN_REASON_USER_DISCONNECT. This should be followed by a second event that reports the result of the new connection attempt.

If the connection attempt was successful the WLCMGR callback is notified with the event WLAN_REASON_SUCCESS, while if the connection attempt fails then either of the events, WLAN_REASON_NETWORK_NOT_FOUND, WLAN_REASON_NETWORK_AUTH_FAILED, WLAN_REASON_CONNECT_FAILED or WLAN_REASON_ADDRESS_FAILED are reported as appropriate.

Parameters

Returns

WM_SUCCESS if a connection attempt was started successfully

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running.

-WM_E_INVAL if there are no known networks to connect to or the network specified by *name* is not in the list of known networks or network *name* is NULL.

-WM_FAIL if an internal error has occurred.

int wlan_reassociate (void)

Reassociate to a Wi-Fi network (access point).

When this function is called, the Wi-Fi connection manager starts reassociation attempts using same SSID as currently connected network . The connection result should be notified asynchronously to the WLCMGR callback when the connection process has completed.

When connecting to a network, the event refers to the connection attempt to that network.

Calling this function when the station interface is in the WLAN_DISCONNECTED state should have no effect.

Calling this function when the station interface is in the WLAN_CONNECTED state should, if successful, cause the interface to reassociate to another network (access point).

If the connection attempt was successful the WLCMGR (Wi-Fi command manager) callback is notified with the event WLAN_REASON_SUCCESS, while if the connection attempt fails then either of the events, WLAN_REASON_NETWORK_AUTH_FAILED, WLAN_REASON_CONNECT_FAILED or WLAN_REASON_ADDRESS_FAILED are reported as appropriate.

Returns

WM_SUCCESS if a reassociation attempt was started successfully

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running. or Wi-Fi connection manager was not in WLAN_CONNECTED state.

-WM_E_INVAL if there are no known networks to connect to

-WM_FAIL if an internal error has occurred.

int wlan_disconnect (void)

Disconnect from the current Wi-Fi network (access point).

When this function is called, the Wi-Fi connection manager attempts to disconnect the station interface from its currently connected network (or cancel an in-progress connection attempt) and return to the WLAN_DISCONNECTED state. Calling this function has no effect if the station interface is already disconnected.

Note

This is an asynchronous function and successful disconnection should be notified using the WLAN_REASON_USER_DISCONNECT.

Returns

WM_SUCCESS if successful

WLAN_ERROR_STATE otherwise

int wlan_start_network (const char * *name*)

Start a Wi-Fi network (access point).

When this function is called, the Wi-Fi connection manager starts the network specified by *name* . The network with the specified *name* is first added using wlan_add_network and is a uAP network with a valid SSID.

Note

The WLCMGR callback is asynchronously notified of the status. On success, the event WLAN_REASON_UAP_SUCCESS is reported, while on failure, the event WLAN_REASON_UAP_START_FAILED is reported.

Parameters

Returns

WM_SUCCESS if successful.

WLAN_ERROR_STATE if in power save state or uAP already running.

-WM_E_INVAL if *name* was NULL or the network *name* was not found or it not have a specified SSID.

int wlan_stop_network (const char * *name*)

Stop a Wi-Fi network (access point).

When this function is called, the Wi-Fi connection manager stops the network specified by *name* . The specified network is a valid uAP network that has already been started.

Note

The WLCMGR callback is asynchronously notified of the status. On success, the event WLAN_REASON_UAP_STOPPED is reported, while on failure, the event WLAN_REASON_UAP_STOP_FAILED is reported.

Parameters

Returns

WM_SUCCESS if successful.

WLAN_ERROR_STATE if uAP is in power save state.

-WM_E_INVAL if *name* was NULL or the network *name* was not found or that the network *name* is not a uAP network or it is a uAP network but does not have a specified SSID.

int wlan_get_mac_address (unsigned char * *dest*)

Retrieve the Wi-Fi MAC address of the station interface.

This function copies the MAC address of the Wi-Fi station interface to the 6-byte array pointed to by *dest* . In the event of an error, nothing is copied to *dest* .

Parameters

Returns

WM_SUCCESS if the MAC address was copied.

-WM_E_INVAL if *dest* is NULL.

int wlan_get_mac_address_uap (uint8_t * *dest*)

Retrieve the Wi-Fi MAC address of the uAP interface.

This function copies the MAC address of the Wi-Fi uAP interface to the 6-byte array pointed to by *dest* . In the event of an error, nothing is copied to *dest* .

Parameters

Returns

WM_SUCCESS if the MAC address was copied.

-WM_E_INVAL if *dest* is NULL.

int wlan_get_wfd_mac_address (unsigned char * *dest*)

Retrieve the wireless MAC address of wfd interface.

This function copies the MAC address of the wireless interface to the 6-byte array pointed to by *dest* . In the event of an error, nothing is copied to *dest* .

Parameters

Returns

WM_SUCCESS if the MAC address was copied.

-WM_E_INVAL if *dest* is NULL.

int wlan_get_address (struct wlan_ip_config * *addr*)

Retrieve the IP address configuration of the station interface.

This function retrieves the IP address configuration of the station interface and copies it to the memory location pointed to by *addr* .

Note

This function may only be called when the station interface is in the WLAN_CONNECTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *addr* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or was not in the WLAN_CONNECTED state.

-WM_FAIL if an internal error occurred when retrieving IP address information from the TCP stack.

int wlan_get_uap_address (struct wlan_ip_config * *addr*)

Retrieve the IP address of the uAP interface.

This function retrieves the current IP address configuration of the uAP and copies it to the memory location pointed to by *addr* .

Note

This function may only be called when the uAP interface is in the WLAN_UAP_STARTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *addr* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or the uAP interface was not in the WLAN_UAP_STARTED state.

-WM_FAIL if an internal error occurred when retrieving IP address information from the TCP stack.

int wlan_get_uap_channel (int * *channel*)

Retrieve the channel of the uAP interface.

This function retrieves the channel number of the uAP and copies it to the memory location pointed to by *channel* .

Note

This function may only be called when the uAP interface is in the WLAN_UAP_STARTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *channel* is NULL.

-WM_FAIL if an internal error has occurred.

int wlan_get_current_network (struct wlan_network * *network*)

Retrieve the current network configuration of the station interface.

This function retrieves the current network configuration of the station interface when the station interface is in the WLAN_CONNECTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *network* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_CONNECTED state.

int wlan_get_current_network_ssid (char * *ssid*)

Retrieve the current network ssid of the station interface.

This function retrieves the current network ssid of the station interface when the station interface is in the WLAN_CONNECTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *ssid* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_CONNECTED state.

int wlan_get_current_network_bssid (char * *bssid*)

Retrieve the current network bssid of the station interface.

This function retrieves the current network bssid of the station interface when the station interface is in the WLAN_CONNECTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *bssid* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_CONNECTED state.

int wlan_get_current_uap_network (struct wlan_network * *network*)

Retrieve the current network configuration of the uAP interface.

This function retrieves the current network configuration of the uAP interface when the uAP interface is in the WLAN_UAP_STARTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *network* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_UAP_STARTED state.

int wlan_get_current_uap_network_ssid (char * *ssid*)

Retrieve the current network ssid of the uAP interface.

This function retrieves the current network ssid of the uAP interface when the uAP interface is in the WLAN_UAP_STARTED state.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *ssid* is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_UAP_STARTED state.

bool is_uap_started (void)

Retrieve the status information of the uAP interface.

Returns

TRUE if uAP interface is in WLAN_UAP_STARTED state.

FALSE otherwise.

bool is_sta_associated (void)

Retrieve the status information of the station interface.

Returns

TRUE if station interface is in or above the WLAN_ASSOCIATED state.

FALSE otherwise.

bool is_sta_connected (void)

Retrieve the status information of the station interface.

Returns

TRUE if station interface is in WLAN_CONNECTED state.

FALSE otherwise.

bool is_sta_ipv4_connected (void)

Retrieve the status information of the ipv4 network of the station interface.

Returns

TRUE if ipv4 network of the station interface is in WLAN_CONNECTED state.

FALSE otherwise.

bool is_sta_ipv6_connected (void)

Retrieve the status information of the ipv6 network of the station interface.

Returns

TRUE if ipv6 network of the station interface is in WLAN_CONNECTED state.

FALSE otherwise.

int wlan_get_network (unsigned int *index*, struct wlan_network * *network*)

Retrieve the information about a known network using *index* .

This function retrieves the contents of a network at *index* in the list of known networks maintained by the Wi-Fi connection manager and copies it to the location pointed to by *network* .

Note

wlan_get_network_count() can be used to retrieve the number of known networks. wlan_get_network() can be used to retrieve information about networks at *index* 0 to one minus the number of networks.

This function can be called regardless of whether the Wi-Fi connection manager is running or not. Calls to this function are synchronous.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *network* is NULL or *index* is out of range.

int wlan_get_network_byname (char * *name*, struct wlan_network * *network*)

Retrieve information about a known network using *name* .

This function retrieves the contents of a named network in the list of known networks maintained by the Wi-Fi connection manager and copies it to the location pointed to by *network* .

Note

This function can be called regardless of whether the Wi-Fi Connection Manager is running or not. Calls to this function are synchronous.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *network* is NULL or *name* is NULL.

int wlan_get_network_count (unsigned int * *count*)

Retrieve the number of networks known to the Wi-Fi connection manager.

This function retrieves the number of known networks in the list maintained by the Wi-Fi connection manager and copies it to *count* .

Note

This function can be called regardless of whether the Wi-Fi Connection Manager is running or not. Calls to this function are synchronous.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *count* is NULL.

int wlan_get_connection_state (enum wlan_connection_state * *state*)

Retrieve the connection state of the station interface.

This function retrieves the connection state of the station interface, which is one of WLAN_DISCONNECTED, WLAN_CONNECTING, WLAN_ASSOCIATED or WLAN_CONNECTED.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *state* is NULL

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running.

int wlan_get_uap_connection_state (enum wlan_connection_state * *state*)

Retrieve the connection state of the uAP interface.

This function retrieves the connection state of the uAP interface, which is one of WLAN_UAP_STARTED, or WLAN_UAP_STOPPED.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *state* is NULL

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running.

int wlan_scan (int(*)(unsigned int count) *cb*)

Scan for Wi-Fi networks.

When this function is called, the Wi-Fi connection manager starts scan for Wi-Fi networks. On completion of the scan the Wi-Fi connection manager can call the specified callback function *cb* . The callback function should then retrieve the scan results by using the wlan_get_scan_result() function.

Note

This function may only be called when the station interface is in the WLAN_DISCONNECTED or WLAN_CONNECTED state. scan is disabled in the WLAN_CONNECTING state.

This function should block until it can issue a scan request if called while another scan is in progress.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_NOMEM if failed to allocated memory for wlan_scan_params_v2_t structure.

-WM_E_INVAL if *cb* scan result callback function pointer is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_DISCONNECTED or WLAN_CONNECTED states.

-WM_FAIL if an internal error has occurred and the system is unable to scan.

int wlan_scan_with_opt (wlan_scan_params_v2_t *t_wlan_scan_param*)

Scan for Wi-Fi networks using options provided.

When this function is called, the Wi-Fi connection manager starts scanning for Wi-Fi networks. On completion of the scan the Wi-Fi connection manager should call the specified callback function *t_wlan_scan_param.cb* . The callback function should then retrieve the scan results by using the wlan_get_scan_result() function.

Note

This function may only be called when the station interface is in the WLAN_DISCONNECTED or WLAN_CONNECTED state. scan is disabled in the WLAN_CONNECTING state.

This function can block until it issues a scan request if called while another scan is in progress.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_NOMEM if failed to allocated memory for wlan_scan_params_v2_t structure.

-WM_E_INVAL if *cb* scan result callback function pointer is NULL.

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running or not in the WLAN_DISCONNECTED or WLAN_CONNECTED states.

-WM_FAIL if an internal error has occurred and the system is unable to scan.

int wlan_get_scan_result (unsigned int *index*, struct wlan_scan_result * *res*)

Retrieve a scan result.

This function can be called to retrieve scan results when the Wi-Fi connection manager has finished scanning. It is called from within the scan result callback (see wlan_scan()) as scan results are valid only in that context. The callback argument ‘count’ provides the number of scan results that can be retrieved and wlan_get_scan_result() can be used to retrieve scan results at *index* 0 through that number.

Note

This function may only be called in the context of the scan results callback.

Calls to this function are synchronous.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *res* is NULL

WLAN_ERROR_STATE if the Wi-Fi connection manager was not running

-WM_FAIL if the scan result at *index* could not be retrieved (that is, *index* is out of range).

int wlan_set_ed_mac_mode (wlan_ed_mac_ctrl_t *wlan_ed_mac_ctrl*)

Configure Energy Detect MAC mode for the station in the Wi-Fi Firmware.

Note

When ED MAC mode is enabled, the Wi-Fi Firmware can behave in the following way:

When the background noise had reached the Energy Detect threshold or above, the Wi-Fi chipset/module should hold the data transmission until the condition is removed. The 2.4GHz and 5GHz bands are configured separately.

Parameters

ed_offset_2g 0 - Default Energy Detect threshold (Default: 0x9) offset value range: 0x80 to 0x7F

Note

If 5GH enabled then add following parameters

ed_ctrl_5g 0 - disable EU adaptivity for 5GHz band

1 - enable EU adaptivity for 5GHz band

ed_offset_5g 0 - Default Energy Detect threshold(Default: 0xC)

offset value range: 0x80 to 0x7F

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_set_uap_ed_mac_mode (wlan_ed_mac_ctrl_t *wlan_ed_mac_ctrl*)

Configure Energy Detect MAC mode for the uAP in the Wi-Fi firmware.

Note

When ED MAC mode is enabled, the Wi-Fi Firmware can behave in the following way:

When the background noise had reached the Energy Detect threshold or above, the Wi-Fi chipset/module should hold data transmission until the condition is removed. The 2.4GHz and 5GHz bands are configured separately.

Parameters

ed_offset_2g 0 - Default energy detect threshold (Default: 0x9) offset value range: 0x80 to 0x7F

Note

If 5GH enabled then add following parameters

ed_ctrl_5g 0 - disable EU adaptivity for 5GHz band

1 - enable EU adaptivity for 5GHz band

ed_offset_5g 0 - Default energy detect threshold(Default: 0xC)

offset value range: 0x80 to 0x7F

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_get_ed_mac_mode (wlan_ed_mac_ctrl_t * *wlan_ed_mac_ctrl*)

This API can be used to get current ED MAC MODE configuration for station.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_get_uap_ed_mac_mode (wlan_ed_mac_ctrl_t * *wlan_ed_mac_ctrl*)

This API can be used to get current ED MAC MODE configuration for uAP.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

void wlan_set_cal_data (const uint8_t * *cal_data*, const unsigned int *cal_data_size*)

Set the Wi-Fi calibration data in the Wi-Fi firmware.

This function can be used to set the Wi-Fi calibration data in the firmware. This should be call before wlan_init() function.

Parameters

int wlan_set_mac_addr (uint8_t * *mac*)

Set the Wi-Fi MAC Address in the Wi-Fi firmware.

This function can be used to set Wi-Fi MAC Address in firmware. When called after Wi-Fi initialization done, the incoming MAC is treated as the STA MAC address directly. And mac[4] plus 1, the modified MAC is used as the uAP MAC address.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_set_sta_mac_addr (uint8_t * *mac*)

Set the Wi-Fi MAC address for the STA in the Wi-Fi firmware.

This function can be used to set the Wi-Fi MAC address for the station in the firmware. Should be called after Wi-Fi initialization done. It sets the station MAC address only.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_set_uap_mac_addr (uint8_t * *mac*)

Set the Wi-Fi MAC address for the uAP in the Wi-Fi firmware.

This function can be used to set the Wi-Fi MAC address for the uAP in the firmware. Should be called after Wi-Fi initialization done. It sets the uAP MAC address only.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_wmm_uapsd_qosinfo (t_u8 * *qos_info*, t_u8 *action*)

Set the QOS info of the UAPSD (unscheduled automatic power save delivery) in the Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_set_wmm_uapsd (t_u8 *uapsd_enable*)

Enable/Disable the UAPSD in the Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_sleep_period (unsigned int * *sleep_period*, t_u8 *action*)

Set/get UAPSD sleep period in the Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

t_u8 wlan_is_wmm_uapsd_enabled (void)

Check whether UAPSD is enabled or not.

Returns

true if UAPSD is enabled.

false if UAPSD is disabled.

void wlan_set_txrx_histogram (struct wlan_txrx_histogram_info * *txrx_histogram*, t_u8 * *data*)

Set TX RX histogram config. This function can be called to set TX RX histogram config.

Parameters

int wlan_set_roaming (const int *enable*, const uint8_t *rssi_low_threshold*)

Set soft roaming config.

This function can be used to enable/disable soft roaming by specifying the RSSI threshold.

Note

RSSI Threshold setting for soft roaming : The provided RSSI low threshold value is used to subscribe RSSI low event from the firmware. On reception of this event, the background scan is started in the firmware with the same RSSI threshold to find out APs with a better signal strength than the RSSI threshold.

If an AP with better signal strength is found, the reassociation is triggered. Otherwise the background scan is started again until the scan count reaches BG_SCAN_LIMIT.

If still AP is not found then Wi-Fi connection manager sends WLAN_REASON_BGSCAN_NETWORK_NOT_FOUND event to application. In this case, if application again wants to use soft roaming then it can call this API again or use wlan_set_rssi_low_threshold API to set RSSI low threshold again.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_get_roaming_status (void)

Get the roaming status.

Returns

1 if roaming is enabled.

0 if roaming is disbled.

int wlan_wowlan_config (uint8_t *is_mef*, t_u32 *wake_up_conds*)

Wowlan (wake on wireless LAN) configuration. This function may be called to configure host sleep in firmware.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

void wlan_config_host_sleep (bool *is_manual*, t_u8 *is_periodic*)

Host sleep configuration. This function may be called to configure host sleep in firmware.

Parameters

status_t wlan_hs_send_event (int *id*, void * *data*)

This function sends host sleep events to mon_thread

Parameters

Returns

kStatus_Success if successful else return -WM_FAIL.

void wlan_cancel_host_sleep (void)

Cancel host sleep. This function is called to cancel the host sleep in the firmware.

void wlan_clear_host_sleep_config (void)

Clear host sleep configurations in driver. This function clears all the host sleep related configures in driver.

int wlan_set_multicast (t_u8 *mef_action*)

This function set multicast MEF (memory efficient filtering) entry

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_set_ieeeps_cfg (struct wlan_ieeeps_config * *ps_cfg*)

Set configuration parameters of IEEE power save mode.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

void wlan_configure_listen_interval (int *listen_interval*)

Configure listening interval of IEEE power save mode.

Note

Delivery traffic indication message (DTIM) : It is a concept in 802.11 It is a time duration after which AP can send out buffered BROADCAST / MULTICAST data and stations connected to the AP should wakeup to take this broadcast / multicast data.

Traffic Indication Map (TIM) : It is a bitmap which the AP sends with each beacon. The bitmap has one bit each for a station connected to AP.

Each station is recognized by an association ID (AID). If AP has buffered data for a station, it will set corresponding bit of bitmap in TIM based on AID. Ideally AP does not buffer any unicast data it just sends unicast data to the station on every beacon when station is not sleeping.

When broadcast data / multicast data is to be send AP sets bit 0 of TIM indicating broadcast / multicast.

The occurrence of DTIM is defined by AP.

Each beacon has a number indicating period at which DTIM occurs.

The number is expressed in terms of number of beacons.

This period is called DTIM Period / DTIM interval.

For example:

If AP has DTIM period = 3 the stations connected to AP have to wake up (if they are sleeping) to receive broadcast /multicast data on every third beacon.

Generic:

When DTIM period is X AP buffers broadcast data / multicast data for X beacons. Then it transmits the data no matter whether station is awake or not.

Listen interval:

This is time interval on station side which indicates when station can be awake to listen i.e. accept data.

Long listen interval:

It comes into picture when station sleeps (IEEE PS) and it does not want to wake up on every DTIM So station is not worried about broadcast data/multicast data in this case.

This should be a design decision what should be chosen Firmware suggests values which are about 3 times DTIM at the max to gain optimal usage and reliability.

In the IEEE power save mode, the Wi-Fi firmware goes to sleep and periodically wakes up to check if the AP has any pending packets for it. A longer listen interval implies that the Wi-Fi SoC stays in power save for a longer duration at the cost of additional delays while receiving data. Note that choosing incorrect value for listen interval causes poor response from device during data transfer. Actual listen interval selected by firmware is equal to closest DTIM.

For example:

AP beacon period : 100 ms

AP DTIM period : 2

Application request value: 500ms

Actual listen interval = 400ms (This is the closest DTIM). Actual listen interval set should be a multiple of DTIM closest to but lower than the value provided by the application.

This API can be called before/after association. The configured listen interval can be used in subsequent association attempt.

Parameters

void wlan_configure_delay_to_ps (unsigned int *timeout_ms*)

Set timeout configuration before Wi-Fi power save mode.

Parameters

void wlan_configure_idle_time (unsigned int *timeout_ms*)

Set timeout value before Wi-Fi enter deep sleep mode.

param [in] timeout_ms: timout time, in milliseconds.

Note

The minimum value of timeout_ms is 10.

unsigned int wlan_get_idle_time (void)

Get timeout value of deep sleep mode, in milliseconds.

Returns

idle time value.

unsigned short wlan_get_listen_interval (void)

Get listen interval .

Returns

listen interval value.

unsigned int wlan_get_delay_to_ps (void)

Get delay time for Wi-Fi power save mode.

Returns

delay time value.

bool wlan_is_power_save_enabled (void)

Check whether Wi-Fi power save is enabled or not.

Returns

TRUE if Wi-Fi power save is enabled, else return FALSE.

void wlan_configure_null_pkt_interval (int *time_in_secs*)

Configure NULL packet interval of IEEE power save mode.

Note

In IEEE PS (power save), station sends a NULL packet to AP to indicate that the station is alive and maintain connection with the AP. If null packet is not sent some APs may disconnect station which might lead to a loss of connectivity. The time is specified in seconds. Default value is 30 seconds.

This API should be called before configuring IEEE Power save.

Parameters

int wlan_set_antcfg (uint32_t *ant*, uint16_t *evaluate_time*, uint8_t *evaluate_mode*)

This API can be used to set the mode of TX/RX antenna. If SAD (software antenna diversity) is enabled, this API can also be used to set SAD antenna evaluate time interval(antenna mode is antenna diversity when set SAD evaluate time interval).

Parameters

Returns

WM_SUCCESS if successful.

WLAN_ERROR_STATE if unsuccessful.

int wlan_get_antcfg (uint32_t * *ant*, uint16_t * *evaluate_time*, uint8_t * *evaluate_mode*, uint16_t * *current_antenna*)

This API can be used to get the mode of TX/RX antenna. If SAD (software antenna diversity) is enabled, this API can also be used to get SAD antenna evaluate time interval(antenna mode is antenna diversity when set SAD evaluate time interval).

Parameters

Returns

WM_SUCCESS if successful.

WLAN_ERROR_STATE if unsuccessful.

char* wlan_get_firmware_version_ext (void)

Get the Wi-Fi firmware version extension string.

Note

This API does not allocate memory for pointer. It just returns pointer of WLCMGR internal static buffer. So no need to free the pointer by caller.

Returns

Wi-Fi firmware version extension string pointer stored in WLCMGR

void wlan_version_extended (void)

Use this API to print Wi-Fi driver and firmware extended version on console.

Note

Call this API when SDK_DEBUGCONSOLE not set to DEBUGCONSOLE_DISABLE.

int wlan_get_tsf (uint32_t * *tsf_high*, uint32_t * *tsf_low*)

Use this API to get the TSF (timing synchronization function) from Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_ieeeps_on (unsigned int *wakeup_conditions*)

Enable IEEE power save with host sleep configuration

When enabled, Wi-Fi SoC is opportunistically put into IEEE power save mode. Before putting the Wi-Fi SoC in power save this also sets the host sleep configuration on the SoC as specified. This makes the SoC generate a wakeup for the processor if any of the wakeup conditions are met.

Parameters

Note

IEEE power save mode applies only when STA has connected to an AP. It could be enabled/disabled when STA connected or disconnected, but only take effect when STA has connected to an AP.

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

int wlan_ieeeps_off (void)

Turn off IEEE power save mode.

Note

IEEE power save mode applies only when STA has connected to an AP. It could be enabled/disabled when STA connected or disconnected, but only take effect when STA has connected to an AP.

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

int wlan_deepsleepps_on (void)

Turn on deep sleep power save mode.

Note

deep sleep power save mode only applies when STA disconnected. It could be enabled/disabled when STA connected or disconnected, but only take effect when STA disconnected.

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

int wlan_deepsleepps_off (void)

Turn off deep sleep power save mode.

Note

deep sleep power save mode only applies when STA disconnected. It could be enabled/disabled when STA connected or disconnected, but only take effect when STA disconnected.

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL otherwise.

int wlan_tcp_keep_alive (wlan_tcp_keep_alive_t * *keep_alive*)

Use this API to configure the TCP keep alive parameters in Wi-Fi firmware. wlan_tcp_keep_alive_t provides the parameters which are available for configuration.

Note

To reset current TCP keep alive configuration, just set the reset member of wlan_tcp_keep_alive_t with value 1, all other parameters are ignored in this case.

This API is called after successful connection and before putting Wi-Fi SoC in IEEE power save mode.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

uint16_t wlan_get_beacon_period (void)

Use this API to get the beacon period of associated BSS from the cached state information.

Returns

beacon_period if operation is successful.

0 if command fails.

uint8_t wlan_get_dtim_period (void)

Use this API to get the dtim period of associated BSS. When this API called, the radio sends a probe request to the AP for this information.

Returns

dtim_period if operation is successful.

0 if DTIM IE is not found in AP's Probe response.

Note

This API should not be called from Wi-Fi event handler registered by application during wlan_start.

int wlan_get_data_rate (wlan_ds_rate * *ds_rate*, mlan_bss_type *bss_type*)

Use this API to get the current TX and RX rates along with bandwidth and guard interval information if rate is 802.11n.

Parameters

Note

If rate is greater than 11 then it is 802.11n rate and from 12 MCS0 rate starts. The bandwidth mapping is like value 0 is for 20MHz, 1 is 40MHz, 2 is for 80MHz. The guard interval value zero means Long otherwise Short.

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_get_pmfcfg (uint8_t * *mfpc*, uint8_t * *mfpr*)

Use this API to get the management frame protection parameters for sta.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_uap_get_pmfcfg (uint8_t * *mfpc*, uint8_t * *mfpr*)

Use this API to get the set management frame protection parameters for uAP.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_set_packet_filters (wlan_flt_cfg_t * *flt_cfg*)

Use this API to set packet filters in Wi-Fi firmware.

Parameters

Note

For example:

MEF Configuration command

mefcfg={

Criteria: bit0-broadcast, bit1-unicast, bit3-multicast

Criteria=2 Unicast frames are received during host sleep mode

NumEntries=1 Number of activated MEF entries

mef_entry_0: example filters to match TCP destination port 80 send by 192.168.0.88 pkt or magic pkt.

mef_entry_0={

mode: bit0–hostsleep mode, bit1–non hostsleep mode

mode=1 HostSleep mode

action: 0–discard and not wake host, 1–discard and wake host 3–allow and wake host

action=3 Allow and Wake host

filter_num=3 Number of filter

RPN only support “&&” and “||” operators, space cannot be removed between operators.

RPN=Filter_0 && Filter_1 || Filter_2
Byte comparison filter's type is 0x41, decimal comparison filter's type is 0x42,
Bit comparison filter's type is 0x43
Filter_0 is decimal comparison filter, it always with type=0x42
Decimal filter always has type, pattern, offset, numbyte 4 field
Filter_0 matches RX packet with TCP destination port 80
Filter_0={
type=0x42 decimal comparison filter
pattern=80 80 is the decimal constant to be compared
offset=44 44 is the byte offset of the field in RX pkt to be compare
numbyte=2 2 is the number of bytes of the field
}
Filter_1 is Byte comparison filter, it always with type=0x41
Byte filter always has type, byte, repeat, offset 4 filed
Filter_1 matches RX packet send by IP address 192.168.0.88
Filter_1={
type=0x41 Byte comparison filter
repeat=1 1 copies of 'c0:a8:00:58'
byte=c0:a8:00:58 'c0:a8:00:58' is the byte sequence constant with each byte
in hex format, with ':' as delimiter between two byte.
offset=34 34 is the byte offset of the equal length field of rx'd pkt.
}
Filter_2 is Magic packet, it can look for 16 contiguous copies of '00:50:43:20:01:02' from the RX pkt's offset 14
Filter_2={
type=0x41 Byte comparison filter
repeat=16 16 copies of '00:50:43:20:01:02'
byte=00:50:43:20:01:02 # '00:50:43:20:01:02' is the byte sequence constant
offset=14 14 is the byte offset of the equal length field of rx'd pkt.
}
}
}
Above filters can be set by filling values in following way in wlan_flt_cfg_t structure.
wlan_flt_cfg_t flt_cfg;
uint8_t byte_seq1[] = {0xc0, 0xa8, 0x00, 0x58};
uint8_t byte_seq2[] = {0x00, 0x50, 0x43, 0x20, 0x01, 0x02};
memset(&flt_cfg, 0, sizeof(wlan_flt_cfg_t));
flt_cfg.criteria = 2;
flt_cfg.nentries = 1;

flt_cfg.mef_entry.mode = 1;

flt_cfg.mef_entry.action = 3;

flt_cfg.mef_entry.filter_num = 3;

flt_cfg.mef_entry.filter_item[0].type = TYPE_DNUM_EQ;

flt_cfg.mef_entry.filter_item[0].pattern = 80;

flt_cfg.mef_entry.filter_item[0].offset = 44;

flt_cfg.mef_entry.filter_item[0].num_bytes = 2;

flt_cfg.mef_entry.filter_item[1].type = TYPE_BYTE_EQ;

flt_cfg.mef_entry.filter_item[1].repeat = 1;

flt_cfg.mef_entry.filter_item[1].offset = 34;

flt_cfg.mef_entry.filter_item[1].num_byte_seq = 4;

memcpy(flt_cfg.mef_entry.filter_item[1].byte_seq, byte_seq1, 4);

flt_cfg.mef_entry.rpn[1] = RPN_TYPE_AND;

flt_cfg.mef_entry.filter_item[2].type = TYPE_BYTE_EQ;

flt_cfg.mef_entry.filter_item[2].repeat = 16;

flt_cfg.mef_entry.filter_item[2].offset = 14;

flt_cfg.mef_entry.filter_item[2].num_byte_seq = 6;

memcpy(flt_cfg.mef_entry.filter_item[2].byte_seq, byte_seq2, 6);

flt_cfg.mef_entry.rpn[2] = RPN_TYPE_OR;

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_wowlan_cfg_ptn_match (enum wlan_bss_type *bss_type*, wlan_wowlan_ptn_cfg_t * *ptn_cfg*)

Use this API to enable WOWLAN (wake-on-wireless-LAN) on magic packet RX in Wi-Fi firmware

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails

void wlan_hs_pre_cfg (void)

Use this API to set configuration before going to host sleep

void wlan_hs_post_cfg (void)

Use this API to get and print the reason of waking up from host sleep

int wlan_get_wakeup_reason (uint16_t * *hs_wakeup_reason*)

Use this API to get host sleep wakeup reason from Wi-Fi firmware after waking up from host sleep by Wi-Fi.

Parameters 1. Non-maskable event matched 6: Non-maskable condition matched (EAPoL rekey) 7: Magic pattern matched Others: reserved. (set to 0)

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_get_current_bssid (uint8_t * *bssid*)

Use this API to get the BSSID of associated BSS when in station mode.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

uint8_t wlan_get_current_channel (void)

Use this API to get the channel number of associated BSS.

Returns

channel number if operation is successful.

0 if command fails.

int wlan_get_log (wlan_pkt_stats_t * *stats*)

Use this API to get the various statistics of STA from Wi-Fi firmware

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_uap_get_log (wlan_pkt_stats_t * *stats*)

Use this API to get the various statistics of the uAP from Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_get_stats (wlan_stats_t * *stats*, enum wlan_bss_type *bss_type*)

Use this API to get the various statistics of STA/uAP from Wi-Fi driver

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_reset_stats (enum wlan_bss_type *bss_type*)

Use this API to reset the various statistics of STA/uAP from Wi-Fi driver

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_get_ps_mode (enum wlan_ps_mode * *ps_mode*)

Get station interface power save mode.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *ps_mode* was NULL.

int wlan_get_ps_mode_cfg (uint8_t * *ps_mode_cfg*)

Get station interface power save configuration.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_INVAL if *ps_mode_cfg* was NULL.

int wlan_wlcmgr_send_msg (enum wifi_event *event*, enum wifi_event_reason *reason*, void * *data*)

Send message to Wi-Fi connection manager thread.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if failed.

int wlan_wfa_basic_cli_init (void)

Register WFA basic Wi-Fi CLI (command line input) commands

This function registers basic Wi-Fi CLI commands like showing version information, MAC address.

Note

This function can only be called by the application after wlan_init() called.

Returns

WLAN_ERROR_NONE if the CLI commands were registered or

WLAN_ERROR_ACTION if they were not registered (for example if this function was called while the CLI commands were already registered).

int wlan_wfa_basic_cli_deinit (void)

Unregister WFA basic Wi-Fi CLI (command line input) commands

This function unregisters basic Wi-Fi CLI commands like showing version information, MAC address.

Note

This function can only be called by the application after wlan_init() called.

Returns

WLAN_ERROR_NONE if the CLI commands were unregistered or

WLAN_ERROR_ACTION if they were not unregistered

int wlan_basic_cli_init (void)

Register basic Wi-Fi CLI (command line input) commands

This function registers basic Wi-Fi CLI commands like showing version information, MAC address.

Note

This function can only be called by the application after wlan_init() called.

This function gets called by wlan_cli_init(), hence only one function out of these two functions should be called in the application.

Returns

WLAN_ERROR_NONE if the CLI commands were registered

WLAN_ERROR_ACTION if they were not registered (for example if this function was called while the CLI commands were already registered).

int wlan_basic_cli_deinit (void)

Unregister basic Wi-Fi CLI commands

This function unregisters basic Wi-Fi CLI commands like showing version information, MAC address.

Note

This function gets called by wlan_cli_deinit(), hence only one function out of these two functions should be called in the application.

Returns

WLAN_ERROR_NONE if the CLI commands were unregistered

WLAN_ERROR_ACTION if they were not unregistered (for example if this function was called while the CLI commands were not registered or were already unregistered).

int wlan_cli_init (void)

Register Wi-Fi CLI (command line input) commands.

Try to register the Wi-Fi CLI commands with the CLI subsystem. This function is available for the application for use.

Note

This function can only be called by the application after wlan_init() called.

This function internally calls wlan_basic_cli_init(), hence only one function out of these two functions should be called in the application.

Returns

WM_SUCCESS if the CLI commands were registered or

-WM_FAIL if they were not (for example if this function was called while the CLI commands were already registered).

int wlan_cli_deinit (void)

Unregister Wi-Fi CLI commands.

Try to unregister the Wi-Fi CLI commands with the CLI subsystem. This function is available for the application for use.

Note

This function can only be called by the application after wlan_init() called.

This function internally calls wlan_basic_cli_deinit(), hence only one function out of these two functions should be called in the application.

Returns

WM_SUCCESS if the CLI commands were unregistered or

-WM_FAIL if they were not (for example if this function was called while the CLI commands were already unregistered).

int wlan_enhanced_cli_init (void)

Register Wi-Fi enhanced CLI commands.

Register the Wi-Fi enhanced CLI commands like set or get tx-power, tx-datarate, tx-modulation etc. with the CLI subsystem.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands were registered or

-WM_FAIL if they were not (for example if this function was called while the CLI commands were already registered).

int wlan_enhanced_cli_deinit (void)

Unregister Wi-Fi enhanced CLI commands.

Unregister the Wi-Fi enhanced CLI commands like set or get tx-power, tx-datarate, tx-modulation etc. with the CLI subsystem.

Note

This function can only be called by the application after wlan_init() called.

Returns

WM_SUCCESS if the CLI commands were unregistered or

-WM_FAIL if they were not unregistered.

unsigned int wlan_get_uap_supported_max_clients (void)

Get maximum number of the stations Wi-Fi firmware supported that can be allowed to connect to the uAP.

Returns

Maximum number of the stations Wi-Fi firmware supported that can be allowed to connect to the uAP.

Note

Get operation is allowed in any uAP state.

int wlan_get_uap_max_clients (unsigned int * *max_sta_num*)

Get current maximum number of the stations that can be allowed to connect to the uAP.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

Note

Get operation is allowed in any uAP state.

int wlan_set_uap_max_clients (unsigned int *max_sta_num*)

Set maximum number of the stations that can be allowed to connect to the uAP.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

Note

Set operation in not allowed in WLAN_UAP_STARTED state.

int wlan_set_htcapinfo (unsigned int *htcapinfo*)

Use this API to configure some of parameters in HT capability information IE (such as short GI, channel bandwidth, and green field support)

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_set_httxcfg (unsigned short *httxcfg*)

Use this API to configure various 802.11n specific configuration for transmit (such as short GI, channel bandwidth and green field support)

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_set_txratecfg (wlan_ds_rate *ds_rate*, mlan_bss_type *bss_type*)

Use this API to set the transmit data rate.

Note

The data rate can be set only after association.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_get_txratecfg (wlan_ds_rate * *ds_rate*, mlan_bss_type *bss_type*)

Use this API to get the transmit data rate.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_get_sta_tx_power (t_u32 * *power_level*)

Get station transmit power

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_set_sta_tx_power (t_u32 *power_level*)

Set station transmit power

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_set_wwsm_txpwrlimit (void)

Set worldwide safe mode TX power limits. Set TX power limit and ru TX power limit according to the region code. TX power limit: rg_power_cfg_info ru TX power limit: ru_power_cfg_info

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_get_mgmt_ie (enum wlan_bss_type *bss_type*, IEEEtypes_ElementId_t *index*, void * *buf*, unsigned int * *buf_len*)

Get Management IE for given BSS type (interface) and index.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

int wlan_set_mgmt_ie (enum wlan_bss_type *bss_type*, IEEEtypes_ElementId_t *id*, void * *buf*, unsigned int *buf_len*)

Set management IE for given BSS type (interface) and index.

Parameters

Returns

Management IE index if successful.

-WM_FAIL if unsuccessful.

int wlan_clear_mgmt_ie (enum wlan_bss_type *bss_type*, IEEEtypes_ElementId_t *index*, int *mgmt_bitmap_index*)

Clear management IE for given BSS type (interface) and index.

Parameters

Returns

WM_SUCCESS if successful.

-WM_FAIL if unsuccessful.

bool wlan_get_11d_enable_status (void)

Get current status of 802.11d support.

Returns

true if 802.11d support is enabled by application.

false if not enabled.

int wlan_get_current_signal_strength (short * *rssi*, int * *snr*)

Get current RSSI and signal to noise ratio from Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if successful.

int wlan_get_average_signal_strength (short * *rssi*, int * *snr*)

Get average RSSI and signal to noise ratio (average value of the former 8 packets) from Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if successful.

int wlan_remain_on_channel (const enum wlan_bss_type *bss_type*, const bool *status*, const uint8_t *channel*, const uint32_t *duration*)

This API is used to set/cancel the remain on channel configuration.

Note

When status is false, channel and duration parameters are ignored.

Parameters

Returns

WM_SUCCESS on success or error code.

int wlan_get_otp_user_data (uint8_t * *buf*, uint16_t *len*)

Get user data from OTP (one-time pramming) memory

Parameters

Returns

WM_SUCCESS if user data read operation is successful.

-WM_E_INVAL if buf is not valid or of insufficient size.

-WM_FAIL if user data field is not present or command fails.

int wlan_get_cal_data (wlan_cal_data_t * *cal_data*)

Get calibration data from Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if calibration data read operation is successful.

-WM_E_INVAL if cal_data is not valid.

-WM_FAIL if command fails.

Note

The user of this API should free the allocated buffer for calibration data.

int wlan_set_region_power_cfg (const t_u8 * *data*, t_u16 *len*)

Set the compressed (use LZW algorithm) TX power limit configuration.

Parameters

Returns

WM_SUCCESS on success, error otherwise.

int wlan_set_chanlist_and_txpwrlimit (wlan_chanlist_t * *chanlist*, wlan_txpwrlimit_t * *txpwrlimit*)

Set the TRPC (transient receptor potential canonical) channel list and TX power limit configuration.

Parameters

Returns

WM_SUCCESS on success, error otherwise.

int wlan_set_chanlist (wlan_chanlist_t * *chanlist*)

Set the channel list configuration wlan_chanlist_t.

Parameters

Returns

WM_SUCCESS on success, error otherwise.

Note

If region enforcement flag is enabled in the OTP then this API should not take effect.

int wlan_get_chanlist (wlan_chanlist_t * *chanlist*)

Get the channel list configuration.

Parameters

Returns

WM_SUCCESS on success, error otherwise.

Note

The wlan_chanlist_t struct allocates memory for a maximum of 54. channels.

int wlan_set_txpwrlimit (wlan_txpwrlimit_t * *txpwrlimit*)

Set the TRPC (transient receptor potential canonical) channel configuration.

Parameters

Returns

WM_SUCCESS on success, error otherwise.

int wlan_get_txpwrlimit (wifi_SubBand_t *subband*, wifi_txpwrlimit_t * *txpwrlimit*)

Get the TRPC (transient receptor potential canonical) channel configuration.

Parameters

Returns

WM_SUCCESS on success, error otherwise.

Note

application can use print_txpwrlimit API to print the content of the txpwrlimit structure.

void wlan_set_reassoc_control (bool *reassoc_control*)

Set reassociation control in Wi-Fi connection manager. When reassociation control enabled, Wi-Fi connection manager attempts reconnection with the network for WLAN_RECONNECT_LIMIT times before giving up.

Note

Reassociation is enabled by default in the Wi-Fi connection manager.

Parameters

void wlan_uap_set_beacon_period (const uint16_t *beacon_period*)

API to set the beacon period of the uAP

Parameters

Note

Call this API before calling uAP start API.

int wlan_uap_set_bandwidth (const uint8_t *bandwidth*)

API to set the bandwidth of the uAP

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

-WM_FAIL if command fails.

Note

Not applicable to 20MHZ only chip sets (Redfinch, SD8801)

Call this API before calling uAP start API.

Default bandwidth setting is 40 MHz.

int wlan_uap_get_bandwidth (uint8_t * *bandwidth*)

API to get the bandwidth of the uAP

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

-WM_FAIL if command fails.

Note

Call this API before calling uAP start API.

int wlan_uap_set_hidden_ssid (const t_u8 *hidden_ssid*)

API to control SSID broadcast capability of the uAP

This API enables/disables the SSID broadcast feature (also known as the hidden SSID feature). When broadcast SSID is enabled, the AP responds to probe requests from client stations that contain null SSID. When broadcast SSID is disabled, the AP does not respond to probe requests that contain null SSID and generates beacons that contain null SSID.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

-WM_FAIL if command fails.

Note

Call this API before calling uAP start API.

void wlan_uap_ctrl_deauth (const bool *enable*)

API to control the deauthentication during uAP channel switch.

Parameters

Note

Call this API before calling uAP start API.

void wlan_uap_set_ecsa (void)

API to enable channel switch announcement functionality on uAP.

Note

Call this API before calling uAP start API. Also note that 802.11n should be enabled on uAP. The channel switch announcement IE is transmitted in 7 beacons before the channel switch, during a station connection attempt on a different channel with Ex-AP.

void wlan_uap_set_htcapinfo (const uint16_t *ht_cap_info*)

API to set the HT capability information of the uAP.

Parameters

Note

Call this API before calling uAP start API.

void wlan_uap_set_httxcfg (unsigned short *httxcfg*)

This API can be used to configure various 802.11n specific configuration for transmit (such as short GI, channel bandwidth and green field support) for uAP interface.

Parameters

Note

Call this API before calling uAP start API.

void wlan_sta_ampdu_tx_enable (void)

This API can be used to enable AMPDU support when station is a transmitter.

Note

By default the station AMPDU TX support is enabled if configuration option CONFIG_STA_AMPDU_TX is defined 1.

void wlan_sta_ampdu_tx_disable (void)

This API can be used to disable AMPDU support when station is a transmitter.

Note

By default the station AMPDU TX support is enabled if configuration option CONFIG_STA_AMPDU_TX is defined 1.

void wlan_sta_ampdu_rx_enable (void)

This API can be used to enable AMPDU support when station is a receiver.

Note

By default the station AMPDU RX support is enabled if configuration option CONFIG_STA_AMPDU_RX is defined 1.

void wlan_sta_ampdu_rx_disable (void)

This API can be used to disable AMPDU support when station is a receiver.

Note

By default the station AMPDU RX support is enabled if configuration option CONFIG_STA_AMPDU_RX is defined 1.

void wlan_uap_ampdu_tx_enable (void)

This API can be used to enable AMPDU support when uAP is a transmitter.

Note

By default the uAP AMPDU TX support is enabled if configuration option CONFIG_UAP_AMPDU_TX is defined 1.

void wlan_uap_ampdu_tx_disable (void)

This API can be used to disable AMPDU support when uAP is a transmitter.

Note

By default the uAP AMPDU TX support is enabled if configuration option CONFIG_UAP_AMPDU_TX is defined 1.

void wlan_uap_ampdu_rx_enable (void)

This API can be used to enable AMPDU support when uAP is a receiver.

Note

By default the uAP AMPDU TX support is enabled if configuration option CONFIG_UAP_AMPDU_RX is defined 1.

void wlan_uap_ampdu_rx_disable (void)

This API can be used to disable AMPDU support when uAP is a receiver.

Note

By default the uAP AMPDU TX support is enabled if configuration option CONFIG_UAP_AMPDU_RX is defined 1.

void wlan_uap_set_scan_chan_list (wifi_scan_chan_list_t *scan_chan_list*)

Set number of channels and channel number used during automatic channel selection of the uAP.

Parameters

Note

Call this API before uAP start API in order to set the user defined channels, otherwise it can have no effect. There is no need to call this API every time before uAP start, if once set same channel configuration can get used in all upcoming uAP start call. If user wish to change the channels at run time then it make sense to call this API before every uAP start API.

int wlan_set_rts (int *rts*)

Set the RTS(Request to Send) threshold of STA in Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_uap_rts (int *rts*)

Set the RTS(Request to Send) threshold of the uAP in Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_frag (int *frag*)

Set the fragment threshold of STA in Wi-Fi firmware. If the size of packet exceeds the fragment threshold, the packet is divided into fragments. For example, if the fragment threshold is set to 300, a ping packet of size 1300 is divided into 5 fragments.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_uap_frag (int *frag*)

Set the fragment threshold of the uAP in Wi-Fi firmware. If the size of packet exceeds the fragment threshold, the packet is divided into fragments. For example, if the fragment threshold is set to 300, a ping packet of size 1300 is divided into 5 fragments.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_sta_mac_filter (int *filter_mode*, int *mac_count*, unsigned char * *mac_addr*)

Set the STA MAC filter in Wi-Fi firmware. Apply for uAP mode only. When STA MAC filter enabled, wlan firmware blocks all the packets from station with MAC address in black list and not blocks packets from station with MAC address in white list.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_crypto_RC4_encrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *KeyIV*, const t_u16 *KeyIVLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto RC4 (rivest cipher 4) algorithm encrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The length of the encrypted data is the same as the origin DataLength.

int wlan_set_crypto_RC4_decrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *KeyIV*, const t_u16 *KeyIVLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto RC4 (rivest cipher 4) algorithm decrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The length of the decrypted data is the same as the origin DataLength.

int wlan_set_crypto_AES_ECB_encrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *KeyIV*, const t_u16 *KeyIVLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_ECB (advanced encryption standard, electronic codebook) algorithm encrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The length of the encrypted data is the same as the origin DataLength.

int wlan_set_crypto_AES_ECB_decrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *KeyIV*, const t_u16 *KeyIVLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_ECB (advanced encryption standard, electronic codebook) algorithm decrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The length of the decrypted data is the same as the origin DataLength.

int wlan_set_crypto_AES_WRAP_encrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *KeyIV*, const t_u16 *KeyIVLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_WRAP (advanced encryption standard wrap) algorithm encrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The encrypted data is 8 bytes more than the original data. Therefore, the address pointed to by Data needs to reserve enough space.

int wlan_set_crypto_AES_WRAP_decrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *KeyIV*, const t_u16 *KeyIVLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_WRAP algorithm decrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The decrypted data is 8 bytes less than the original data.

int wlan_set_crypto_AES_CCMP_encrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *AAD*, const t_u16 *AADLength*, const t_u8 * *Nonce*, const t_u16 *NonceLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_CCMP (counter mode with cipher block chaining message authentication code protocol) algorithm encrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The encrypted data is 8 bytes (when key length is 16) or 16 bytes (when key length is 32) more than the original data. Therefore, the address pointed to by Data needs to reserve enough space.

int wlan_set_crypto_AES_CCMP_decrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *AAD*, const t_u16 *AADLength*, const t_u8 * *Nonce*, const t_u16 *NonceLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_CCMP algorithm decrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The decrypted data is 8 bytes (when key length is 16) or 16 bytes (when key length is 32) less than the original data.

int wlan_set_crypto_AES_GCMP_encrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *AAD*, const t_u16 *AADLength*, const t_u8 * *Nonce*, const t_u16 *NonceLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_GCMP (galois/counter mode with AES-GMAC) algorithm encrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the encrypted data. The value of DataLength is updated to the encrypted data length. The encrypted data is 16 bytes more than the original data. Therefore, the address pointed to by Data needs to reserve enough space.

int wlan_set_crypto_AES_GCMP_decrypt (const t_u8 * *Key*, const t_u16 *KeyLength*, const t_u8 * *AAD*, const t_u16 *AADLength*, const t_u8 * *Nonce*, const t_u16 *NonceLength*, t_u8 * *Data*, t_u16 * *DataLength*)

Set crypto AES_CCMP algorithm decrypt command parameters.

Parameters

Returns

WM_SUCCESS if successful.

-WM_E_PERM if not supported.

-WM_FAIL if failure.

Note

If the function returns WM_SUCCESS, the data in the memory pointed to by data is overwritten by the decrypted data. The value of DataLength is updated to the decrypted data length. The decrypted data is 16 bytes less than the original data.

int wlan_enable_disable_htc (uint8_t *option*)

This function is used to enable/disable HTC (high throughput control).

Parameters

Returns

WM_SUCCESS if operation is successful, otherwise return -WM_FAIL

int wlan_set_11ax_tx_omi (const t_u8 *interface*, const t_u16 *tx_omi*, const t_u8 *tx_option*, const t_u8 *num_data_pkts*)

Use this API to set the set 802.11ax TX OMI (operating mode indication).

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_set_11ax_tol_time (const t_u32 *tol_time*)

Set 802.11ax OBSS (overlapping basic service set) narrow bandwidth RU (resource unit) tolerance time In uplink transmission, AP sends a trigger frame to all the stations that can be involved in the upcoming transmission, and then these stations transmit Trigger-based(TB) PPDU in response to the trigger frame. If STA connects to AP which channel is set to 100,STA doesn't support 26 tones RU. The API should be called when station is in disconnected state.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_11ax_rutxpowerlimit (const void * *rutx_pwr_cfg*, uint32_t *rutx_pwr_cfg_len*)

Use this API to set the RU TX power limit.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_set_11ax_rutxpowerlimit_legacy (const wlan_rutxpwrlimit_t * *ru_pwr_cfg*)

Use this API to set the RU TX power limit by channel based approach.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_get_11ax_rutxpowerlimit_legacy (wlan_rutxpwrlimit_t * *ru_pwr_cfg*)

Use this API to get the RU TX power limit by channel based approach.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_set_11ax_cfg (wlan_11ax_config_t * *ax_config*)

Set 802.11ax configuration parameters

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

wlan_11ax_config_t* wlan_get_11ax_cfg (void)

Get default 802.11ax configuration parameters

Returns

802.11ax configuration parameters default array.

int wlan_set_btwt_cfg (wlan_btwt_config_t * *btwt_cfg*)

Set broadcast TWT (target wake time) configuration parameters

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_btwt_cfg (wlan_btwt_config_t * *btwt_cfg*)

Get broadcast TWT (target wake time) configuration parameters

Parameters

Returns

WM_SUCCESS if successful otherwise failure.

int wlan_set_twt_setup_cfg (const wlan_twt_setup_config_t * *twt_setup*)

Set TWT setup configuration parameters

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

wlan_twt_setup_config_t* wlan_get_twt_setup_cfg (void)

Get TWT setup configuration parameters

Returns

TWT setup parameters default array.

int wlan_set_twt_teardown_cfg (const wlan_twt_teardown_config_t * *teardown_config*)

Set TWT teardown configuration parameters

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

wlan_twt_teardown_config_t* wlan_get_twt_teardown_cfg (void)

Get TWT teardown configuration parameters

Returns

TWT Teardown parameters default array

int wlan_get_twt_report (wlan_twt_report_t * *twt_report*)

Get TWT report

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_twt_information (wlan_twt_information_t * *twt_information*)

Twt information

Parameters

Returns

WM_SUCCESS if successful otherwise failure.

int wlan_set_mmsf (const t_u8 *enable*, const t_u8 *Density*, const t_u8 *MMSF*)

Set 802.11ax AMPDU (aggregate medium access control (MAC) protocol data unit) density configuration.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_mmsf (t_u8 * *enable*, t_u8 * *Density*, t_u8 * *MMSF*)

Get 802.11ax AMPDU density configuration.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_channel_load (wlan_802_11_chan_load_t * *chan_load*)

Set Wi-Fi channel load info.

int wlan_get_channel_load (wlan_802_11_chan_load_t * *chan_load*)

Get Wi-Fi channel load info.

int wlan_set_clocksync_cfg (const wlan_clock_sync_gpio_tsf_t * *tsf_latch*)

Set clock sync GPIO based TSF (time synchronization function).

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_tsf_info (wlan_tsf_info_t * *tsf_info*)

Get TSF info from firmware using GPIO latch.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_ft_roam (const t_u8 * *bssid*, const t_u8 *channel*)

Start FT roaming : This API is used to initiate fast BSS transition based roaming.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_rx_mgmt_indication (const enum wlan_bss_type *bss_type*, const uint32_t *mgmt_subtype_mask*, int(*)(const enum wlan_bss_type bss_type, const wlan_mgmt_frame_t *frame, const size_t len) *rx_mgmt_callback*)

This API can be used to start/stop the management frame forwarded to host through data path.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

Note

Pass management subtype mask all zero to disable all the management frame forward to host.

void wlan_set_scan_channel_gap (unsigned *scan_chan_gap*)

Set scan channel gap.

Parameters

int wlan_host_11k_cfg (int *enable_11k*)

Enable/Disable host 802.11k feature.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

bool wlan_get_host_11k_status (void)

Get enable/disable host 802.11k feature flag.

Returns

TRUE if 802.11k is enabled, return FALSE if 802.11k is disabled.

int wlan_host_11k_neighbor_req (const char * *ssid*)

Host send neighbor report request.

Parameters

Note

ssid parameter is optional, pass NULL pointer to ignore SSID input if not specify SSID

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_host_11v_bss_trans_query (t_u8 *query_reason*)

Host send BSS transition management query. STA sends BTM (BSS transition management) query, and the AP supporting 11V will response BTM request, the AP will parse neighbor report in the BTM request and response the BTM response to AP to indicate the receive status.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_mbo_peferch_cfg (const char * *non_pref_chan*)

Multi band operation (MBO) non-preferred channels

A space delimited list of non-preferred channels where each channel is a colon delimited list of values.

Format:

non_pref_chan=oper_class:chan:preference:reason Example:

non_pref_chan=81:5:10:2 81:1:0:2 81:9:0:2

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_mbo_set_cell_capa (t_u8 *cell_capa*)

MBO set cellular data capabilities

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_mbo_set_oce (t_u8 *oce*)

Optimized connectivity experience (OCE)

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_okc (t_u8 *okc*)

Opportunistic key caching (also known as proactive key caching) default This parameter can be used to set the default behavior for the proactive_key_caching parameter. By default, OKC is disabled unless enabled with the global okc=1 parameter or with the per-network pkc(proactive_key_caching)=1 parameter. With okc=1, OKC is enabled by default, but can be disabled with per-network pkc(proactive_key_caching)=0 parameter.

Parameters

0 = Disable OKC (default) 1 = Enable OKC

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_pmksa_list (char * *buf*, size_t *buflen*)

Dump text list of entries in PMKSA (pairwise master key security association) cache.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_pmksa_flush (void)

Flush PTKSA cache entries

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_scan_interval (int *scan_int*)

Set wpa supplicant scan interval in seconds

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_uap_set_ecsa_cfg (t_u8 *block_tx*, t_u8 *oper_class*, t_u8 *channel*, t_u8 *switch_count*, t_u8 *band_width*)

Send the ecsa configuration parameter to FW.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_subscribe_event (unsigned int *event_id*, unsigned int *thresh_value*, unsigned int *freq*)

Subscribe specified event from the Wi-Fi firmware. Wi-Fi firmware report the registered event to driver upon configured report conditions are met.

Parameters

Returns

WM_SUCCESS if set successfully, otherwise return failure.

int wlan_get_subscribe_event (wlan_ds_subscribe_evt * *sub_evt*)

Get all subscribed events from Wi-Fi firmware along with threshold value and report frequency.

Parameters

Returns

WM_SUCCESS if set successfully, otherwise return failure.

int wlan_clear_subscribe_event (unsigned int *event_id*)

cancel the subscribe event to firmware

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_threshold_link_quality (unsigned int *evend_id*, unsigned int *link_snr*, unsigned int *link_snr_freq*, unsigned int *link_rate*, unsigned int *link_rate_freq*, unsigned int *link_tx_latency*, unsigned int *link_tx_lantency_freq*)

subscribe link quality event

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_tsp_cfg (t_u16 * *enable*, t_u32 * *back_off*, t_u32 * *highThreshold*, t_u32 * *lowThreshold*, t_u32 * *dutycycstep*, t_u32 * *dutycycmin*, int * *highthrtemp*, int * *lowthrtemp*, int * *currCAUTemp*, int * *currRFUTemp*)

Get TSP (thermal safeguard protection) configuration. TSP algorithm monitors PA Tj and primarily backs off data throughput.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_tsp_cfg (t_u16 *enable*, t_u32 *back_off*, t_u32 *highThreshold*, t_u32 *lowThreshold*, t_u32 *dutycycstep*, t_u32 *dutycycmin*, int *highthrtemp*, int *lowthrtemp*)

Set TSP (thermal safeguard protection) configuration. TSP algorithm monitors and primarily backs off data throughput.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_reg_access (wifi_reg_t *type*, uint16_t *action*, uint32_t *offset*, uint32_t * *value*)

This function reads/writes adapter registers value.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_tx_ampdu_prot_mode (tx_ampdu_prot_mode_para * *prot_mode*, t_u16 *action*)

Set/Get TX AMPDU protect mode.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_mef_set_auto_arp (t_u8 *mef_action*)

This function set auto ARP configuration.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_mef_set_auto_ping (t_u8 *mef_action*)

This function set auto ping configuration.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_mef_set_multicast (t_u8 *mef_action*)

This function set multicast packet as low power wake up condition.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_config_mef (int *type*, t_u8 *mef_action*)

This function set/delete MEF entries configuration.

Parameters

Returns

WM_SUCCESS if the call was successful.

-WM_FAIL if failed.

int wlan_set_ipv6_ns_mef (t_u8 *mef_action*)

Use this API to enable IPv6 neighbor solicitation offload in Wi-Fi firmware.

Parameters

Returns

WM_SUCCESS if operation is successful.

-WM_FAIL if command fails.

int wlan_csi_cfg (wlan_csi_config_params_t * *csi_params*)

Send the CSI configuration parameter to firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_register_csi_user_callback (int(*)(void *buffer, size_t len) *csi_data_recv_callback*)

This function registers callback which are used to deliver CSI (channel state information) data to user.

Parameters Memory layout of buffer:

size(byte) items

2 buffer len[bit 0:12]

2 CSI signature, 0xABCD fixed

4 User defined HeaderID

2 Packet info

2 Frame control field for the received packet

8 Timestamp when packet received

6 Received packet destination MAC Address

6 Received packet source MAC address

1 RSSI for antenna A

1 RSSI for antenna B

1 Noise floor for antenna A

1 Noise floor for antenna B

1 RX signal strength above noise floor

1 Channel

2 user defined chip ID

4 Reserved

4 CSI data length in DWORDs

CSI data

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_unregister_csi_user_callback (void)

This function unregisters callback which are used to deliver CSI data to user.

Returns

WM_SUCCESS if successful

wlan_csi_config_params_t* wlan_get_csi_cfg_param_default (void)

This function get CSI default configuration data.

Returns

CSI data pointer.

int wlan_set_csi_cfg_param_default (wlan_csi_config_params_t * *in_csi_cfg*)

This function set CSI default configuration data.

Parameters

Returns

if successful return 1 else return 0.

void wlan_reset_csi_filter_data (void)

This function reset Wi-Fi CSI filter data.

void wlan_set_rssi_low_threshold (uint8_t *threshold*)

Use this API to set the RSSI threshold value for low RSSI event subscription. When RSSI falls below this threshold firmware can generate the low RSSI event to driver. This low RSSI event is used when either of CONFIG_11R, CONFIG_11K, CONFIG_11V or CONFIG_ROAMING is enabled.

Note

By default RSSI low threshold is set at -70 dbm.

Parameters

void wlan_wps_generate_pin (uint32_t * *pin*)

This function generate pin for WPS pin session.

Parameters

int wlan_start_wps_pin (const struct netif * *netif*, const char * *pin*)

Start WPS pin session.

This function starts WPS pin session.

Parameters

Returns

WM_SUCCESS if the pin entered is valid.

-WM_FAIL if invalid pin entered.

int wlan_start_wps_pbc (const struct netif * *netif*)

Start WPS PBC (push button configuration) session.

This function starts WPS PBC (push button configuration) session.

Returns

WM_SUCCESS if successful

-WM_FAIL if invalid pin entered.

int wlan_wps_cancel (void)

Cancel WPS session.

This function cancels ongoing WPS session.

Returns

WM_SUCCESS if successful

-WM_FAIL if invalid pin entered.

int wlan_start_ap_wps_pin (const char * *pin*)

Start WPS pin session.

This function starts AP WPS pin session.

Parameters

Returns

WM_SUCCESS if the pin entered is valid.

-WM_FAIL if invalid pin entered.

int wlan_start_ap_wps_pbc (void)

Start WPS PBC session.

This function starts AP WPS PBC session.

Returns

WM_SUCCESS if successful

-WM_FAIL if invalid pin entered.

int wlan_wps_ap_cancel (void)

Cancel AP's WPS session.

This function cancels ongoing WPS session.

Returns

WM_SUCCESS if successful

-WM_FAIL if invalid pin entered.

int wlan_set_entp_cert_files (int *cert_type*, t_u8 * *data*, t_u32 *data_len*)

This function specifies the enterprise certificate file This function is used before adding network profile. It can store certificate data in "wlan" global structure.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

t_u32 wlan_get_entp_cert_files (int *cert_type*, t_u8 ** *data*)

This function get enterprise certificate data from "wlan" global structure

Parameters

Returns

size of raw data

void wlan_free_entp_cert_files (void)

This function free the temporary memory of enterprise certificate data After add new enterprise network profile, the certificate data has been parsed by mbedtls into another data, which can be freed.

int wlan_net_monitor_cfg (wlan_net_monitor_t * *monitor*)

Send the network monitor configuration parameter to firmware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

void wlan_register_monitor_user_callback (int(*)(void *buffer, t_u16 data_len) *monitor_data_recv_callback*)

This function registers callback which are used to deliver monitor data to user.

Parameters

void wlan_deregister_net_monitor_user_callback (void)

This function deregisters monitor callback.

int wlan_mgmtframe_tx_cfg (wlan_host_tx_frame_params_t * *mgmtframe*)

Send the mgmt/data frame config parameter and payload to FW.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

uint8_t wlan_check_11n_capa (unsigned int *channel*)

Check if Wi-Fi hardware support 802.11n for on 2.4G or 5G bands.

Parameters

Returns

true if 802.11n is supported or false if not.

uint8_t wlan_check_11ac_capa (unsigned int *channel*)

Check if Wi-Fi hardware support 802.11ac for on 2.4G or 5G bands.

Parameters

Returns

true if 802.11ac is supported or false if not.

uint8_t wlan_check_11ax_capa (unsigned int *channel*)

Check if Wi-Fi hardware support 802.11ax for on 2.4G or 5G bands.

Parameters

Returns

true if 802.11ax is supported or false if not.

int wlan_set_ips (int *option*)

Config IEEE power save mode (IPS). If the option is 1, the IPS hardware listens to beacon frames after Wi-Fi CPU enters power save mode. When there is work needed to done by Wi-Fi CPU, Wi-Fi CPU can be woken up by ips hardware.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_signal_info (wlan_rssi_info_t * *signal*)

Get RSSI information.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_bandcfg (wlan_bandcfg_t * *bandcfg*)

Set band configuration.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_get_bandcfg (wlan_bandcfg_t * *bandcfg*)

Get band configuration.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_rg_power_cfg (t_u16 *region_code*)

Set TX power table according to region code

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_ru_power_cfg (t_u16 *region_code*)

set ru tx power table

Parameters

Returns

WM_SUCCESS if successful otherwise failure.

void wlan_set_ps_cfg (t_u16 *multiple_dtims*, t_u16 *bcn_miss_timeout*, t_u16 *local_listen_interval*, t_u16 *adhoc_wake_period*, t_u16 *mode*, t_u16 *delay_to_ps*)

Set multiple dtim for next wakeup RX beacon time

Parameters

int wlan_set_country_code (const char * *alpha2*)

Set country code

Note

This API should be called after Wi-Fi is initialized but before starting uAP interface.

Parameters

For the third octet, STA is always 0. for uAP environment: All environments of the current frequency band and country (default) alpha2[2]=0x20 Outdoor environment only alpha2[2]=0x4f Indoor environment only alpha2[2]=0x49 Noncountry entity (country_code=XX) alpha[2]=0x58 IEEE 802.11 standard Annex E table indication: 0x01 .. 0x1f Annex E, Table E-4 (Global operating classes) alpha[2]=0x04

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_country_ie_ignore (uint8_t * *ignore*)

Set ignore region code.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_region_code (unsigned int *region_code*)

Set region code.

Parameters

Returns

WM_SUCCESS if successful otherwise fail.

int wlan_get_region_code (unsigned int * *region_code*)

Get region code.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_set_11d_state (int *bss_type*, int *state*)

Set STA/uAP 802.11d feature Enable/Disable.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_single_ant_duty_cycle (t_u16 *enable*, t_u16 *nbTime*, t_u16 *wlanTime*)

Set single antenna: duty cycle.

Parameters

Note

wlanTime should not equal to wlanTime-nbTime

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dual_ant_duty_cycle (t_u16 *enable*, t_u16 *nbTime*, t_u16 *wlanTime*, t_u16 *wlan-BlockTime*)

Set dual antenna duty cycle.

Parameters

Note

nbTime, wlanTime and wlanBlockTime should not equal to each other

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_external_coex_pta_cfg (ext_coex_pta_cfg *coex_pta_config*)

Set external coex PTA (packet traffic arbitration) parameters.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_configurator_add (int *is_ap*, const char * *cmd*)

Add a DPP (device provisioning protocol) configurator.

If this device is DPP configurator, add it to get configurator ID.

Parameters

Returns

configurator ID if successful otherwise return -WM_FAIL.

void wlan_dpp_configurator_params (int *is_ap*, const char * *cmd*)

Set DPP (device provisioning protocol) configurator parameter

set DPP configurator params. for example:" conf=<sta-dpp/ap-dpp> ssid=<hex ssid> configurator=conf_id" #space character exists between " & conf word.

Parameters

Returns

void

void wlan_dpp_mud_url (int *is_ap*, const char * *cmd*)

MUD URL for enrollee's DPP configuration request (optional)

Wi-Fi_CERTIFIED_Easy_Connect_Test_Plan_v3.0.pdf 5.1.23 STAUT sends the MUD URL

Parameters

Returns

void

int wlan_dpp_bootstrap_gen (int *is_ap*, const char * *cmd*)

Generate QR code.

This function generates QR code and return bootstrap-id

Parameters

Returns

bootstrap-id if successful otherwise return -WM_FAIL.

const char* wlan_dpp_bootstrap_get_uri (int *is_ap*, unsigned int *id*)

Get QR code by bootstrap-id.

This function gets QR code string by bootstrap-id

Parameters

Returns

QR code string if successful otherwise NULL.

int wlan_dpp_qr_code (int *is_ap*, char * *uri*)

Enter the QR code in the DPP device.

This function set the QR code and return qr-code-id.

Parameters

Returns

qr-code-id if successful otherwise return -WM_FAIL.

int wlan_dpp_auth_init (int *is_ap*, const char * *cmd*)

Send provisioning auth request to responder.

This function send Auth request to responder by qr-code-id.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_listen (int *is_ap*, const char * *cmd*)

Make device listen to DPP request.

Responder generates QR code and listening on its operating channel to wait Auth request.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_stop_listen (int *is_ap*)

DPP stop listen.

Stop dpp listen and clear listen frequency

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_pkex_add (int *is_ap*, const char * *cmd*)

Set bootstrapping through PKEX (Public Key Exchange).

Support in-band bootstrapping through PKEX

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_chirp (int *is_ap*, const char * *cmd*)

sends DPP presence announcement.

Send DPP presence announcement from responder. After the Initiator enters the QR-code URI provided by the Responder, the Responder sends the presence announcement to trigger Auth Request from Initiator.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_reconfig (const char * *cmd*)

DPP reconfig.

DPP reconfig and make a new DPP connection.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_dpp_configurator_sign (int *is_ap*, const char * *cmd*)

Configurator configures itself as an Enrollee AP/STA.

Wi-Fi_CERTIFIED_Easy_Connect_Test_Plan_v3.0.pdf 5.3.8 & 5.3.9 Configurator configures itself as an Enrollee AP/STA

for example:" conf=<sta-dpp/ap-dpp> ssid=<hex ssid> configurator=conf_id" #space character exists between " & conf word.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_find (const char * *cmd*)

Initiate P2P discovery.

This function triggers the P2P discovery process by instructing wpa_supplicant to scan for available P2P devices. Optional arguments (such as scan timeout, scan type, or device filters) can be used to tailor the search behavior. Use this function when you want to start discovering nearby P2P devices for later connection or service discovery.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_stop_find (void)

Stop the P2P discovery process.

This command stops an ongoing P2P discovery process initiated by a previous call to wlan_p2p_find. It frees up radio resources and halts further scanning.

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_listen (const char * *cmd*)

Initiates P2P listen mode.

This function sends a generic command to wpa_wpa_supplicant to initiate P2P listen mode.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_connect (char * *cmd*)

Initiate a P2P connection.

After identifying a target P2P device using the discovery process, this function begins the connection sequence. It negotiates connection parameters (often involving WPS configuration) and starts the group formation process.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_group_add (char * *cmd*)

Create a new P2P group.

This function requests the creation of a new P2P group (i.e., starting a group owner instance) in wpa_supplicant. It configures the group parameters, including the SSID and security settings, so that client devices can join.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_get_passphrase (void)

Retrieve the group passphrase.

Once a P2P group has been established, this function prints the WPA-PSK passphrase that secures the group. Client devices can use this passphrase to connect to the group.

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_invite (char * *cmd*)

Issue a group invitation.

This function sends an invitation request to a target P2P device, inviting it to join an existing P2P group. The invitation bypasses the standard negotiation procedure by directly inviting a device.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_prov_disc (char * *cmd*)

Initiate provisioning discovery.

This command starts the provisioning discovery phase, which is used to determine the optimal method (e.g., PIN or PBC) for configuring a new P2P connection as part of the WPS process.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_cancel (void)

Cancel ongoing P2P operations.

This function cancels any active P2P operations, including discovery, connection attempts, or group formation. It resets the P2P state to idle, making it possible to start a new operation afterward.

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_remove_client (char * *cmd*)

Remove a client from the P2P group.

When a P2P group owner needs to disconnect a client, this function removes the specified client from the group. It ensures that the client's association with the group is terminated, and cleans up related state.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_servvice_add (char * *cmd*)

Advertise a service.

This function adds a service advertisement to the device's P2P service discovery framework. It allows the device to broadcast information about services (e.g., file sharing, printing) that may be available to peers.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_serv_disc_req (char * *cmd*)

Send a service discovery request.

A device can use this function to query a discovered P2P peer for details about available services. The request typically includes the type of service or specific query parameters, and the peer is expected to respond with matching service information.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_serv_disc_resp (char * *cmd*)

Send a service discovery response.

This function is used by a P2P device to respond to a service discovery request. It sends detailed information about the services that are available, enabling the requesting peer to decide if the advertised service meets its requirements.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_group_remove (char * *cmd*)

Tear down an existing P2P group.

This function ends an active P2P group by terminating the group owner instance and disconnecting all associated clients. It performs necessary resource cleanup and notifies clients that the group has been disbanded.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_peers (char * *peers_buf*, int *peer_buf_size*, int * *peers_buf_len*)

Retrieves the list of available P2P peers.

This function executes the equivalent of the wpa_cli 'p2p_peers' command. It fills the provided peers_buf with peer information and sets peers_buf_len to reflect the number of bytes written to the buffer.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_peer (char * *cmd*, char * *peer_info_buf*, int *peer_info_buf_size*, int * *peer_info_len*)

Retrieves detailed information for a specified P2P peer.

This function sends a generic wpa_cli command (given by p2p_peer cmd) to obtain detailed information about a specific P2P peer. The resulting output is stored in the buffer provided by peer_info_buf, and the length of the retrieved information is returned via peer_info_len.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_p2p_status (char * *buf*, size_t *buflen*, int * *reslen*)

Retrieves detailed information for a P2P interface.

This function sends a generic wpa_cli command (given by status cmd) to obtain detailed information about a P2P interface. The resulting output is stored in the buffer provided by buf, and the length of the retrieved information is returned via reslen.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_sta_inactivityto (wlan_inactivity_to_t * *inac_to*, t_u16 *action*)

Get/Set inactivity timeout extend

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

t_u16 wlan_get_status_code (enum wlan_event_reason *reason*)

Get 802.11 Status Code.

Parameters

Returns

status code defined in IEEE 802.11-2020 standard.

int32_t wlan_get_temperature (void)

Get board temperature.

Returns

board temperature.

int wlan_auto_null_tx (wlan_auto_null_tx_t * *auto_null_tx*, mlan_bss_type *bss_type*)

Start/Stop auto TX null. Call this API to auto transmit and one shot quality of service data packets to get the CSI after STA connected one AP or uAP was connected with external STA.

Note

STA cannot send auto NULL data if not connected AP, not support auto TX without connecting AP. uAP cannot send auto NULL data if is not connected, not support auto tx without connecting with external STA.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

char* wlan_string_dup (const char * *s*)

Allocate memory for a string and copy the string to the allocated memory

Parameters

Returns

new string if successful, otherwise return -WM_FAIL.

uint32_t wlan_get_board_type (void)

Get board type.

Returns

board type. 0x02: RW610_PACKAGE_TYPE_BGA 0xFF: others

int wlan_uap_disconnect_sta (uint8_t * *sta_addr*)

Disconnect to STA which is connected with internal uAP.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_11n_allowed (struct wlan_network * *network*)

Check if 802.11n is allowed in capability.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_11ac_allowed (struct wlan_network * *network*)

Check if 802.11ac is allowed in capability.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

int wlan_11ax_allowed (struct wlan_network * *network*)

Check if 802.11ax is allowed in capability.

Parameters

Returns

WM_SUCCESS if successful otherwise return -WM_FAIL.

Macro Documentation

#define ACTION_GET (0U)

Action GET

#define ACTION_SET (1)

Action SET

#define IEEEtypes_SSID_SIZE 32U

Maximum SSID length

#define IEEEtypes_ADDRESS_SIZE 6

MAC Address length

#define WLAN_RESCAN_LIMIT CONFIG_MAX_RESCAN_LIMIT

The number of times that the Wi-Fi connection manager look for a network before giving up.

#define WLAN_RECONNECT_LIMIT 5U

The number of times that the Wi-Fi connection manager attempts a reconnection with the network before giving up.

#define WLAN_NETWORK_NAME_MIN_LENGTH 1U

Minimum length for network names, see wlan_network.

#define WLAN_NETWORK_NAME_MAX_LENGTH 32U

Maximum length for network names, see wlan_network

#define WLAN_PSK_MIN_LENGTH 8U

Minimum WPA2 passphrase can be up to 8 ASCII chars

#define WLAN_PSK_MAX_LENGTH 65U

Maximum WPA2 passphrase can be up to 63 ASCII chars or 64 hexadecimal digits + 1 '\0' char

#define WLAN_PASSWORD_MIN_LENGTH 8U

Minimum WPA3 password can be up to 8 ASCII chars

#define WLAN_PASSWORD_MAX_LENGTH 255U

Maximum WPA3 password can be up to 255 ASCII chars

#define IDENTITY_MAX_LENGTH 64U

Maximum enterprise identity can be up to 64 characters

#define PASSWORD_MAX_LENGTH 128U

Maximum enterprise password can be up to 128 characters

#define MAX_USERS 8U

Maximum identities for EAP server users

#define PAC_OPAQUE_ENCR_KEY_MAX_LENGTH 33U

Maximum length of encryption key for EAP-FAST PAC-Opaque values.

#define A_ID_MAX_LENGTH 33U

Maximum length of A-ID, A-ID indicates the identity of the authority that issues PACs.

#define HASH_MAX_LENGTH 40U

Maximum length of CA certification hash

#define DOMAIN_MATCH_MAX_LENGTH 64U

Maximum length of domain match

#define WLAN_MAX_KNOWN_NETWORKS CONFIG_WLAN_KNOWN_NETWORKS

The size of the list of known networks maintained by the Wi-Fi connection manager

#define WLAN_PMK_LENGTH 32

Length of a pairwise master key (PMK). It’s always 256 bits (32 Bytes)

#define WLAN_ERROR_NONE 0

Error codes The operation was successful.

#define WLAN_ERROR_PARAM 1

The operation failed due to an error with one or more parameters.

#define WLAN_ERROR_NOMEM 2

The operation could not be performed because there is not enough memory.

#define WLAN_ERROR_STATE 3

The operation could not be performed in the current system state.

#define WLAN_ERROR_ACTION 4

The operation failed due to an internal error.

#define WLAN_ERROR_PS_ACTION 5

The operation to change power state could not be performed

#define WLAN_ERROR_NOT_SUPPORTED 6

The requested feature is not supported

#define WLAN_MGMT_ACTION MBIT(13)

BITMAP for Action frame

#define WLAN_KEY_MGMT_FT **Value:** (WLAN_KEY_MGMT_FT_PSK | WLAN_KEY_MGMT_FT_IEEE8021X | WLAN_KEY_MGMT_FT_IEEE8021X_SHA384 | WLAN_KEY_MGMT_FT_SAE | \

WLAN_KEY_MGMT_FT_FILS_SHA256 | WLAN_KEY_MGMT_FT_FILS_SHA384)

Fast BSS Transition(11r) key management

#define MAX_CHANNEL_LIST 6

Configuration for Wi-Fi scan

Typedef Documentation

typedef wifi_pkt_stats_t wlan_pkt_stats_t

Wi-Fi firmware stat from wifi_pkt_stats_t

typedef wifi_stats_t wlan_stats_t

Wi-Fi driver stat from wifi_stats_t

typedef wifi_scan_channel_list_t wlan_scan_channel_list_t

Configuration for Wi-Fi scan channel list from wifi_scan_channel_list_t

typedef wifi_scan_params_v2_t wlan_scan_params_v2_t

Configuration for Wi-Fi scan parameters v2 from wifi_scan_params_v2_t

typedef wifi_cal_data_t wlan_cal_data_t

Configuration for Wi-Fi calibration data from wifi_cal_data_t

typedef wifi_flt_cfg_t wlan_flt_cfg_t

Configuration for memory efficient filters in Wi-Fi firmware from wifi_flt_cfg_t

typedef wifi_wowlan_ptn_cfg_t wlan_wowlan_ptn_cfg_t

Configuration for wowlan pattern parameters from wifi_wowlan_ptn_cfg_t

typedef wifi_tcp_keep_alive_t wlan_tcp_keep_alive_t

Configuration for TCP keep alive parameters from wifi_tcp_keep_alive_t

typedef wifi_ds_rate wlan_ds_rate

Configuration for TX rate and get data rate from wifi_ds_rate

typedef wifi_ed_mac_ctrl_t wlan_ed_mac_ctrl_t

Configuration for ED MAC Control parameters from wifi_ed_mac_ctrl_t

typedef wifi_bandcfg_t wlan_bandcfg_t

Configuration for band from wifi_bandcfg_t

typedef wifi_cw_mode_ctrl_t wlan_cw_mode_ctrl_t

Configuration for CW mode parameters from wifi_cw_mode_ctrl_t

typedef wifi_chanlist_t wlan_chanlist_t

Configuration for channel list from wifi_chanlist_t

typedef wifi_txpwrlimit_t wlan_txpwrlimit_t

Configuration for TX power Limit from wifi_txpwrlimit_t

typedef wifi_rutxpwrlimit_t wlan_rutxpwrlimit_t

Configuration for RU TX power limit from wifi_rutxpwrlimit_t

typedef wifi_11ax_config_t wlan_11ax_config_t

Configuration for 802.11ax capabilities wifi_11ax_config_t

typedef wifi_twt_setup_config_t wlan_twt_setup_config_t

Configuration for TWT setup wifi_twt_setup_config_t

typedef wifi_twt_teardown_config_t wlan_twt_teardown_config_t

Configuration for TWT teardown wifi_twt_teardown_config_t

typedef wifi_btwt_config_t wlan_btwt_config_t

Configuration for Broadcast TWT Setup wifi_btwt_config_t

typedef wifi_twt_report_t wlan_twt_report_t

Configuration for TWT Report wifi_twt_report_t

typedef wifi_twt_information_t wlan_twt_information_t

Configuration for TWT Information wifi_twt_information_t

typedef wifi_clock_sync_gpio_tsf_t wlan_clock_sync_gpio_tsf_t

Configuration for clock sync GPIO TSF latch wifi_clock_sync_gpio_tsf_t

typedef wifi_tsf_info_t wlan_tsf_info_t

Configuration for TSF info wifi_tsf_info_t

typedef wifi_csi_config_params_t wlan_csi_config_params_t

Configuration for CSI config params from wifi_csi_config_params_t

typedef wifi_net_monitor_t wlan_net_monitor_t

Configuration for net monitor from wifi_net_monitor_t

typedef wifi_host_tx_frame_params_t wlan_host_tx_frame_params_t

Configuration for host tx frame from wifi_host_tx_frame_params_t

typedef txrate_setting wlan_txrate_setting

Configuration for TX rate setting from txrate_setting

typedef wifi_rssi_info_t wlan_rssi_info_t

Configuration for RSSI information wifi_rssi_info_t

typedef wifi_ds_subscribe_evt wlan_ds_subscribe_evt

Configuration for subscribe events from wlan_ds_subscribe_evt

typedef wifi_auto_null_tx_t wlan_auto_null_tx_t

Configuration for auto null TX parameters from wifi_auto_null_tx_t

Enumeration Type Documentation

enum wm_wlan_errno

Enum for Wi-Fi errors

Enumerator:

enum wlan_event_reason

Wi-Fi connection manager event reason

Enumerator:

enum wlan_wakeup_event_t

Wakeup event bitmap

Enumerator:

enum wlan_connection_state

Wi-Fi station/uAP/Wi-Fi direct connection/status state

Enumerator:

enum wlan_ps_mode

Station power save mode

Enumerator:

enum wlan_security_type

Network security types

Enumerator:

enum eap_tls_cipher_type

EAP TLS Cipher types

Enumerator:

enum address_types

Address types to be used by the element wlan_ip_config.addr_type below

Enumerator:

enum sub_event_id

Type enum definition of subscribe event

Enumerator:

Chapter 2

RTOS

2.1 FreeRTOS

2.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

FreeRTOS kernel for MCUXpresso SDK Readme

FreeRTOS kernel for MCUXpresso SDK

Overview The purpose of this document is to describes the FreeRTOS kernel repo integration into the NXP MCUXpresso Software Development Kit: mcuxsdk. MCUXpresso SDK provides a comprehensive development solutions designed to optimize, ease, and help accelerate embedded system development of applications based on MCUs from NXP. This project involves the FreeRTOS kernel repo fork with:

- cmake and Kconfig support to allow the configuration and build in MCUXpresso SDK ecosystem
- FreeRTOS OS additions, such as FreeRTOS driver wrappers, RTOS ready FatFs file system, and the implementation of FreeRTOS tickless mode

The history of changes in FreeRTOS kernel repo for MCUXpresso SDK are summarized in *CHANGELOG_mcuxsdk.md* file.

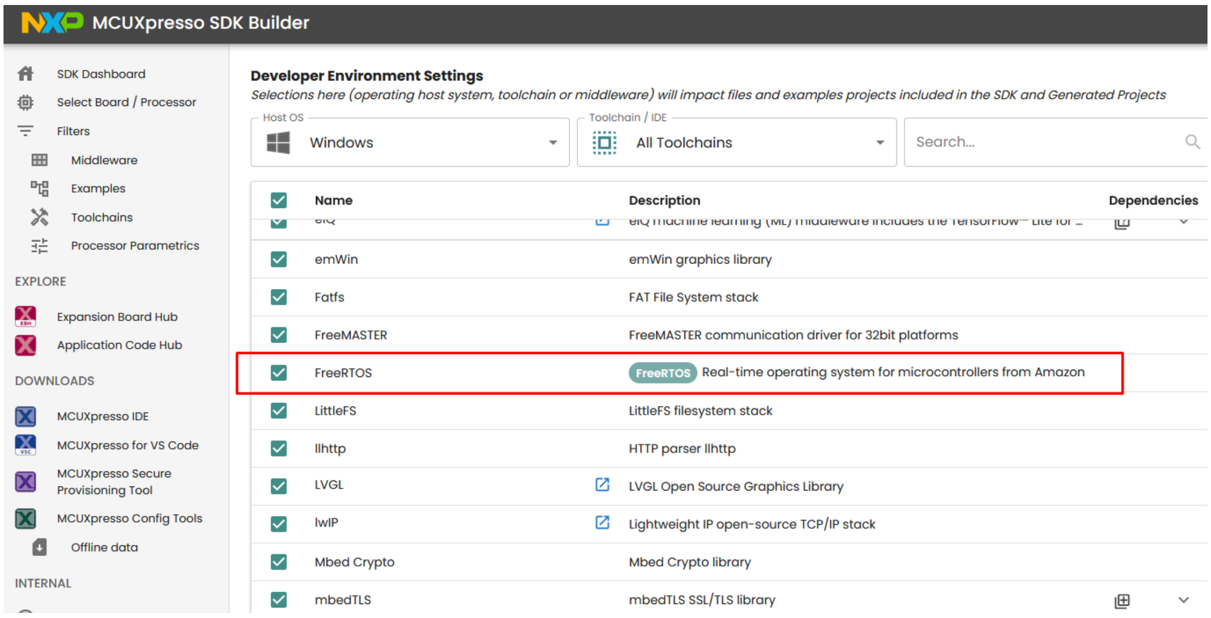
The MCUXpresso SDK framework also contains a set of FreeRTOS examples which show basic FreeRTOS OS features. This makes it easy to start a new FreeRTOS project or begin experimenting with FreeRTOS OS. Selected drivers and middleware are RTOS ready with related FreeRTOS adaptation layer.

FreeRTOS example applications The FreeRTOS examples are written to demonstrate basic FreeRTOS features and the interaction between peripheral drivers and the RTOS.

List of examples The list of freertos_examples, their description and availability for individual supported MCUXpresso SDK development boards can be obtained here: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/examples/freertos_examples/index.html

Location of examples The FreeRTOS examples are located in mcuxsdk-examples repository, see the freertos_examples folder.

Once using MCUXpresso SDK zip packages created via the MCUXpresso SDK Builder the FreeRTOS kernel library and associated freertos_examples are added into final zip package once FreeRTOS components is selected on the Developer Environment Settings page:



Developer Environment Settings

Selections here (operating host system, toolchain or middleware) will impact files and examples projects included in the SDK and Generated Projects

Host OS: Windows

Toolchain / IDE: All Toolchains

Search...

	Name	Description	Dependencies
✓	eIQ	eIQ machine learning (ML) middleware includes the TensorFlow™ Lite for ...	
✓	emWin	emWin graphics library	
✓	Fatfs	FAT File System stack	
✓	FreeMASTER	FreeMASTER communication driver for 32bit platforms	
✓	FreeRTOS	FreeRTOS Real-time operating system for microcontrollers from Amazon	
✓	LittleFS	LittleFS filesystem stack	
✓	llhttp	HTTP parser llhttp	
✓	LVGL	LVGL Open Source Graphics Library	
✓	lwIP	Lightweight IP open-source TCP/IP stack	
✓	Mbed Crypto	Mbed Crypto library	
✓	mbedTLS	mbedTLS SSL/TLS library	

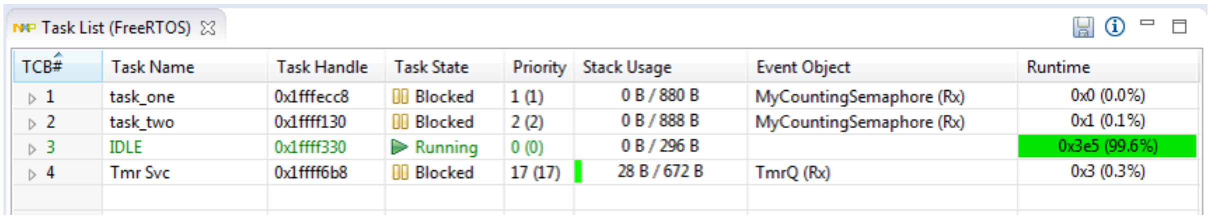
The FreeRTOS examples in MCUXpresso SDK zip packages are located in <MCUXpressoSDK_install_dir>/boards/<board_name>/freertos_examples/ subfolders.

Building a FreeRTOS example application For information how to use the cmake and Kconfig based build and configuration system and how to build freertos_examples visit: MCUXpresso SDK documentation for Build And Configuration MCUXpresso SDK Getting Start Guide

Tip: To list all FreeRTOS example projects and targets that can be built via the west build command, use this west list_project command in mcuxsdk workspace:

```
west list_project -p examples/freertos_examples
```

FreeRTOS aware debugger plugin NXP provides FreeRTOS task aware debugger for GDB. The plugin is compatible with Eclipse-based (MCUXpressoIDE) and is available after the installation.



TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Object	Runtime
1	task_one	0x1fffecc8	Blocked	1 (1)	0 B / 880 B	MyCountingSemaphore (Rx)	0x0 (0.0%)
2	task_two	0x1ffff130	Blocked	2 (2)	0 B / 888 B	MyCountingSemaphore (Rx)	0x1 (0.1%)
3	IDLE	0x1ffff330	Running	0 (0)	0 B / 296 B		0x3e5 (99.6%)
4	Tmr Svc	0x1ffff6b8	Blocked	17 (17)	28 B / 672 B	TmrQ (Rx)	0x3 (0.3%)

FreeRTOS kernel for MCUXpresso SDK ChangeLog

Changelog FreeRTOS kernel for MCUXpresso SDK All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[Unreleased]

Added

- Kconfig added CONFIG_FREERTOS_USE_CUSTOM_CONFIG_FRAGMENT config to optionally include custom FreeRTOSConfig fragment
  include file FreeRTOSConfig_frag.h. File must be provided by application.
- Added missing Kconfig option for configUSE_PICOLIBC_TLS.
- Add correct header files to build when configUSE_NEWLIB_REENTRANT and configUSE_PICOLIBC_TLS is selected in config.

[11.1.0_rev0]

- update amazon freertos version

[11.0.1_rev0]

- update amazon freertos version

[10.5.1_rev0]

- update amazon freertos version

[10.4.3_rev1]

- Apply CM33 security fix from 10.4.3-LTS-Patch-2. See rtos\freertos\freertos_kernel\History.txt
- Apply CM33 security fix from 10.4.3-LTS-Patch-1. See rtos\freertos\freertos_kernel\History.txt

[10.4.3_rev0]

- update amazon freertos version.

[10.4.3_rev0]

- update amazon freertos version.

[9.0.0_rev3]

- New features:
  - Tickless idle mode support for Cortex-A7. Add fsl_tickless_epit.c and fsl_tickless_generic.h in portable/IAR/ARM_CA9 folder.
  - Enabled float context saving in IAR for Cortex-A7. Added configUSE_TASK_FPU_SUPPORT macros. Modified port.c and portmacro.h in portable/IAR/ARM_CA9 folder.
- Other changes:
  - Transformed ARM_CM core specific tickless low power support into generic form under freertos/Source/portable/low_power_tickless/.

[9.0.0_rev2]

- New features:
  - Enabled MCUXpresso thread aware debugging. Add freertos_tasks_c_additions.h and configINCLUDE_FREERTOS_TASK_C_ADDITIONS_H and configFRTOS_MEMORY_SCHEME macros.

[9.0.0_rev1]

- New features:
  - Enabled -flto optimization in GCC by adding **attribute**((used)) for vTaskSwitchContext.
  - Enabled KDS Task Aware Debugger. Apply FreeRTOS patch to enable configRECORD_STACK_HIGH_ADDRESS macro. Modified files are task.c and FreeRTOS.h.

[9.0.0_rev0]

- New features:
  - Example freertos_sem_static.
  - Static allocation support RTOS driver wrappers.
- Other changes:
  - Tickless idle rework. Support for different timers is in separated files (fsl_tickless_systick.c, fsl_tickless_lptmr.c).
  - Removed configuration option configSYSTICK_USE_LOW_POWER_TIMER. Low power timer is now selected by linking of apropriate file fsl_tickless_lptmr.c.
  - Removed configOVERRIDE_DEFAULT_TICK_CONFIGURATION in RVDS port. Use of **attribute**((weak)) is the preferred solution. Not same as _weak!

[8.2.3]

- New features:
  - Tickless idle mode support.
  - Added template application for Kinetis Expert (KEx) tool (template_application).
- Other changes:
  - Folder structure reduction. Keep only Kinetis related parts.

FreeRTOS kernel Readme

MCUXpresso SDK: FreeRTOS kernel This repository is a fork of FreeRTOS kernel (https://github.com/FreeRTOS/FreeRTOS-Kernel)(11.1.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS kernel repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

For more information about the FreeRTOS kernel repo adoption see README_mcuxsdk.md: FreeRTOS kernel for MCUXpresso SDK Readme document.

CMock Unit Tests passing

Getting started This repository contains FreeRTOS kernel source/header files and kernel ports only. This repository is referenced as a submodule in FreeRTOS/FreeRTOS repository, which contains pre-configured demo application projects under `FreeRTOS/Demo` directory.

The easiest way to use FreeRTOS is to start with one of the pre-configured demo application projects. That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the FreeRTOS Kernel Quick Start Guide for detailed instructions and other useful links.

Additionally, for FreeRTOS kernel feature information refer to the Developer Documentation, and API Reference.

Also for contributing and creating a Pull Request please refer to *the instructions here*.

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the FreeRTOS Community Support Forum.

To consume FreeRTOS-Kernel

Consume with CMake If using CMake, it is recommended to use this repository using FetchContent. Add the following into your project's main or a subdirectory's `CMakeLists.txt`:

- Define the source and version/tag you want to use:

```
FetchContent_Declare( freertos_kernel
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Kernel.git
  GIT_TAG        main #Note: Best practice to use specific git-hash or tagged version
)
```

In case you prefer to add it as a git submodule, do:

```
git submodule add https://github.com/FreeRTOS/FreeRTOS-Kernel.git <path of the submodule>
git submodule update --init
```

- Add a freertos_config library (typically an INTERFACE library) The following assumes the directory structure:
  - `include/FreeRTOSConfig.h`

```
add_library(freertos_config INTERFACE)

target_include_directories(freertos_config SYSTEM
INTERFACE
    include
)

target_compile_definitions(freertos_config
  INTERFACE
    projCOVERAGE_TEST=0
)
```

In case you installed FreeRTOS-Kernel as a submodule, you will have to add it as a subdirectory:

```
add_subdirectory(${FREERTOS_PATH})
```

- Configure the FreeRTOS-Kernel and make it available
  - this particular example supports a native and cross-compiled build option.

```
set( FREERTOS_HEAP "4" CACHE STRING "" FORCE)
# Select the native compile PORT
set( FREERTOS_PORT "GCC_POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  set(FREERTOS_PORT "GCC_ARM_CA9" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_kernel)
```

- In case of cross compilation, you should also add the following to freertos_config:

```
target_compile_definitions(freertos_config INTERFACE ${definitions})
target_compile_options(freertos_config INTERFACE ${options})
```

Consuming stand-alone - Cloning this repository To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Kernel.git
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Kernel.git
```

Repository structure

- The root of this repository contains the three files that are common to every port - list.c, queue.c and tasks.c. The kernel is contained within these three files. croutine.c implements the optional co-routine functionality - which is normally only used on very memory limited systems.
- The ./portable directory contains the files that are specific to a particular microcontroller and/or compiler. See the readme file in the ./portable directory for more information.
- The ./include directory contains the real time kernel header files.
- The ./template_configuration directory contains a sample FreeRTOSConfig.h to help jumpstart a new project. See the *FreeRTOSConfig.h* file for instructions.

Code Formatting FreeRTOS files are formatted using the "uncrustify" tool. The configuration file used by uncrustify can be found in the FreeRTOS/CI-CD-GitHub-Actions's uncrustify.cfg file.

Line Endings File checked into the FreeRTOS-Kernel repository use unix-style LF line endings for the best compatibility with git.

For optimal compatibility with Microsoft Windows tools, it is best to enable the git autocrlf feature. You can enable this setting for the current repository using the following command:

```
git config core.autocrlf true
```

Git History Optimizations Some commits in this repository perform large refactors which touch many lines and lead to unwanted behavior when using the git blame command. You can configure git to ignore the list of large refactor commits in this repository with the following command:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

Spelling and Formatting We recommend using Visual Studio Code, commonly referred to as VSCode, when working on the FreeRTOS-Kernel. The FreeRTOS-Kernel also uses cSpell as part of its spelling check. The config file for which can be found at *cspell.config.yaml* There is additionally a cSpell plugin for VSCode that can be used as well. .cSpellWords.txt contains words that are not traditionally found in an English dictionary. It is used by the spellchecker to verify the various jargon, variable names, and other odd words used in the FreeRTOS code base are correct. If your pull request fails to pass the spelling and you believe this is a mistake, then add the word to .cSpellWords.txt. When adding a word please then sort the list, which can be done by running the bash command: `sort -u .cSpellWords.txt -o .cSpellWords.txt` Note that only the FreeRTOS-Kernel Source Files, *include*, *portable/MemMang*, and *portable/Common* files are checked for proper spelling, and formatting at this time.

2.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

2.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

Readme

MCUXpresso SDK: backoffAlgorithm Library This repository is a fork of backoffAlgorithm library (https://github.com/FreeRTOS/backoffalgorithm)(1.3.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable backoffAlgorithm repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

backoffAlgorithm Library This repository contains the backoffAlgorithm library, a utility library to calculate backoff period using an exponential backoff with jitter algorithm for retrying network operations (like failed network connection with server). This library uses the "Full Jitter" strategy for the exponential backoff with jitter algorithm. More information about the algorithm can be seen in the Exponential Backoff and Jitter AWS blog.

The backoffAlgorithm library is distributed under the *MIT Open Source License*.

Exponential backoff with jitter is typically used when retrying a failed network connection or operation request with the server. An exponential backoff with jitter helps to mitigate failed network operations with servers, that are caused due to network congestion or high request load on the server, by spreading out retry requests across multiple devices attempting network operations. Besides, in an environment with poor connectivity, a client can get disconnected at any time. A backoff strategy helps the client to conserve battery by not repeatedly attempting reconnections when they are unlikely to succeed.

See memory requirements for this library *here*.

backoffAlgorithm v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.

backoffAlgorithm v1.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Reference example The example below shows how to use the backoffAlgorithm library on a POSIX platform to retry a DNS resolution query for amazon.com.

```
#include "backoff_algorithm.h"
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>
#include <time.h>

/* The maximum number of retries for the example code. */
#define RETRY_MAX_ATTEMPTS            ( 5U )

/* The maximum back-off delay (in milliseconds) for between retries in the example. */
#define RETRY_MAX_BACKOFF_DELAY_MS    ( 5000U )

/* The base back-off delay (in milliseconds) for retry configuration in the example. */
#define RETRY_BACKOFF_BASE_MS         ( 500U )

int main()
{
    /* Variables used in this example. */
    BackoffAlgorithmStatus_t retryStatus = BackoffAlgorithmSuccess;
    BackoffAlgorithmContext_t retryParams;
    char serverAddress[] = "amazon.com";
    uint16_t nextRetryBackoff = 0;

    int32_t dnsStatus = -1;
    struct addrinfo hints;
    struct addrinfo ** pListHead = NULL;
    struct timespec tp;

    /* Add hints to retrieve only TCP sockets in getaddrinfo. */
    ( void ) memset( &hints, 0, sizeof( hints ) );

    /* Address family of either IPv4 or IPv6. */
    hints.ai_family = AF_UNSPEC;
    /* TCP Socket. */
    hints.ai_socktype = ( int32_t ) SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    /* Initialize reconnect attempts and interval. */
    BackoffAlgorithm_InitializeParams( &retryParams,
                                       RETRY_BACKOFF_BASE_MS,
                                       RETRY_MAX_BACKOFF_DELAY_MS,
                                       RETRY_MAX_ATTEMPTS );


    /* Seed the pseudo random number generator used in this example (with call to
     * rand() function provided by ISO C standard library) for use in backoff period
     * calculation when retrying failed DNS resolution. */

    /* Get current time to seed pseudo random number generator. */
    ( void ) clock_gettime( CLOCK_REALTIME, &tp );
    /* Seed pseudo random number generator with seconds. */
    srand( tp.tv_sec );

    do
    {
        /* Perform a DNS lookup on the given host name. */
        dnsStatus = getaddrinfo( serverAddress, NULL, &hints, pListHead );
```

(continues on next page)

(continued from previous page)

```
        /* Retry if DNS resolution query failed. */
        if( dnsStatus != 0 )
        {
            /* Generate a random number and get back-off value (in milliseconds) for the next retry.
             * Note: It is recommended to use a random number generator that is seeded with
             * device-specific entropy source so that backoff calculation across devices is different
             * and possibility of network collision between devices attempting retries can be avoided.
             *
             * For the simplicity of this code example, the pseudo random number generator, rand()
             * function is used. */
            retryStatus = BackoffAlgorithm_GetNextBackoff( &retryParams, rand(), &nextRetryBackoff );

            /* Wait for the calculated backoff period before the next retry attempt of querying DNS.
             * As usleep() takes nanoseconds as the parameter, we multiply the backoff period by 1000. */
            ( void ) usleep( nextRetryBackoff * 1000U );
        }
    } while( ( dnsStatus != 0 ) && ( retryStatus != BackoffAlgorithmRetriesExhausted ) );

    return dnsStatus;
}
```

Building the library A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Additionally, the library uses a header file introduced in ISO C99, stdint.h. For compilers that do not provide this header file, the *source/include* directory contains *stdint.readme*, which can be renamed to stdint.h to build the backoffAlgorithm library.

For instance, if the example above is copied to a file named example.c, *gcc* can be used like so:

```
gcc -I source/include example.c source/backoff_algorithm.c -o example
./example
```

gcc can also produce an output file to be linked:

```
gcc -I source/include -c source/backoff_algorithm.c
```

Building unit tests

Checkout Unity Submodule By default, the submodules in this repository are configured with update=none in *.gitmodules*, to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/Unity
```

Platform Prerequisites

- For running unit tests
  - C89 or later compiler like gcc
  - CMake 3.13.0 or later
- For running the coverage target, gcov is additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described *above*.)
2. Create build directory: `mkdir build && cd build`
3. Run *cmake* while inside build directory: `cmake -S ../test`
4. Run this command to build the library and unit tests: `make all`
5. The generated test executables will be present in `build/bin/tests` folder.
6. Run `ctest` to execute all tests and view the test run summary.

Contributing See *CONTRIBUTING.md* for information on contributing.

2.1.4 corehttp

C language HTTP client library designed for embedded platforms.

MCUXpresso SDK: coreHTTP Client Library

This repository is a fork of coreHTTP Client library (https://github.com/FreeRTOS/corehttp)(3.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreHTTP Client repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreHTTP Client Library

This repository contains a C language HTTP client library designed for embedded platforms. It has no dependencies on any additional libraries other than the standard C library, llhttp, and a customer-implemented transport interface. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety and data structure invariance through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

coreHTTP v3.0.0 source code is part of the FreeRTOS 202210.00 LTS release.

coreHTTP v2.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

coreHTTP Config File The HTTP client library exposes configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core_http_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named `core_http_config.h` can be provided by the user application to the library.

By default, a `core_http_config.h` custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide `HTTP_DO_NOT_USE_CUSTOM_CONFIG` as a compile time preprocessor macro.

The HTTP client library can be built by either:

- Defining a `core_http_config.h` file in the application, and adding it to the include directories for the library build. **OR**
- Defining the `HTTP_DO_NOT_USE_CUSTOM_CONFIG` preprocessor macro for the library build.

Building the Library The *httpFilePaths.cmake* file contains the information of all source files and header include paths required to build the HTTP client library.

As mentioned in the *previous section*, either a custom config file (i.e. `core_http_config.h`) OR `HTTP_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the HTTP client library.

For a CMake example of building the HTTP library with the `httpFilePaths.cmake` file, refer to the `coverity_analysis` library target in *test/CMakeLists.txt* file.

Building Unit Tests

Platform Prerequisites

- For running unit tests, the following are required:
  - **C90 compiler** like gcc
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is required for this repository's CMock test framework.
- For running the coverage target, the following are required:
  - **gcov**
  - **lcov**

Steps to build Unit Tests

1. Go to the root directory of this repository.
2. Run the *cmake* command: `cmake -S test -B build -DBUILD_CLONE_SUBMODULES=ON`
3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples The AWS IoT Device SDK for Embedded C repository contains demos of using the HTTP client library here on a POSIX platform. These can be used as reference examples for the library API.

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of coreHTTP may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

2.1.5 corejson

JSON parser.

Readme

MCUXpresso SDK: coreJSON Library This repository is a fork of coreJSON library (https://github.com/FreeRTOS/corejson)(3.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreJSON repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreJSON Library This repository contains the coreJSON library, a parser that strictly enforces the ECMA-404 JSON standard and is suitable for low memory footprint embedded devices. The coreJSON library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

coreJSON v3.2.0 source code is part of the FreeRTOS 202210.00 LTS release.

coreJSON v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Reference example

```
#include <stdio.h>
#include "core_json.h"

int main()
{
    // Variables used in this example.
    JSONStatus_t result;
    char buffer[] = "{\"foo\":\"abc\",\"bar\":{\"foo\":\"xyz\"}}";
    size_t bufferLength = sizeof( buffer ) - 1;
    char queryKey[] = "bar.foo";
    size_t queryKeyLength = sizeof( queryKey ) - 1;
    char * value;
    size_t valueLength;

    // Calling JSON_Validate() is not necessary if the document is guaranteed to be valid.
    result = JSON_Validate( buffer, bufferLength );

    if( result == JSONSuccess )
    {
        result = JSON_Search( buffer, bufferLength, queryKey, queryKeyLength,
                              &value, &valueLength );
    }

    if( result == JSONSuccess )
    {
        // The pointer "value" will point to a location in the "buffer".
        char save = value[ valueLength ];
        // After saving the character, set it to a null byte for printing.
        value[ valueLength ] = '\0';
        // "Found: bar.foo -> xyz" will be printed.
        printf( "Found: %s -> %s\n", queryKey, value );
        // Restore the original character.
        value[ valueLength ] = save;
    }

    return 0;
}
```

A search may descend through nested objects when the `queryKey` contains matching key strings joined by a separator, `.`. In the example above, `bar` has the value `{"foo":"xyz"}`. Therefore, a search for query key `bar.foo` would output `xyz`.

Building coreJSON

A compiler that supports **C90 or later** such as *gcc* is required to build the library.

Additionally, the library uses 2 header files introduced in ISO C99, `stdbool.h` and `stdint.h`. For compilers that do not provide this header file, the *source/include* directory contains *stdbool.readme* and *stdint.readme*, which can be renamed to `stdbool.h` and `stdint.h` respectively.

For instance, if the example above is copied to a file named `example.c`, *gcc* can be used like so:

```
gcc -I source/include example.c source/core_json.c -o example
./example
```

gcc can also produce an output file to be linked:

```
gcc -I source/include -c source/core_json.c
```

Documentation

Existing documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of the coreJSON library may differ across repositories.

Generating documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Building unit tests

Checkout Unity Submodule By default, the submodules in this repository are configured with update=none in *.gitmodules*, to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of Unity is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/Unity
```

Platform Prerequisites

- For running unit tests
  - C90 compiler like gcc
  - CMake 3.13.0 or later
  - Ruby 2.0.0 or later is additionally required for the Unity test framework (that we use).
- For running the coverage target, gcov is additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **Unity** submodule is cloned as described *above*.)
2. Create build directory: `mkdir build && cd build`
3. Run *cmake* while inside build directory: `cmake -S ../test`
4. Run this command to build the library and unit tests: `make all`
5. The generated test executables will be present in `build/bin/tests` folder.
6. Run `ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The test/cbmc/proofs directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Contributing See *CONTRIBUTING.md* for information on contributing.

2.1.6 coremqtt

MQTT publish/subscribe messaging library.

MCUXpresso SDK: coreMQTT Library

This repository is a fork of coreMQTT library (https://github.com/FreeRTOS/coremqtt)(2.1.1). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreMQTT Client Library

This repository contains the coreMQTT library that has been optimized for a low memory footprint. The coreMQTT library is compliant with the MQTT 3.1.1 standard. It has no dependencies on any additional libraries other than the standard C library, a customer-implemented network transport interface, and *optionally* a user-implemented platform time function. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

coreMQTT v2.1.1 source code is part of the FreeRTOS 202210.01 LTS release.

MQTT Config File The MQTT client library exposes build configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core_mqtt_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named core_mqtt_config.h can be provided by the application to the library.

By default, a core_mqtt_config.h custom config is required to build the library. To disable this requirement and build the library with default configuration values, provide MQTT_DO_NOT_USE_CUSTOM_CONFIG as a compile time preprocessor macro.

Thus, the MQTT library can be built by either:

- Defining a core_mqtt_config.h file in the application, and adding it to the include directories list of the library
  OR
- Defining the MQTT_DO_NOT_USE_CUSTOM_CONFIG preprocessor macro for the library build.

Sending metrics to AWS IoT When establishing a connection with AWS IoT, users can optionally report the Operating System, Hardware Platform and MQTT client version information of their device to AWS. This information can help AWS IoT provide faster issue resolution and technical support. If users want to report this information, they can send a specially formatted string (see below) in the username field of the MQTT CONNECT packet.

Format

The format of the username string with metrics is:

```
<Actual_Username>?SDK=<OS_Name>&Version=<OS_Version>&Platform=<Hardware_Platform>&
↪MQTTLib=<MQTT_Library_name>@<MQTT_Library_version>
```

Where

- <Actual_Username> is the actual username used for authentication, if username and password are used for authentication. When username and password based authentication is not used, this is an empty value.
- <OS_Name> is the Operating System the application is running on (e.g. FreeRTOS)
- <OS_Version> is the version number of the Operating System (e.g. V10.4.3)
- <Hardware_Platform> is the Hardware Platform the application is running on (e.g. WinSim)
- <MQTT_Library_name> is the MQTT Client library being used (e.g. coreMQTT)
- <MQTT_Library_version> is the version of the MQTT Client library being used (e.g. 1.0.2)

Example

- Actual_Username = "iotuser", OS_Name = FreeRTOS, OS_Version = V10.4.3, Hardware_Platform_Name = WinSim, MQTT_Library_Name = coremqtt, MQTT_Library_version = 2.1.1. If username is not used, then "iotuser" can be removed.

```
/* Username string:
 * iotuser?SDK=FreeRTOS&Version=v10.4.3&Platform=WinSim&MQTTLib=coremqtt@2.1.1
 */

#define OS_NAME                   "FreeRTOS"
#define OS_VERSION                "V10.4.3"
#define HARDWARE_PLATFORM_NAME    "WinSim"
#define MQTT_LIB                  "coremqtt@2.1.1"

#define USERNAME_STRING           "iotuser?SDK=" OS_NAME "&Version=" OS_VERSION "&
↪Platform=" HARDWARE_PLATFORM_NAME "&MQTTLib=" MQTT_LIB
#define USERNAME_STRING_LENGTH    ( ( uint16_t ) ( sizeof( USERNAME_STRING ) - 1 ) )

MQTTConnectInfo_t connectInfo;
connectInfo.pUserName = USERNAME_STRING;
connectInfo.userNameLength = USERNAME_STRING_LENGTH;
mqttStatus = MQTT_Connect( pMqttContext, &connectInfo, NULL, CONNACK_RECV_TIMEOUT_MS,␣
↪pSessionPresent );
```

Upgrading to v2.0.0 and above With coreMQTT versions >=v2.0.0, there are breaking changes. Please refer to the *coreMQTT version >=v2.0.0 Migration Guide*.

Building the Library The *mqttFilePaths.cmake* file contains the information of all source files and the header include path required to build the MQTT library.

Additionally, the MQTT library requires two header files that are not part of the ISO C90 standard library, `stdbool.h` and `stdint.h`. For compilers that do not provide these header files, the *source/include* directory contains the files *stdbool.readme* and *stdint.readme*, which can be renamed to `stdbool.h` and `stdint.h`, respectively, to provide the type definitions required by MQTT.

As mentioned in the previous section, either a custom config file (i.e. `core_mqtt_config.h`) OR `MQTT_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the MQTT library.

For a CMake example of building the MQTT library with the `mqttFilePaths.cmake` file, refer to the `coverity_analysis` library target in *test/CMakeLists.txt* file.

Building Unit Tests

Checkout CMock Submodule By default, the submodules in this repository are configured with `update=none` in *.gitmodules* to avoid increasing clone time and disk space usage of other repositories (like amazon-freertos that submodules this repository).

To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

Platform Prerequisites

- Docker

or the following:

- For running unit tests
  - **C90 compiler** like gcc
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build Unit Tests

1. If using docker, launch the container:
   1. `docker build -t coremqtt .`
   2. `docker run -it -v "$PWD":/workspaces/coreMQTT -w /workspaces/coreMQTT coremqtt`
2. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described *above*)
3. Run the *cmake* command: `cmake -S test -B build`
4. Run this command to build the library and unit tests: `make -C build all`
5. The generated test executables will be present in `build/bin/tests` folder.
6. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples Please refer to the demos of the MQTT client library in the following locations for reference examples on POSIX and FreeRTOS platforms:

Platform	Location	Transport Interface Implementation
POSIX	AWS IoT Device SDK for Embedded C	POSIX sockets for TCP/IP and OpenSSL for TLS stack
FreeRTOS	FreeRTOS/FreeRTOS	FreeRTOS+TCP for TCP/IP and mbedTLS for TLS stack
FreeRTOS	FreeRTOS AWS Reference Integrations	Based on Secure Sockets Abstraction

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of coreMQTT may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Contributing See *CONTRIBUTING.md* for information on contributing.

2.1.7 coremqtt-agent

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library.

Readme

MCUXpresso SDK: coreMQTT Agent Library This repository is a fork of coreMQTT Agent library (https://github.com/FreeRTOS/coremqtt-agent)(1.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT Agent repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

coreMQTT Agent Library The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library. The library provides thread safe equivalents to the coreMQTT's APIs, greatly simplifying its use in multi-threaded environments. The coreMQTT Agent library manages the MQTT connection by serializing the access to the coreMQTT library and reducing implementation overhead (e.g., removing the need for the application to repeatedly call to MQTT_ProcessLoop). This allows your multi-threaded applications to share the same MQTT connection, and enables you to design an embedded application without having to worry about coreMQTT thread safety.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis, and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

Cloning this repository This repo uses Git Submodules to bring in dependent components.

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

Using SSH:

```
git clone git@github.com:FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

If you have downloaded the repo without using the --recurse-submodules argument, you need to run:

```
git submodule update --init --recursive
```

coreMQTT Agent Library Configurations The MQTT Agent library uses the same core_mqtt_config.h configuration file as coreMQTT, with the addition of configuration constants listed at the top of *core_mqtt_agent.h* and *core_mqtt_agent_command_functions.h*. Documentation for these configurations can be found here.

To provide values for these configuration values, they must be either:

- Defined in core_mqtt_config.h used by coreMQTT **OR**
- Passed as compile time preprocessor macros

Porting the coreMQTT Agent Library In order to use the MQTT Agent library on a platform, you need to supply thread safe functions for the agent's *messaging interface*.

Messaging Interface Each of the following functions must be thread safe.

Function Pointer	Description
MQTTAgentMessageSend_t	A function that sends commands (as MQTTAgentCommand_t * pointers) to be received by MQTTAgent_CommandLoop. This can be implemented by pushing to a thread safe queue.
MQTTAgentMessageRecv_t	A function used by MQTTAgent_CommandLoop to receive MQTTAgentCommand_t * pointers that were sent by API functions. This can be implemented by receiving from a thread safe queue.
MQTTAgentCommandGet_t	A function that returns a pointer to an allocated MQTTAgentCommand_t structure, which is used to hold information and arguments for a command to be executed in MQTTAgent_CommandLoop(). If using dynamic memory, this can be implemented using malloc().
MQTTAgentCommandRelease_t	A function called to indicate that a command structure that had been allocated with the MQTTAgentCommandGet_t function pointer will no longer be used by the agent, so it may be freed or marked as not in use. If using dynamic memory, this can be implemented with free().

Reference implementations for the interface functions can be found in the *reference examples* below.

Additional Considerations

Static Memory If only static allocation is used, then the MQTTAgentCommandGet_t and MQTTAgentCommandRelease_t could instead be implemented with a pool of MQTTAgentCommand_t structures, with a queue or semaphore used to control access and provide thread safety. The below *reference examples* use static memory with a command pool.

Subscription Management The MQTT Agent does not track subscriptions for MQTT topics. The receipt of any incoming PUBLISH packet will result in the invocation of a single MQTTAgentIncomingPublishCallback_t callback, which is passed to MQTTAgent_Init() for initialization. If it is desired for different handlers to be invoked for different incoming topics, then the publish callback will have to manage subscriptions and fan out messages. A platform independent subscription manager example is implemented in the *reference examples* below.

Building the Library You can build the MQTT Agent source files that are in the *source* directory, and add *source/include* to your compiler's include path. Additionally, the MQTT Agent library requires the coreMQTT library, whose files follow the same source/ and source/include pattern as the agent library; its build instructions can be found here.

If using CMake, the *mqttAgentFilePaths.cmake* file contains the above information of the source files and the header include path from this repository. The same information is found for coreMQTT from mqttFilePaths.cmake in the *coreMQTT submodule*.

For a CMake example of building the MQTT Agent library with the mqttAgentFilePaths.cmake file, refer to the coverity_analysis library target in *test/CMakeLists.txt* file.

Building Unit Tests

Checkout CMock Submodule To build unit tests, the submodule dependency of CMock is required. Use the following command to clone the submodule:

```
git submodule update --checkout --init --recursive test/unit-test/CMock
```

Unit Test Platform Prerequisites

- For running unit tests
  - **C90 compiler** like gcc
  - **CMake 3.13.0 or later**
  - **Ruby 2.0.0 or later** is additionally required for the CMock test framework (that we use).
- For running the coverage target, **gcov** and **lcov** are additionally required.

Steps to build Unit Tests

1. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described *above*)
2. Run the *cmake* command: `cmake -S test -B build`
3. Run this command to build the library and unit tests: `make -C build all`
4. The generated test executables will be present in `build/bin/tests` folder.
5. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples Please refer to the demos of the MQTT Agent library in the following locations for reference examples on FreeRTOS platforms:

Location
coreMQTT Agent Demos
FreeRTOS/FreeRTOS

Documentation The MQTT Agent API documentation can be found here.

Generating documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages yourself, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Getting help You can use your Github login to get support from both the FreeRTOS community and directly from the primary FreeRTOS developers on our active support forum. You can find a list of frequently asked questions here.

Contributing See *CONTRIBUTING.md* for information on contributing.

License This library is licensed under the MIT License. See the *LICENSE* file.

2.1.8 corepkcs11

PKCS #11 key management library.

Readme

MCUXpresso SDK: corePKCS11 Library This repository is a fork of PKCS #11 key management library (https://github.com/FreeRTOS/corePKCS11/tree/v3.5.0)(v3.5.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable corepkcs11 repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

corePKCS11 Library PKCS #11 is a standardized and widely used API for manipulating common cryptographic objects. It is important because the functions it specifies allow application software to use, create, modify, and delete cryptographic objects, without ever exposing those objects to the application's memory. For example, FreeRTOS AWS reference integrations use a small subset of the PKCS #11 API to, among other things, access the secret (private) key necessary to create a network connection that is authenticated and secured by the Transport Layer Security (TLS) protocol – without the application ever 'seeing' the key.

The Cryptoki or PKCS #11 standard defines a platform-independent API to manage and use cryptographic tokens. The name, "PKCS #11", is used interchangeably to refer to the API itself and the standard which defines it.

This repository contains a software based mock implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. Using a software mock enables rapid development and flexibility, but it is expected that the mock be replaced by an implementation specific to your chosen secure key storage in production devices.

Only a subset of the PKCS #11 standard is implemented, with a focus on operations involving asymmetric keys, random number generation, and hashing.

The targeted use cases include certificate and key management for TLS authentication and code-sign signature verification, on small embedded devices.

corePKCS11 is implemented on PKCS #11 v2.4.0, the full PKCS #11 standard can be found on the oasis website.

This library has gone through code quality checks including verification that no function has a GNU Complexity score over 8, and checks against deviations from mandatory rules in the MISRA coding standard. Deviations from the MISRA C:2012 guidelines are documented under *MISRA Deviations*. This library has also undergone both static code analysis from Coverity static analysis and validation of memory safety through the CBMC automated reasoning tool.

See memory requirements for this library *here*.

corePKCS11 v3.5.0 source code is part of the FreeRTOS 202210.00 LTS release.

corePKCS11 v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Purpose Generally vendors for secure cryptoprocessors such as Trusted Platform Module (TPM), Hardware Security Module (HSM), Secure Element, or any other type of secure hardware enclave, distribute a PKCS #11 implementation with the hardware. The purpose of the corePKCS11 software only mock library is therefore to provide a non hardware specific PKCS #11 implementation that allows for rapid prototyping and development before switching to a cryptoprocessor specific PKCS #11 implementation in production devices.

Since the PKCS #11 interface is defined as part of the PKCS #11 specification replacing this library with another implementation should require little porting effort, as the interface will not change. The system tests distributed in this repository can be leveraged to verify the behavior of a different implementation is similar to corePKCS11.

corePKCS11 Configuration The corePKCS11 library exposes preprocessor macros which must be defined prior to building the library. A list of all the configurations and their default values are defined in the doxygen documentation for this library.

Build Prerequisites

Library Usage For building the library the following are required:

- **A C99 compiler**
- **mbedcrypto** library from mbedtls version 2.x or 3.x.
- **pkcs11 API header(s)** available from OASIS or OpenSC

Optionally, variables from the pkcsFilePaths.cmake file may be referenced if your project uses cmake.

Integration and Unit Tests In order to run the integration and unit test suites the following are dependencies are necessary:

- **C Compiler**
- **CMake 3.13.0 or later**
- **Ruby 2.0.0 or later** required by CMock.
- **Python 3** required for configuring mbedtls.
- **git** required for fetching dependencies.
- **GNU Make** or **Ninja**

The *mbedtls*, *CMock*, and *Unity* libraries are downloaded and built automatically using the cmake FetchContent feature.

Coverage Measurement and Instrumentation The following software is required to run the coverage target:

- Linux, MacOS, or another POSIX-like environment.
- A recent version of **GCC** or **Clang** with support for gcov-like coverage instrumentation.
- **gcov** binary corresponding to your chosen compiler
- **lcov** from the Linux Test Project
- **perl** needed to run the lcov utility.

Coverage builds are validated on recent versions of Ubuntu Linux.

Running the Integration and Unit Tests

1. Navigate to the root directory of this repository in your shell.
2. Run **cmake** to construct a build tree: `cmake -S test -B build`
   - You may specify your preferred build tool by appending `-G'Unix Makefiles'` or `-GNinja` to the command above.
   - You may append `-DUNIT_TESTS=0` or `-DSYSTEM_TESTS=0` to disable Unit Tests or Integration Tests respectively.
3. Build the test binaries: `cmake --build ./build --target all`
4. Run `ctest --test-dir ./build` or `cmake --build ./build --target test` to run the tests without capturing coverage.
5. Run `cmake --build ./build --target coverage` to run the tests and capture coverage data.

CBMC To learn more about CBMC and proofs specifically, review the training material here.

The `test/cbmc/proofs` directory contains CBMC proofs.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.

Reference examples The FreeRTOS-Labs repository contains demos using the PKCS #11 library here using FreeRTOS on the Windows simulator platform. These can be used as reference examples for the library API.

Porting Guide Documentation for porting corePKCS11 to a new platform can be found on the AWS docs web page.

corePKCS11 is not meant to be ported to projects that have a TPM, HSM, or other hardware for offloading crypto-processing. This library is specifically meant to be used for development and prototyping.

Related Example Implementations These projects implement the PKCS #11 interface on real hardware and have similar behavior to corePKCS11. It is preferred to use these, over corePKCS11, as they allow for offloading Cryptography to separate hardware.

- ARM's Platform Security Architecture.
- Microchip's cryptoauthlib.
- Infineon's Optiga Trust X.

Documentation

Existing Documentation For pre-generated documentation, please see the documentation linked in the locations below:

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

Note that the latest included version of corePKCS11 may differ across repositories.

Generating Documentation The Doxygen references were created using Doxygen version 1.9.2. To generate the Doxygen pages, please run the following command from the root of this repository:

```
doxygen docs/doxygen/config.doxyfile
```

Security See *CONTRIBUTING* for more information.

License This library is licensed under the MIT-0 License. See the LICENSE file.

2.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme

MCUXpresso SDK: FreeRTOS-Plus-TCP Library This repository is a fork of FreeRTOS-Plus-TCP library (https://github.com/FreeRTOS/freertos-plus-tcp)(4.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS-Plus-TCP repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(https://github.com/nxp-mcuxpresso/mcuxsdk-manifests) for the complete delivery of MCUXpresso SDK.

Introduction This branch contains unified IPv4 and IPv6 functionalities. Refer to the Getting started Guide (found here) for more details.

FreeRTOS-Plus-TCP Library FreeRTOS-Plus-TCP is a lightweight TCP/IP stack for FreeRTOS. It provides a familiar Berkeley sockets interface, making it as simple to use and learn as possible. FreeRTOS-Plus-TCP's features and RAM footprint are fully scalable, making FreeRTOS-Plus-TCP equally applicable to smaller lower throughput microcontrollers as well as larger higher throughput microprocessors.

This library has undergone static code analysis and checks for compliance with the MISRA coding standard. Any deviations from the MISRA C:2012 guidelines are documented under MISRA Deviations. The library is validated for memory safety and data structure invariance through the CBMC automated reasoning tool for the functions that parse data originating from the network. The library is also protocol tested using Maxwell protocol tester for both IPv4 and IPv6.

Getting started The easiest way to use the 4.0.0 version of FreeRTOS-Plus-TCP is to refer the Getting started Guide (found here) Another way is to start with the pre-configured demo application project (found in this directory). That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the FreeRTOS Kernel Quick Start Guide for detailed instructions and other useful links.

Additionally, for FreeRTOS-Plus-TCP source code organization refer to the Documentation, and API Reference.

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the FreeRTOS Community Support Forum. Please also refer to FAQ for frequently asked questions.

Also see the Submitting a bugs/feature request section of CONTRIBUTING.md for more details.

Note: All the remaining sections are generic and applies to all the versions from V3.0.0 onwards.

Upgrading to V3.0.0 and V3.1.0 In version 3.0.0 or 3.1.0, the folder structure of FreeRTOS-Plus-TCP has changed and the files have been broken down into smaller logically separated modules. This change makes the code more modular and conducive to unit-tests. FreeRTOS-Plus-TCP V3.0.0 improves the robustness, security, and modularity of the library. Version 3.0.0 adds comprehensive unit test coverage for all lines and branches of code and has undergone protocol testing, and penetration testing by AWS Security to reduce the exposure to security vulnerabilities. Additionally, the source files have been moved to a `source` directory. This change requires modification of any existing project(s) to include the modified source files and directories. There are examples on how to use the new files and directory structure. For an example based on the Xilinx Zynq-7000, use the code in this branch and follow these instructions to build and run the demo.

FreeRTOS-Plus-TCP V3.1.0 source code(.c .h) is part of the FreeRTOS 202210.00 LTS release.

Generating pre V3.0.0 folder structure for backward compatibility: If you wish to continue using a version earlier than V3.0.0 i.e. continue to use your existing source code organization, a script is provided to generate the folder structure similar to this.

Note: After running the script, while the .c files will have same names as the pre V3.0.0 source, the files in the `include` directory will have different names and the number of files will differ as well. This should, however, not pose any problems to most projects as projects generally include all files in a given directory.

Running the script to generate pre V3.0.0 folder structure: For running the script, you will need Python version > 3.7. You can download/install it from here.

Once python is downloaded and installed, you can verify the version from your terminal/command window by typing `python --version`.

To run the script, you should switch to the FreeRTOS-Plus-TCP directory that was created using the *Cloning this repository* step above. And then run `python <Path/to/the/script>/GenerateOriginalFiles.py`.

To consume FreeRTOS+TCP

Consume with CMake If using CMake, it is recommended to use this repository using FetchContent. Add the following into your project's main or a subdirectory's `CMakeLists.txt`:

- Define the source and version/tag you want to use:

```
FetchContent_Declare( freertos_plus_tcp
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git
  GIT_TAG        master #Note: Best practice to use specific git-hash or tagged version
  GIT_SUBMODULES "" # Don't grab any submodules since not latest
)
```

- Configure the FreeRTOS-Kernel and make it available
  - this particular example supports a native and cross-compiled build option.

```
set( FREERTOS_PLUS_FAT_DEV_SUPPORT OFF CACHE BOOL "" FORCE)
# Select the native compile PORT
set( FREERTOS_PLUS_FAT_PORT "POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  # Eg. Zynq 2019_3 version of port
  set(FREERTOS_PLUS_FAT_PORT "ZYNQ_2019_3" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_plus_tcp)
```

Consuming stand-alone This repository uses Git Submodules to bring in dependent components.

Note: If you download the ZIP file provided by GitHub UI, you will not get the contents of the submodules. (The ZIP file is also not a valid Git repository)

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel
```

Porting The porting guide is available on this page.

Repository structure This repository contains the FreeRTOS-Plus-TCP repository and a number of supplementary libraries for testing/PR Checks. Below is the breakdown of what each directory contains:

- tools
  - This directory contains the tools and related files (CMock/uncrustify) required to run tests/checks on the TCP source code.
- tests
  - This directory contains all the tests (unit tests and CBMC) and the dependencies (FreeRTOS-Kernel/Litani-port) the tests require.
- source/portable
  - This directory contains the portable files required to compile the FreeRTOS-Plus-TCP source code for different hardware/compilers.
- source/include
  - The include directory has all the 'core' header files of FreeRTOS-Plus-TCP source.
- source
  - This directory contains all the [.c] source files.

Note At this time it is recommended to use BufferAllocation_2.c in which case it is essential to use the heap_4.c memory allocation scheme. See memory management.

Kernel sources The FreeRTOS Kernel Source is in FreeRTOS/FreeRTOS-Kernel repository, and it is consumed by testing/PR checks as a submodule in this repository.

The version of the FreeRTOS Kernel Source in use could be accessed at ./test/FreeRTOS-Kernel directory.

CBMC The test/cbmc/proofs directory contains CBMC proofs.

To learn more about CBMC and proofs specifically, review the training material here.

In order to run these proofs you will need to install CBMC and other tools by following the instructions here.