



MCUXpresso SDK Documentation

Release 25.06.00



NXP
Jun 26, 2025



Table of contents

1	FRDM-RW612	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	35
1.3.1	Getting Started with MCUXpresso SDK Repository	35
1.4	Release Notes	48
1.4.1	MCUXpresso SDK Release Notes	48
1.5	ChangeLog	55
1.5.1	MCUXpresso SDK Changelog	55
1.6	Driver API Reference Manual	111
1.7	Middleware Documentation	111
1.7.1	Wireless Connectivity Framework	111
1.7.2	MCU Boot	111
1.7.3	FreeMASTER	111
1.7.4	AWS IoT	111
1.7.5	NXP Wi-Fi	111
1.7.6	FreeRTOS	111
1.7.7	Wireless EdgeFast Bluetooth PAL	112
1.7.8	lwIP	112
1.7.9	File systemFatfs	112
2	RW612	113
2.1	ACOMP: Analog Comparator	113
2.2	ADC: Analog Digital Converter	122
2.3	CACHE: CACHE Memory Controller	138
2.4	CDOG	142
2.5	Clock Driver	146
2.6	CRC: Cyclic Redundancy Check Driver	164
2.7	CTIMER: Standard counter/timers	167
2.8	DAC: Digital Analog Converter	176
2.9	DMA: Direct Memory Access Controller Driver	187
2.10	DMIC: Digital Microphone	204
2.11	DMIC DMA Driver	204
2.12	DMIC Driver	206
2.13	ENET: Ethernet MAC Driver	215
2.14	FLEXCOMM: FLEXCOMM Driver	246
2.15	FLEXCOMM Driver	246
2.16	FLEXSPI: Flexible Serial Peripheral Interface Driver	247
2.17	FLEXSPI DMA Driver	264
2.18	FMEAS: Frequency Measure Driver	266
2.19	GDMA: General DMA(GDMA) Driver	267
2.20	I2C: Inter-Integrated Circuit Driver	276
2.21	I2C DMA Driver	276
2.22	I2C Driver	278
2.23	I2C Master Driver	281

2.24	I2C Slave Driver	290
2.25	I2S: I2S Driver	300
2.26	I2S_BRIDGE: I2S bridging and signal sharing configuration	300
2.27	I2S DMA Driver	302
2.28	I2S Driver	305
2.29	IMU: Inter CPU Messaging Unit	314
2.30	INPUTMUX: Input Multiplexing Driver	321
2.31	IO_MUX Driver	335
2.32	IPED Driver	344
2.33	Intrusion and Tamper Response Controller	348
2.34	ITRC	348
2.35	Common Driver	351
2.36	LCDIC Driver	363
2.37	LCDIC DMA Driver	382
2.38	LCDIC: LCD Interface Controller	384
2.39	GPIO: General Purpose I/O	384
2.40	MRT: Multi-Rate Timer	387
2.41	This type defines status return values used by NBOOT functions that are not easily disturbed by Fault Attacks	392
2.42	OCOTP Driver	393
2.43	OSTIMER: OS Event Timer Driver	394
2.44	PINT: Pin Interrupt and Pattern Match Driver	398
2.45	Power Driver	406
2.46	POWERQUAD: PowerQuad hardware accelerator	414
2.47	Reset Driver	444
2.48	RTC: Real Time Clock	448
2.49	Sbloader	454
2.50	SCTimer: SCTimer/PWM (SCT)	458
2.51	Sdioslv_sdu_driver	474
2.52	Smart Card	487
2.53	Smart Card PHY Driver	495
2.54	Smart Card PHY USIM W	496
2.55	Smart Card USIM Driver	496
2.56	SPI: Serial Peripheral Interface Driver	499
2.57	SPI DMA Driver	499
2.58	SPI Driver	503
2.59	TRNG: True Random Number Generator	511
2.60	USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver	516
2.61	USART DMA Driver	516
2.62	USART Driver	519
2.63	UTICK: MicroTick Timer Driver	536
2.64	WWDT: Windowed Watchdog Timer Driver	537
3	Middleware	541
3.1	Boot	541
3.1.1	MCUXpresso SDK : mcuxsdk-middleware-mcuboot_opensource	541
3.1.2	MCUboot	542
3.2	Motor Control	543
3.2.1	FreeMASTER	543
3.3	Wireless	580
3.3.1	NXP Wireless Framework and Stacks	580
3.3.2	Wi-Fi	642
4	RTOS	759
4.1	FreeRTOS	759
4.1.1	FreeRTOS kernel	759
4.1.2	FreeRTOS drivers	759
4.1.3	backoffalgorithm	759

4.1.4	corehttp	759
4.1.5	corejson	759
4.1.6	coremqtt	760
4.1.7	coremqtt-agent	760
4.1.8	corepkcs11	760
4.1.9	freertos-plus-tcp	760

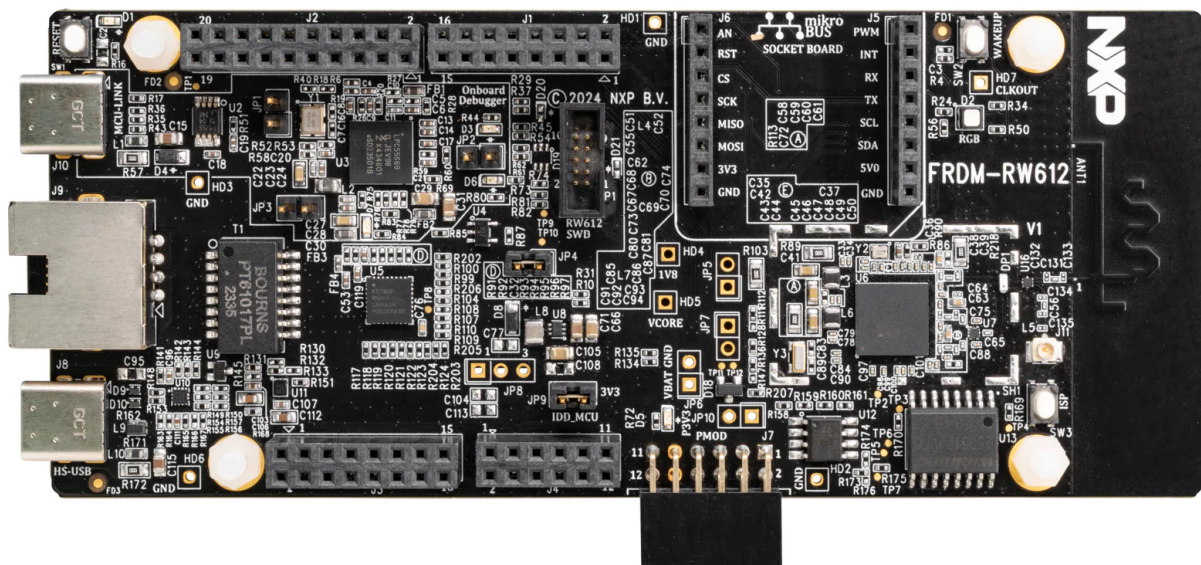
This documentation contains information specific to the frdmrw612 board.

Chapter 1

FRDM-RW612

1.1 Overview

FRDM-RW612 is a compact and scalable development board for rapid prototyping of the RW61x series of Wi-Fi 6 + Bluetooth Low Energy + 802.15.4 tri-radio wireless MCUs. It offers easy access to the MCU's I/Os and peripherals, integrated open-standard serial interfaces, external flash memory and on-board MCU-Link debugger.



MCU device and part on board is shown below:

- Device: RW612
- PartNumber: RW612ETA2I

1.2 Getting Started with MCUXpresso SDK Package

1.2.1 Getting Started with Package

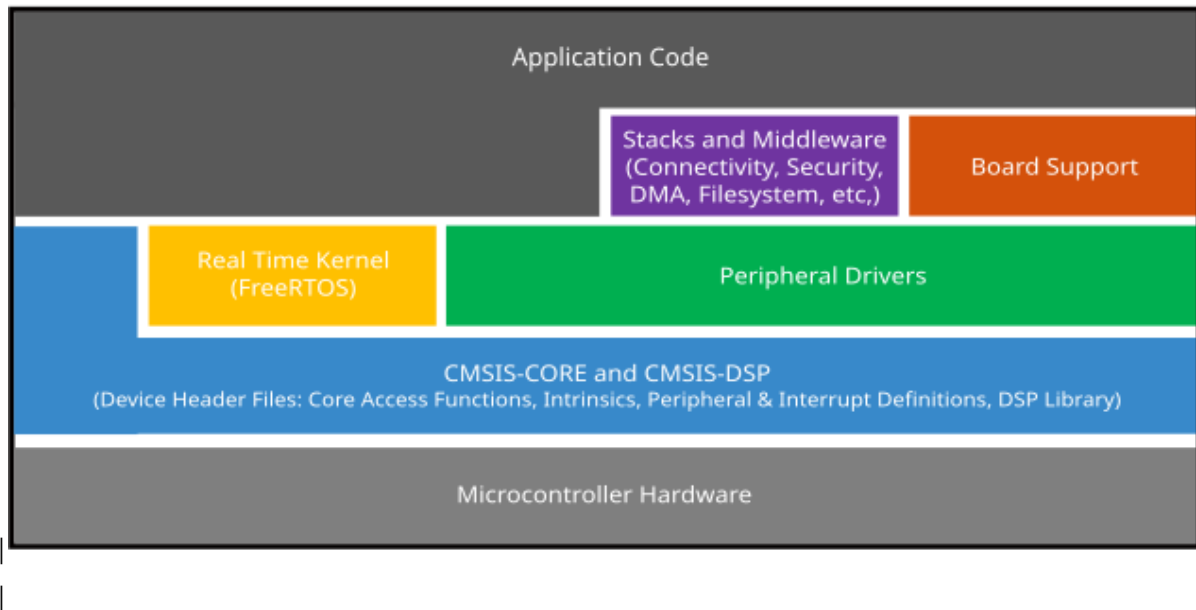
Overview

The NXP MCUXpresso software and tools offer comprehensive development solutions designed to optimize, ease and help accelerate embedded system development of applications based on

general purpose, crossover and Bluetooth-enabled MCUs from NXP. The MCUXpresso SDK includes a flexible set of peripheral drivers designed to speed up and simplify development of embedded applications. Along with the peripheral drivers, the MCUXpresso SDK provides an extensive and rich set of example applications covering everything from basic peripheral use case examples to full demo applications. The MCUXpresso SDK contains optional RTOS integrations such as FreeRTOS and Azure RTOS, a USB host and device stack, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes for FRDM-RW612* (document MCUXSDKFRDMRW612RN).

For more details about MCUXpresso SDK, see [MCUXpresso Software Development Kit \(SDK\)](#).



MCUXpresso SDK board support package folders

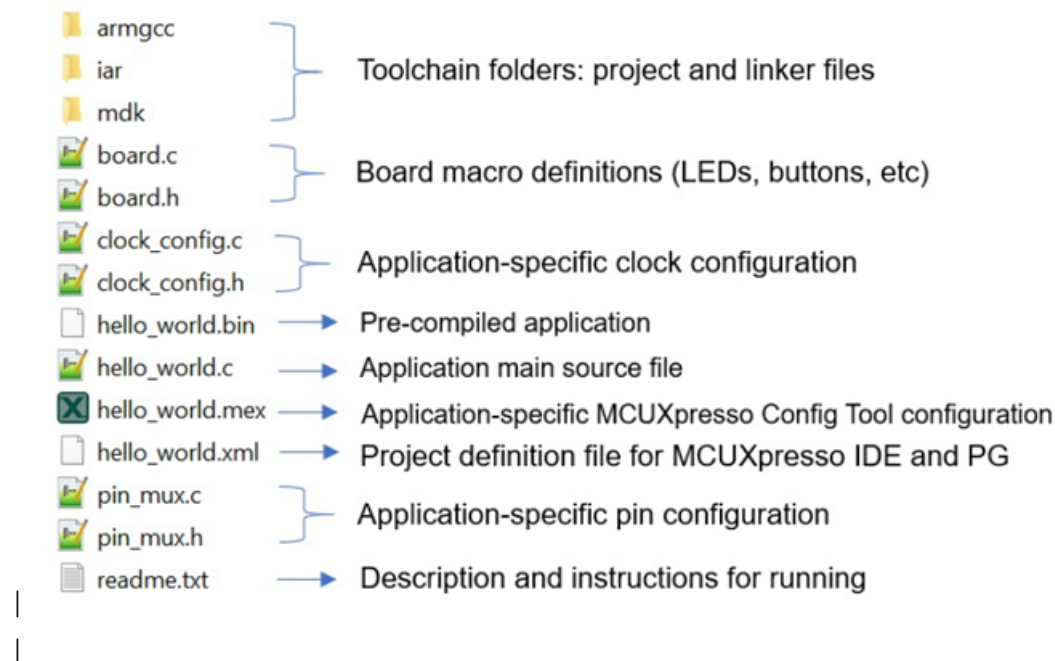
MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, i.mxrt EVK boards, and LPCXpresso boards. Board support packages are found inside the top level boards folder and each supported board has its own folder (an MCUXpresso SDK package can support multiple boards). Within each `<board_name>` folder, there are various sub-folders to classify the type of examples it contain. These include (but are not limited to):

- `demo_apps`: Full-featured applications that highlight key functionality and use cases of the target MCU. These applications typically use multiple MCU peripherals and may leverage stacks and middleware.
- `driver_examples`: Simple applications that show how to use the MCUXpresso SDK's peripheral drivers for a single use case. These applications typically only use a single peripheral but there are cases where multiple peripherals are used (for example, SPI conversion using DMA).
- `rtos_examples`: Basic FreeRTOS OS examples that show the use of various RTOS objects (semaphores, queues, and so on) and interfaces with the MCUXpresso SDK's RTOS drivers

Example application structure This section describes how the various types of example applications interact with the other components in the MCUXpresso SDK. To get a comprehensive understanding of all MCUXpresso SDK components and folder structure, see *MCUXpresso SDK API Reference Manual*.

Each `<board_name>` folder in the boards directory contains a comprehensive set of examples that are relevant to that specific piece of hardware. Although we use the `hello_world` example (part of the `demo_apps` folder), the same general rules apply to any type of example in the `<board_name>` folder.

In the `hello_world` application folder you see the following contents:



All files in the application folder are specific to that example, so it is easy to copy and paste an existing example to start developing a custom application based on a project provided in the MCUXpresso SDK.

Parent topic: [MCUXpresso SDK board support package folders](#)

Locating example application source files When opening an example application in any of the supported IDEs, a variety of source files are referenced. The MCUXpresso SDK devices folder is the central component to all example applications. It means the examples reference the same source files and, if one of these files is modified, it could potentially impact the behavior of other examples.

The main areas of the MCUXpresso SDK tree used in all example applications are:

- `devices/<device_name>`: The device's CMSIS header file, MCUXpresso SDK feature file and a few other files
- `devices/<device_name>/drivers`: All of the peripheral drivers for your specific MCU
- `devices/<device_name>/<tool_name>`: Toolchain-specific startup code, including vector table definitions
- `devices/<device_name>/utilities`: Items such as the debug console that are used by many of the example applications

For examples containing an RTOS, there are references to the appropriate source code. RTOSes are in the `rtos` folder. The core files of each of these are shared, so modifying one could have potential impacts on other projects that depend on that file.

Parent topic: [MCUXpresso SDK board support package folders](#)

Run a demo using MCUXpresso IDE

Note: Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

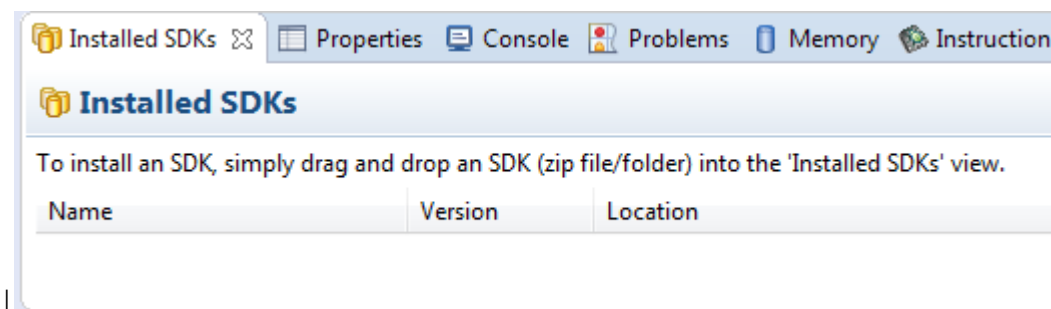
This section describes the steps required to configure MCUXpresso IDE to build, run, and debug example applications. The `hello_world` demo application targeted for the FRDM-RW612 hardware platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

Select the workspace location Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the workspace. The location of the workspace can be anywhere, but it is recommended that the workspace be located outside of the MCUXpresso SDK tree.

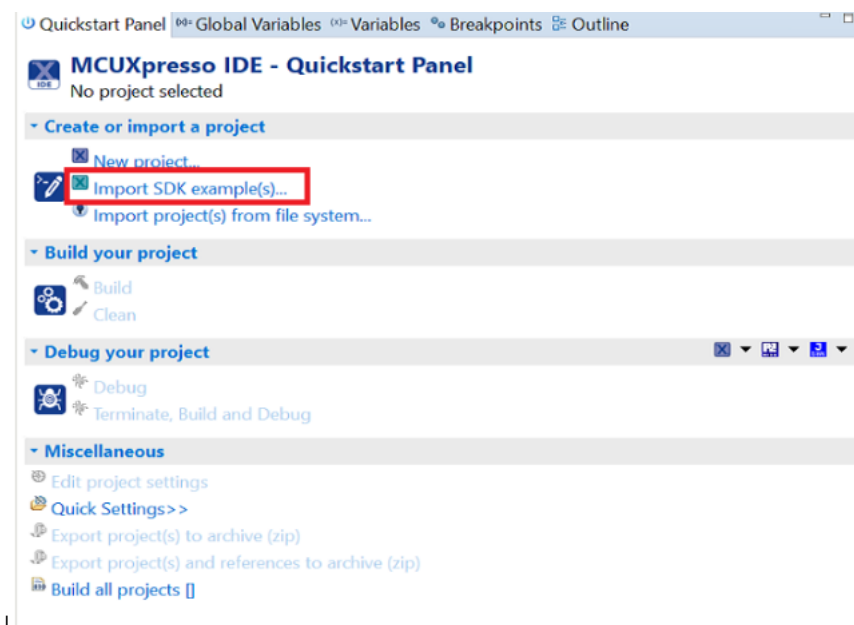
Parent topic: [Run a demo using MCUXpresso IDE](#)

Build an example application To build an example application, follow these steps.

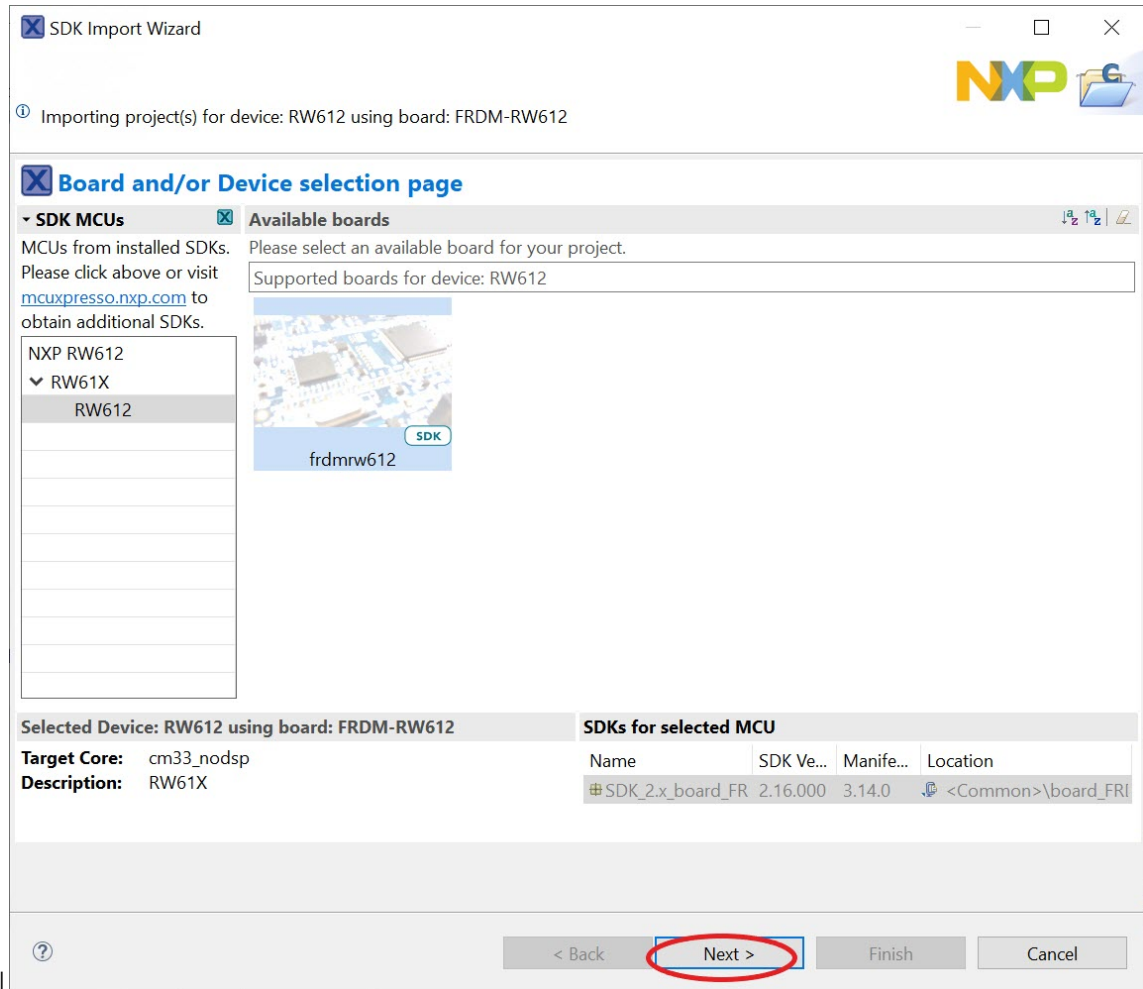
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



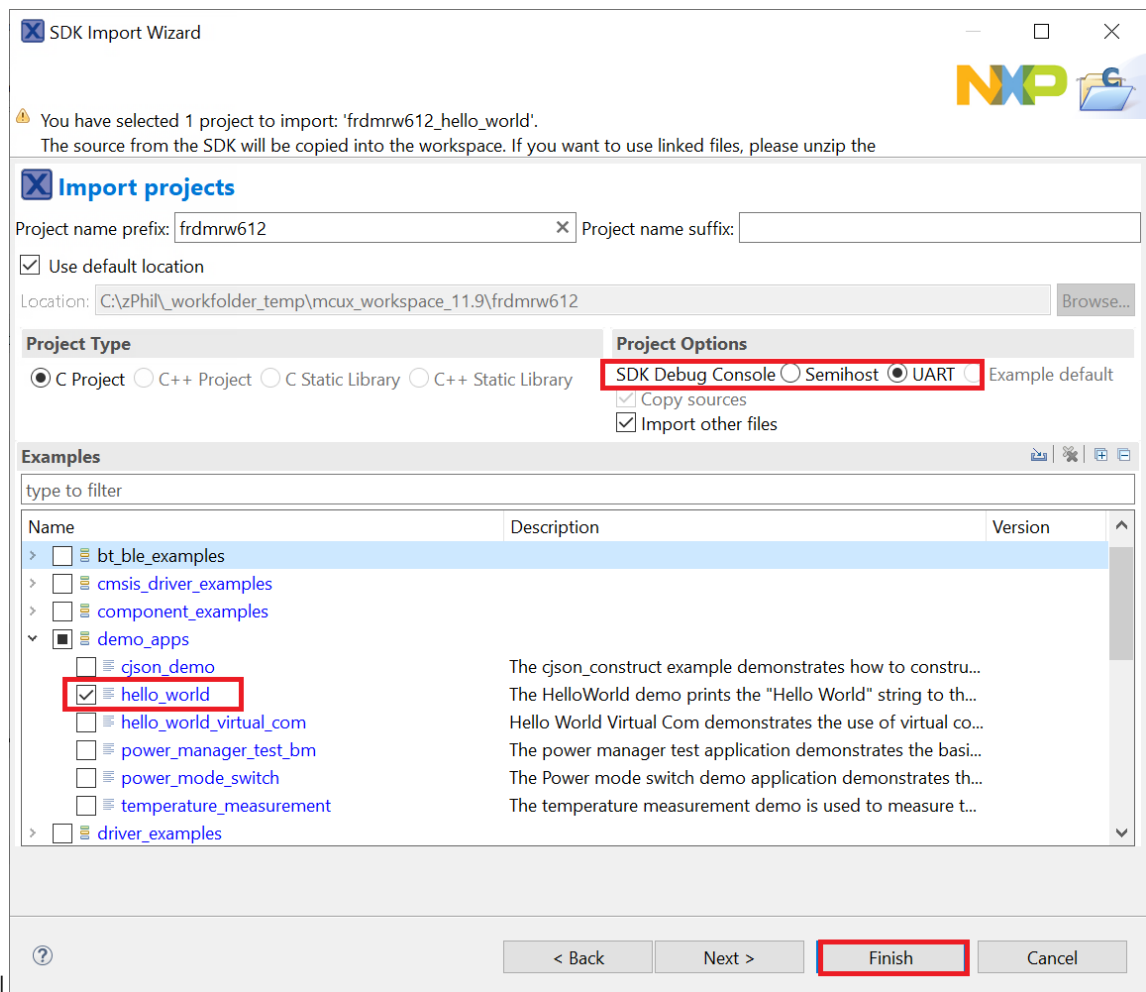
2. On the **Quickstart Panel**, click **Import SDK example(s)...**



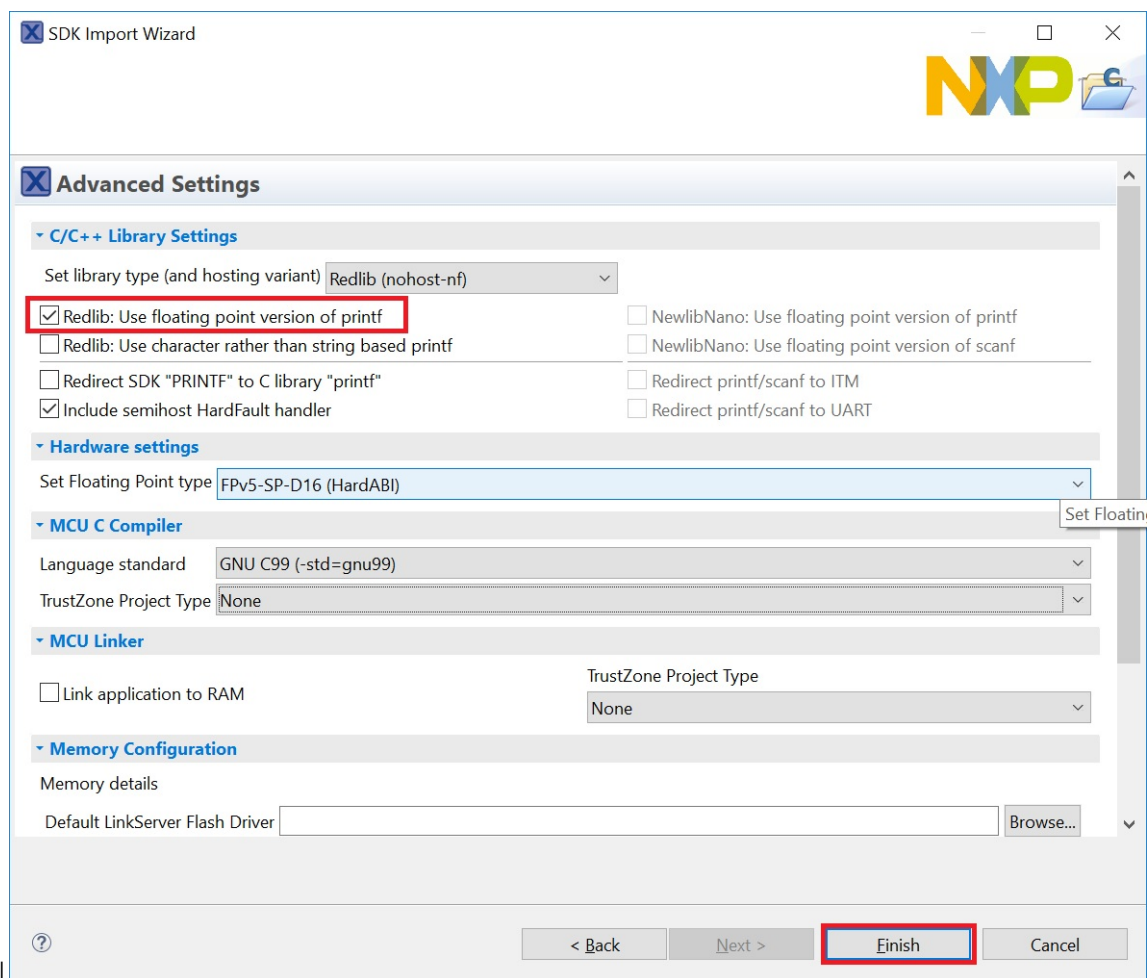
3. In the window that appears, expand the **RW61X** folder and select **RW612**.
4. Select **frdmrw612** and click **Next**.



5. Expand the **demo_apps** folder and select **hello_world** . Then, click **Next** .



6. Ensure **Redlib: Use floating point version of printf** is selected if the example prints floating point numbers on the terminal. Otherwise, it is not necessary to select this option. Then, click **Finish**.



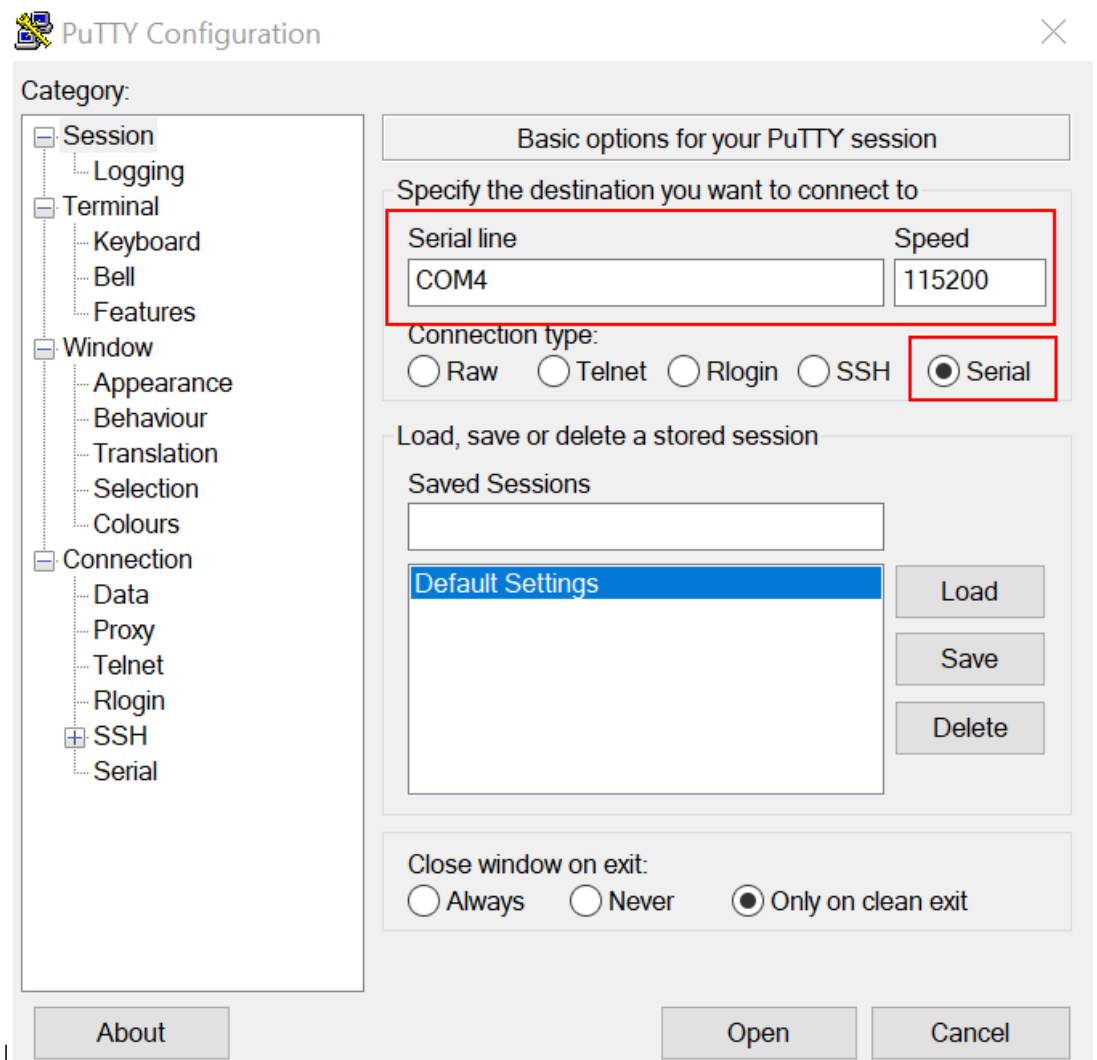
Parent topic: [Run a demo using MCUXpresso IDE](#)

Run an example application For more information on debug probe support in the MCUXpresso IDE, see community.nxp.com.

To download and run the application, perform the following steps:

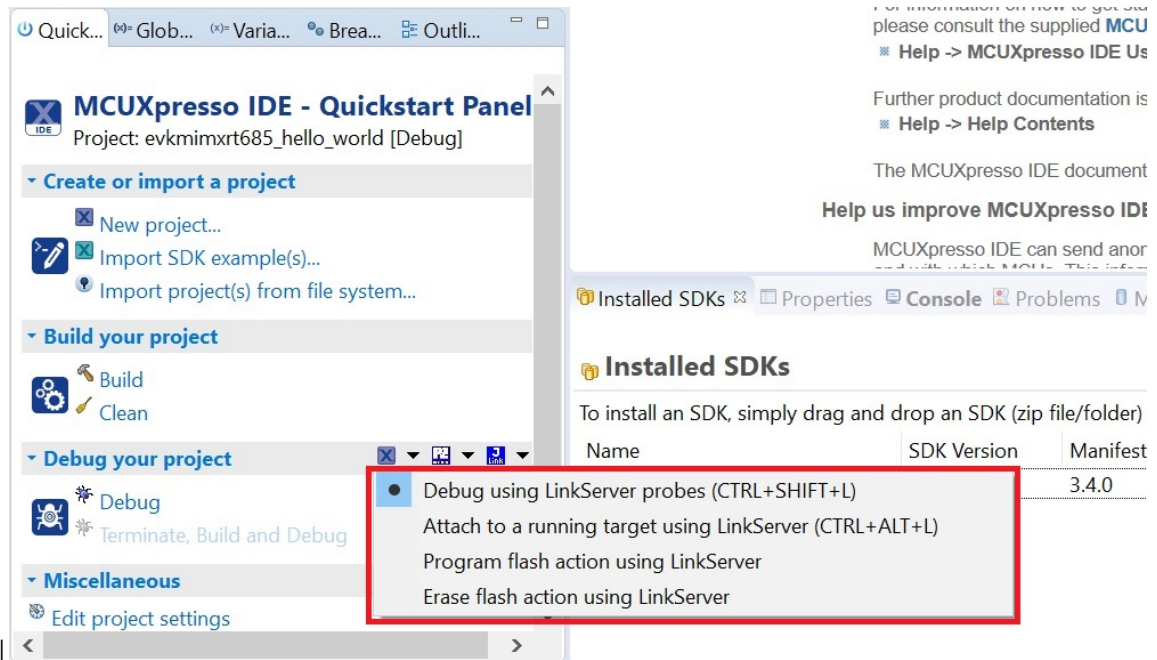
1. See the table in [Default debug interfaces](#) to determine the debug interface that comes loaded on your specific hardware platform.
 - For boards with CMSIS-DAP/mbd/DAPLink interfaces, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
 - For boards with a P&E Micro interface, see [PE micro](#) to download and install the P&E Micro Hardware Interface Drivers package.
2. Connect the development platform to your PC via a USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
 2. No parity

3. 8 data bits

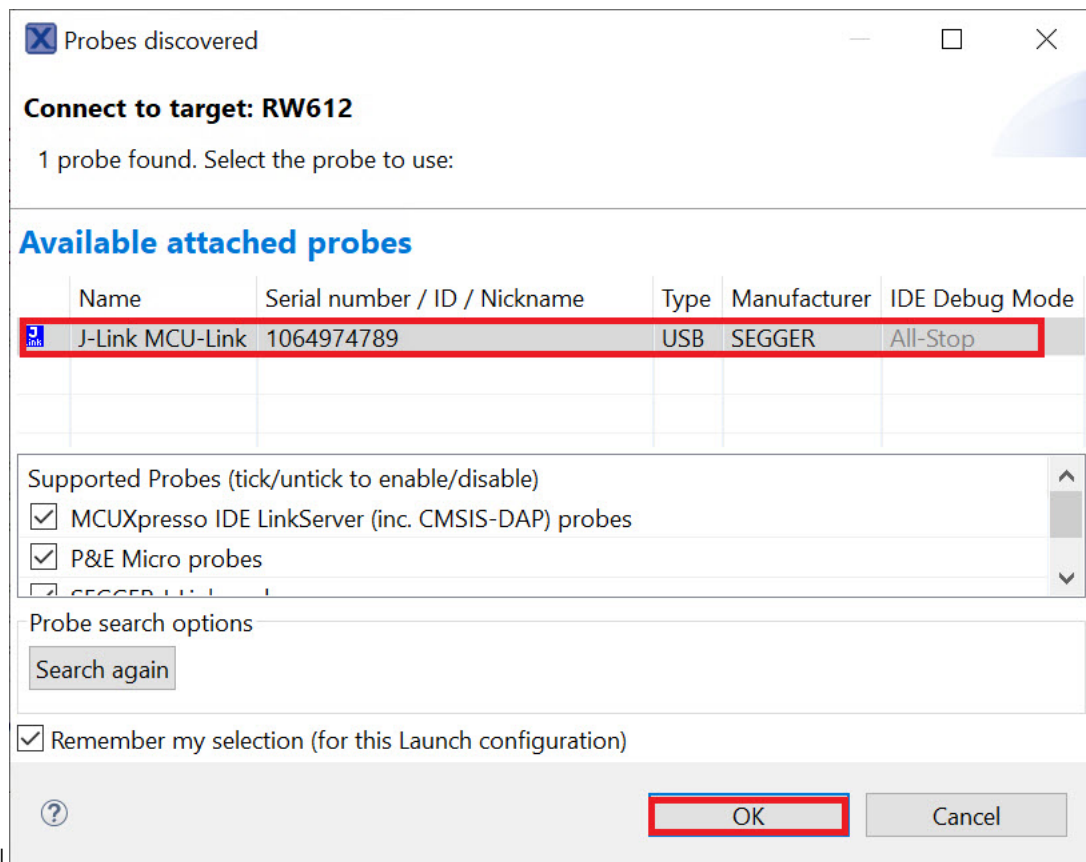


4. 1 stop bit

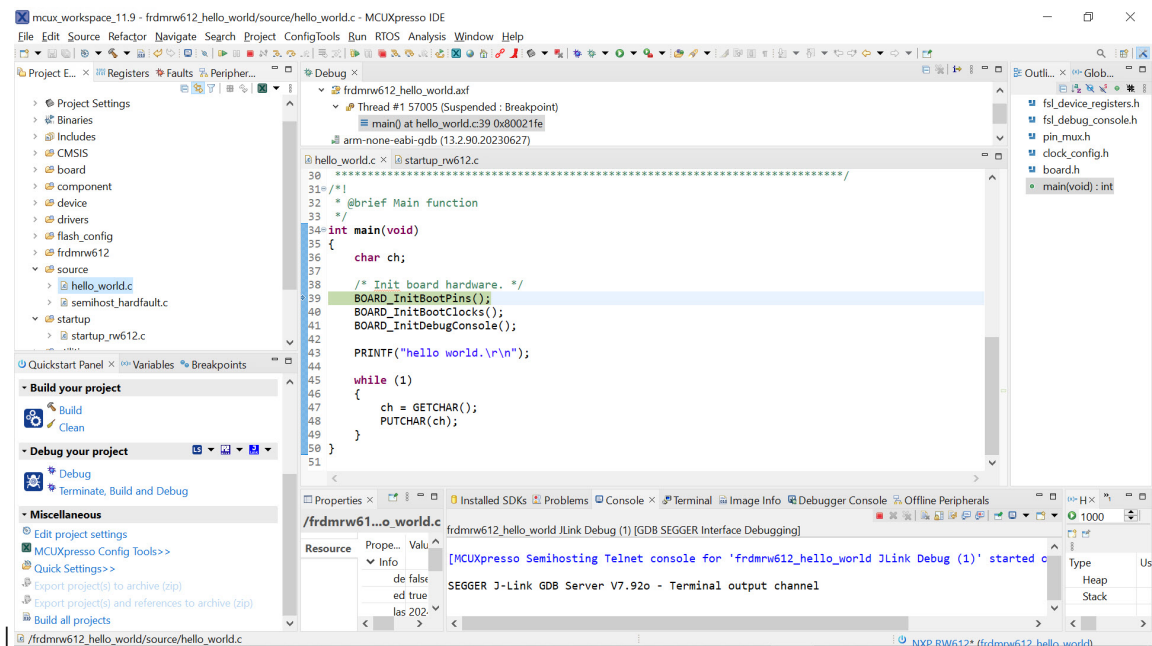
- On the **Quickstart Panel**, click on **Debug** frdmrw612_hello_world [Debug] to launch the debug session.



- The first time you debug a project, the **Debug Emulator Selection** dialog is displayed, showing all supported probes that are attached to your computer. Select the probe through which you want to debug and click **OK**. (For any future debug sessions, the stored probe selection is automatically used, unless the probe cannot be found.)



- The application is downloaded to the target and automatically runs to `main()`.



7. Start the application by clicking **Resume**.



The hello_world application is now running and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.



Parent topic: [Run a demo using MCUXpresso IDE](#)

Run a demo application using IAR

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

Note: IAR Embedded Workbench for Arm version 9.50.2 is used in the following example. The IAR toolchain should correspond to the latest supported version, as described in the *MCUXpresso SDK Release Notes* (document ID: MCUXSDKRN). IAR/Segger still does not support RW61x well. Therefore, contact the support team to get `iar_support_patch_frdmrw612_0.zip` and install the IAR patch before opening the IAR project.

Build an example application Do the following steps to build the `hello_world` example application.

1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/iar
```

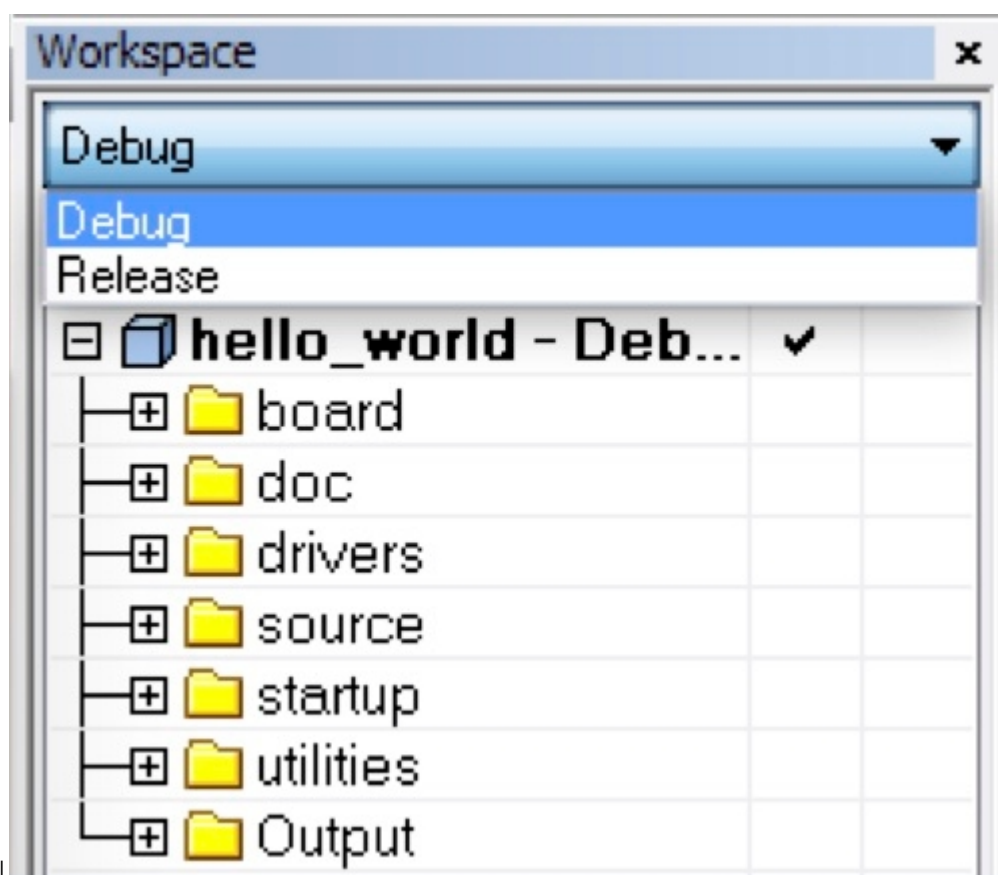
Using the FRDM-RW612 hardware platform as an example, the `hello_world` workspace is located in:

```
<install_dir>/boards/frdmrw612/demo_apps/hello_world/iar/hello_world.eww
```

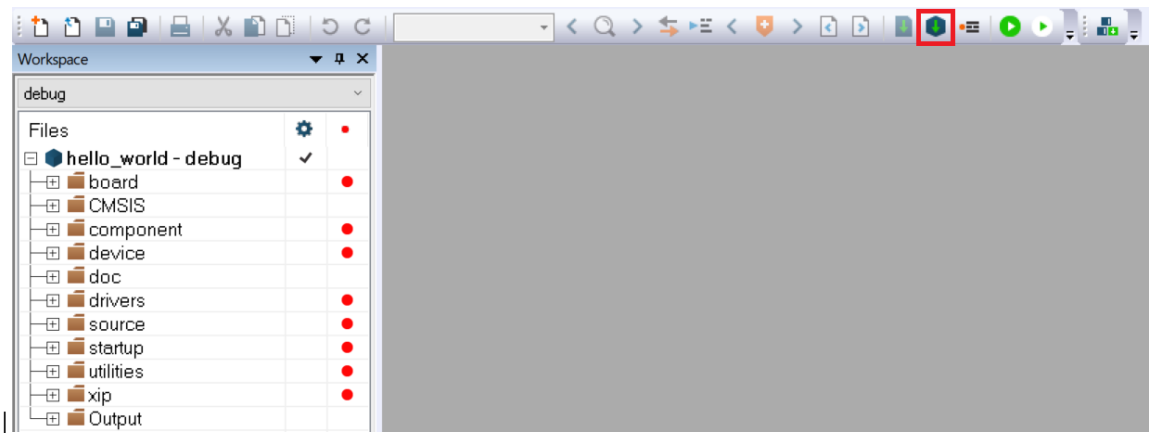
Other example applications may have additional folders in their path.

2. Select the desired build target from the drop-down menu.

For this example, select **hello_world – debug**.



3. To build the demo application, click **Make**, highlighted in red in Figure 2.

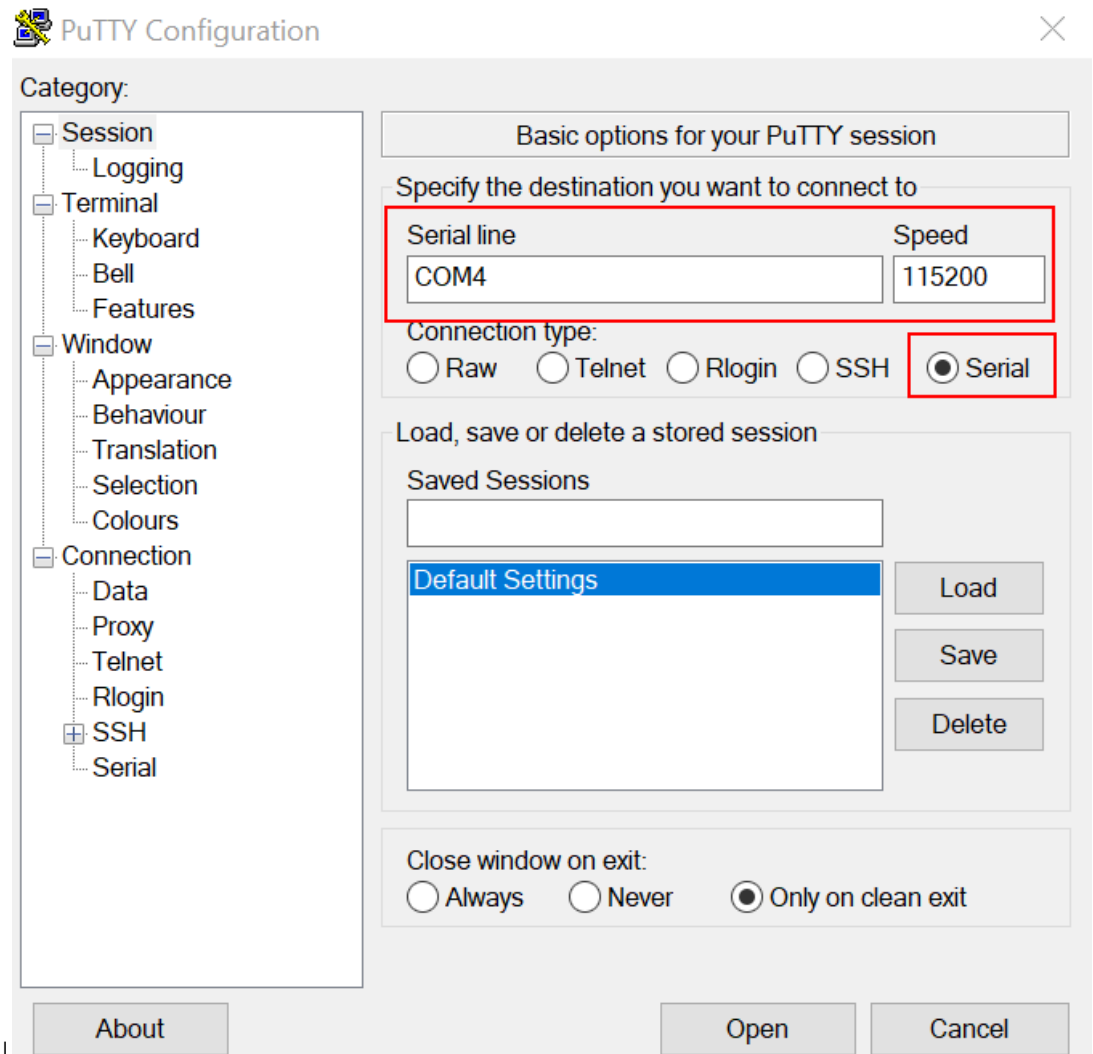


- |
4. The build completes without errors.

Parent topic: [Run a demo application using IAR](#)

Run an example application To download and run the application, perform these steps:

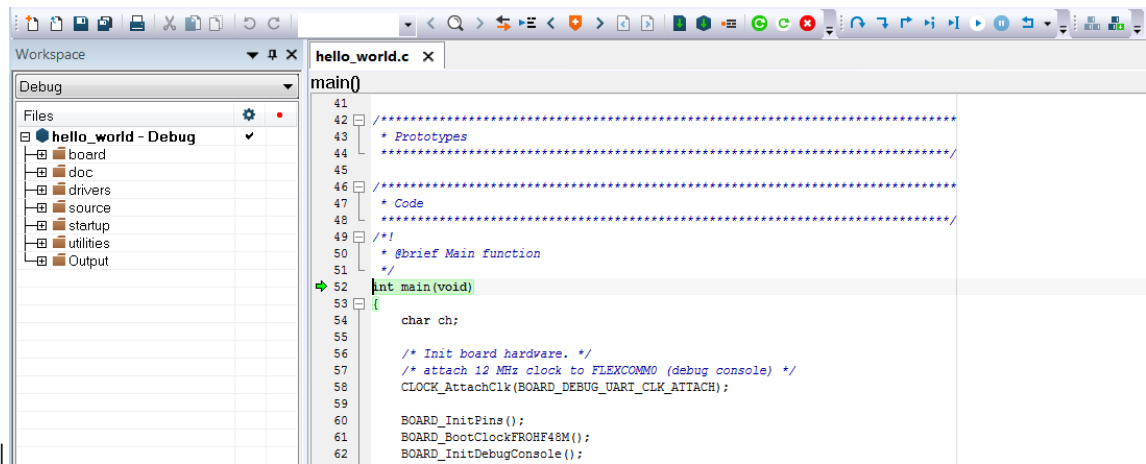
1. See the table in [Default debug interfaces](#) to determine the debug interface that comes loaded on your specific hardware platform.
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits



4. In IAR, click the **Download and Debug** button to download the application to the target.



5. The application is then downloaded to the target and automatically runs to the `main()` function.



6. Run the code by clicking the **Go** button.



7. The `hello_world` application is now running and a banner is displayed on the terminal. If it does not appear, check your terminal settings and connections.



Parent topic: [Run a demo application using IAR](#)

IAR RAM debugging notes In the SDK examples, there are many build configurations. For example, debug, release, flash_debug, and flash_release. The debug/release configuration demonstrates the application running in RAM, while flash_debug/flash_release runs in the flash (XIP).

XIP debugging is clean with the “*boot from flash*” setting on the board because the debugging has the same sequence as a normal boot.

The debugging steps are as follows:

1. The debugger downloads the image to the flash.
2. The debugger resets the device and boots from ROM.
3. The debugger halts the device and waits for the command from the user.

However, with RAM debugging with the *boot from flash* setting on the board, the situation is complicated.

1. The debugger downloads the image to the RAM.
2. The debugger resets the device and boots it from ROM.

Note: The image downloaded for debugging and the image running from the flash are different. The images are different because the image in flash might be written by the previous debugging.

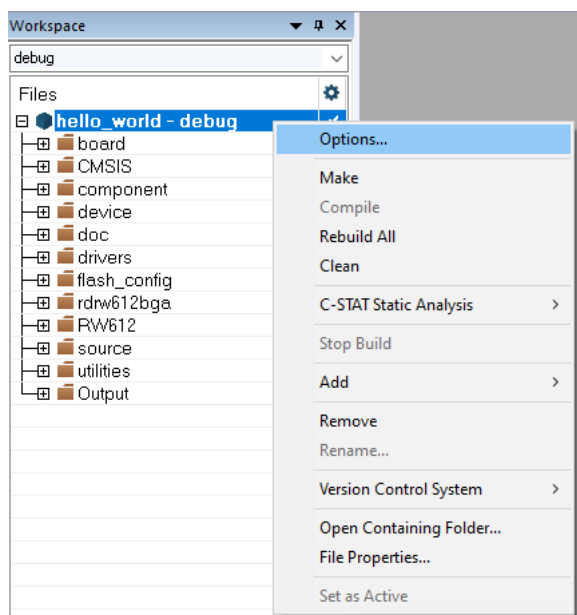
3. The debugger halts the device and sets the PC register value to the entry address of the downloaded image.

The problem here is that step 2 might overwrite the RAM written in step 1. The other issue is that the hardware status is undetermined when PC is set to the start of the RAM application.

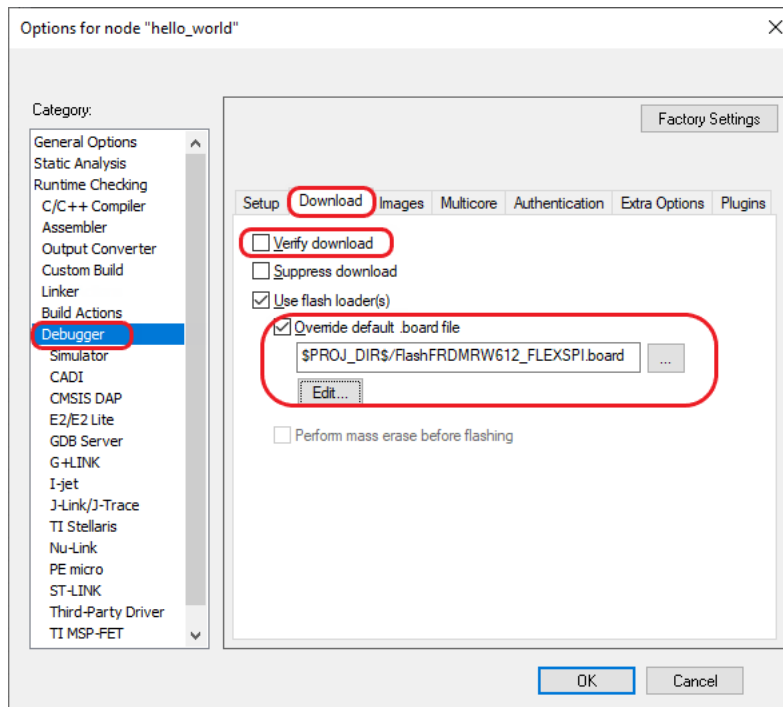
To avoid the RAM debugging issue, MCUX IDE/armgcc/MDK provides a plain load feature. To get a clean debugging status, the RAM image is downloaded to the flash.

IAR needs a special setting to achieve the plain load feature as follows:

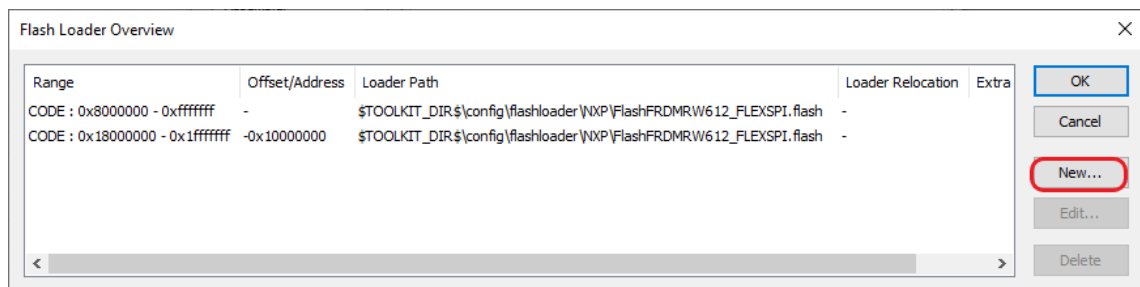
1. Right-click the project and select **Options**.



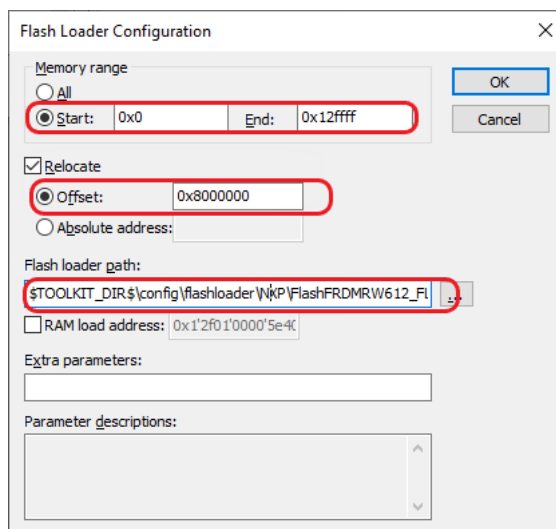
2. Select **Debugger > Download** in the ****Options for node "<project_name>"**** dialog box.
3. Clear the **Verify download** checkbox because the download address is different from the real address.
4. Select the **Override default.board file** option.
5. Copy the default *.board file to the IAR workspace folder.
6. Change the *.board file path.
7. Click the **Edit** button.



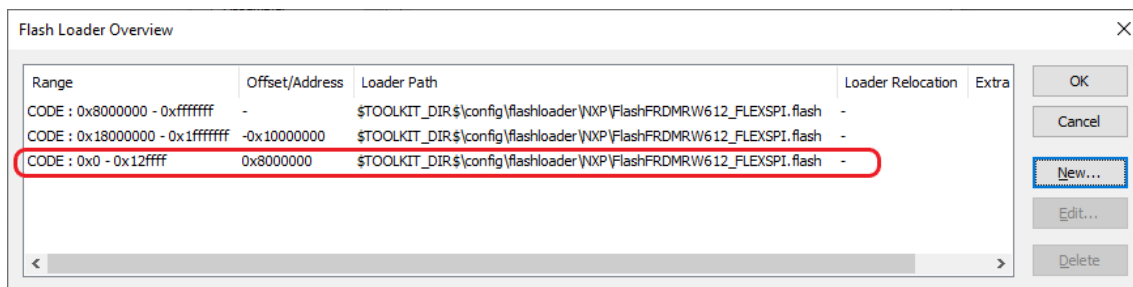
8. Create a range for RAM.



9. Set the memory range, relocate offset, and flashloader path.



The new RAM range appears in the flashloader overview.



- Follow the instructions in Run an example application and now it is safe to debug the RAM application. As a result, the downloaded RAM application can also boot on reset.

Parent topic: [Run a demo application using IAR](#)

Run a demo using Arm GCC

This section describes the steps to configure the command-line Arm GCC tools to build, run, and debug demo applications and necessary driver libraries provided in the MCUXpresso SDK. The `hello_world` demo application is targeted for the FRDM-RW612 hardware platform which is used as an example.

Note: ARMGCC version 8-2019-q3 is used as an example in this document. The latest GCC version for this package is as described in the *MCUXpresso SDK Release Notes Supporting FRDM-RW612* (document MCUXSDKFRDMRW612RN). IAR/Segger officially does not support RW61x. Therefore, contact the support team to get `iar_support_patch_frdmrw612_0.zip` and install the JLink patch before debugging with gdb.

Set up toolchain This section contains the steps to install the necessary components required to build and run an MCUXpresso SDK demo application with the Arm GCC toolchain, as supported by the MCUXpresso SDK. There are many ways to use Arm GCC tools, but this example focuses on a Windows operating system environment.

Install GCC Arm Embedded tool chain Download and run the installer from GNU Arm Embedded Toolchain. This is the actual toolset (in other words, compiler, linker, and so on). The GCC toolchain should correspond to the latest supported version, as described in *MCUXpresso SDK Release Notes*.

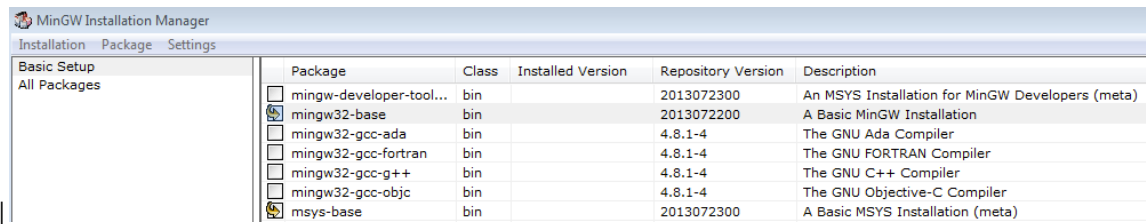
Parent topic: Set up toolchain

Install MinGW (only required on Windows OS) The Minimalist GNU for Windows (MinGW) development tools provide a set of tools that are not dependent on third-party C-Runtime DLLs (such as Cygwin). The build environment used by the MCUXpresso SDK does not use the MinGW build tools, but does leverage the base install of both MinGW and MSYS. MSYS provides a basic shell with a Unix-like interface and tools.

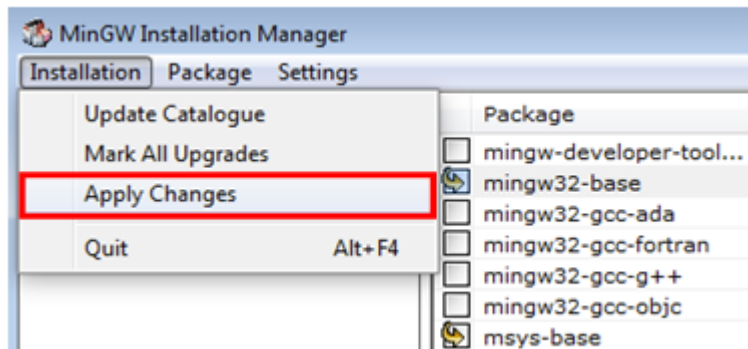
- Download the latest MinGW mingw-get-setup installer from [MinGW](#).
- Run the installer. The recommended installation path is `C:\MinGW`, however, you may install to any location.

Note: The installation path cannot contain any spaces.

- Ensure that the **mingw32-base** and **msys-base** are selected under **Basic Setup**.



4. In the **Installation** menu, click **Apply Changes** and follow the remaining instructions to complete the installation.

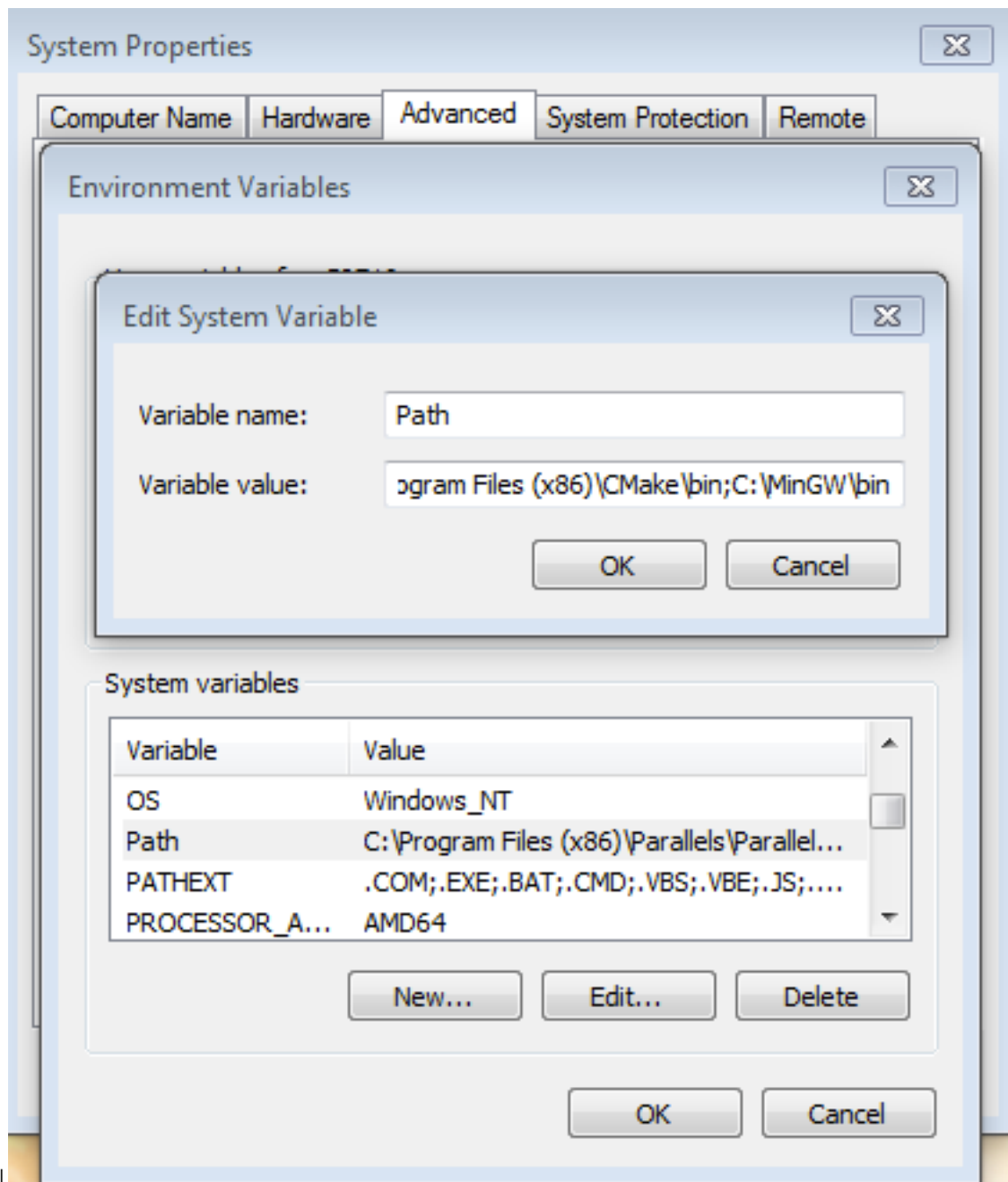


5. Add the appropriate item to the Windows operating system path environment variable. It can be found under **Control Panel->System and Security->System->Advanced System Settings** in the **Environment Variables...** section. The path is:

```
<mingw_install_dir>\bin
```

Assuming the default installation path, C:\MinGW, an example is shown below. If the path is not set correctly, the toolchain will not work.

Note: If you have C:\MinGW\msys\x.x\bin in your PATH variable (as required by Kinetis SDK 1.0.0), remove it to ensure that the new GCC build system works correctly.



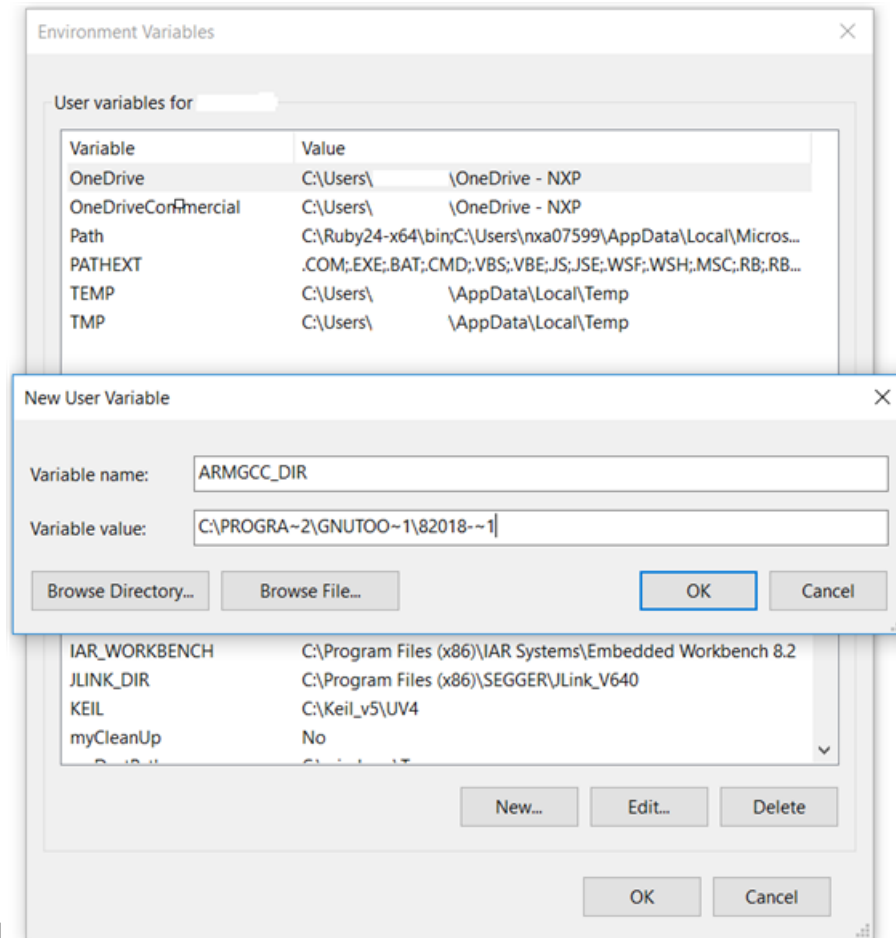
Parent topic:Set up toolchain

Add a new system environment variable for ARMGCC_DIR Create a new *system* environment variable and name it as `ARMGCC_DIR`. The value of this variable should point to the Arm GCC Embedded tool chain installation path. For this example, the path is:

See the installation folder of the GNU Arm GCC Embedded tools for the exact path name of your installation.

Short path should be used for path setting, you could convert the path to short path by running command for %I in (.) do echo %~sI in above path.

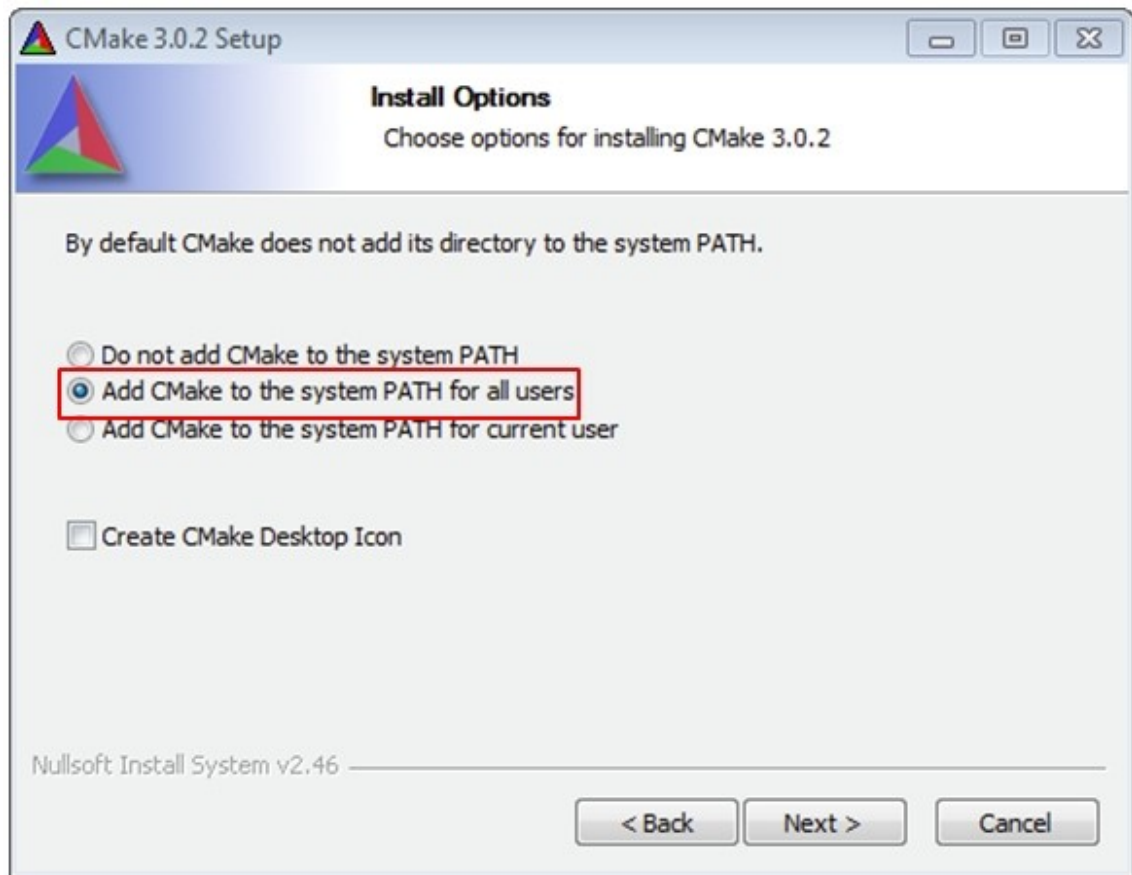
```
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>for %I in (.) do echo %~sI
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>echo C:\PROGRA~2\GNUTOO~1\82018~1
C:\PROGRA~2\GNUTOO~1\82018~1
```



Parent topic: Set up toolchain

Install CMake

1. Download CMake 3.0.x from www.cmake.org/cmake/resources/software.html.
2. Install CMake, ensuring that the option **Add CMake to system PATH** is selected when installing. The user chooses to select whether it is installed into the PATH for all users or just the current user. In this example, it is installed for all users.



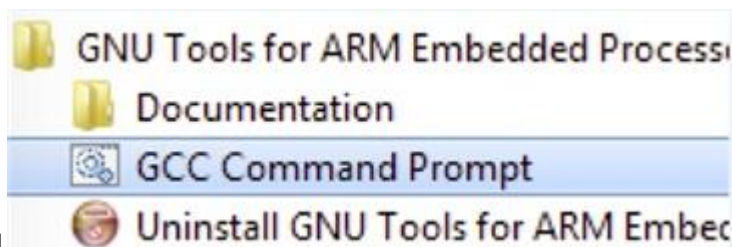
3. Follow the remaining instructions of the installer.
4. You may need to reboot your system for the PATH changes to take effect.
5. Make sure `sh.exe` is not in the Environment Variable PATH. This is a limitation of `mingw32-make`.

Parent topic:Set up toolchain

Parent topic:[Run a demo using Arm GCC](#)

Build an example application To build an example application, follow these steps.

1. Open a GCC Arm Embedded tool chain command window. To launch the window, from the Windows operating system **Start** menu, go to **Programs > GNU Tools Arm Embedded <version>** and select **GCC Command Prompt**.



2. Change the directory to the example application project directory which has a path similar to the following:

```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc
```

For this example, the exact path is:

```
<install_dir>/examples/frdmrw612/demo_apps/hello_world/armgcc
```

Note: To change directories, use the `cd` command.

3. Type **build_debug.bat** on the command line or double click on **build_debug.bat** file in Windows Explorer to build it. The output is as shown in Figure 2.

```

C:\ Command Prompt
[ 91%] Building C object CMakeFiles/hello_world.elf.dir/C:/SDK_2_16_000_RW612/devices/RW612/utiliti
es/debug_console_lite/fsl_assert.c.obj
[ 94%] Building C object CMakeFiles/hello_world.elf.dir/C:/SDK_2_16_000_RW612/devices/RW612/utiliti
es/debug_console_lite/fsl_debug_console.c.obj
[ 97%] Building C object CMakeFiles/hello_world.elf.dir/C:/SDK_2_16_000_RW612/devices/RW612/utiliti
es/str/fsl_str.c.obj
[100%] Linking C executable debug\hello_world.elf
Memory region      Used Size  Region Size  %age Used
m_flash:           31576 B      2 MB         1.51%
m_interrupts:       640 B       640 B       100.00%
m_text:            26840 B     781696 B      3.43%
m_data:             2504 B      448 KB        0.55%
[100%] Built target hello_world.elf

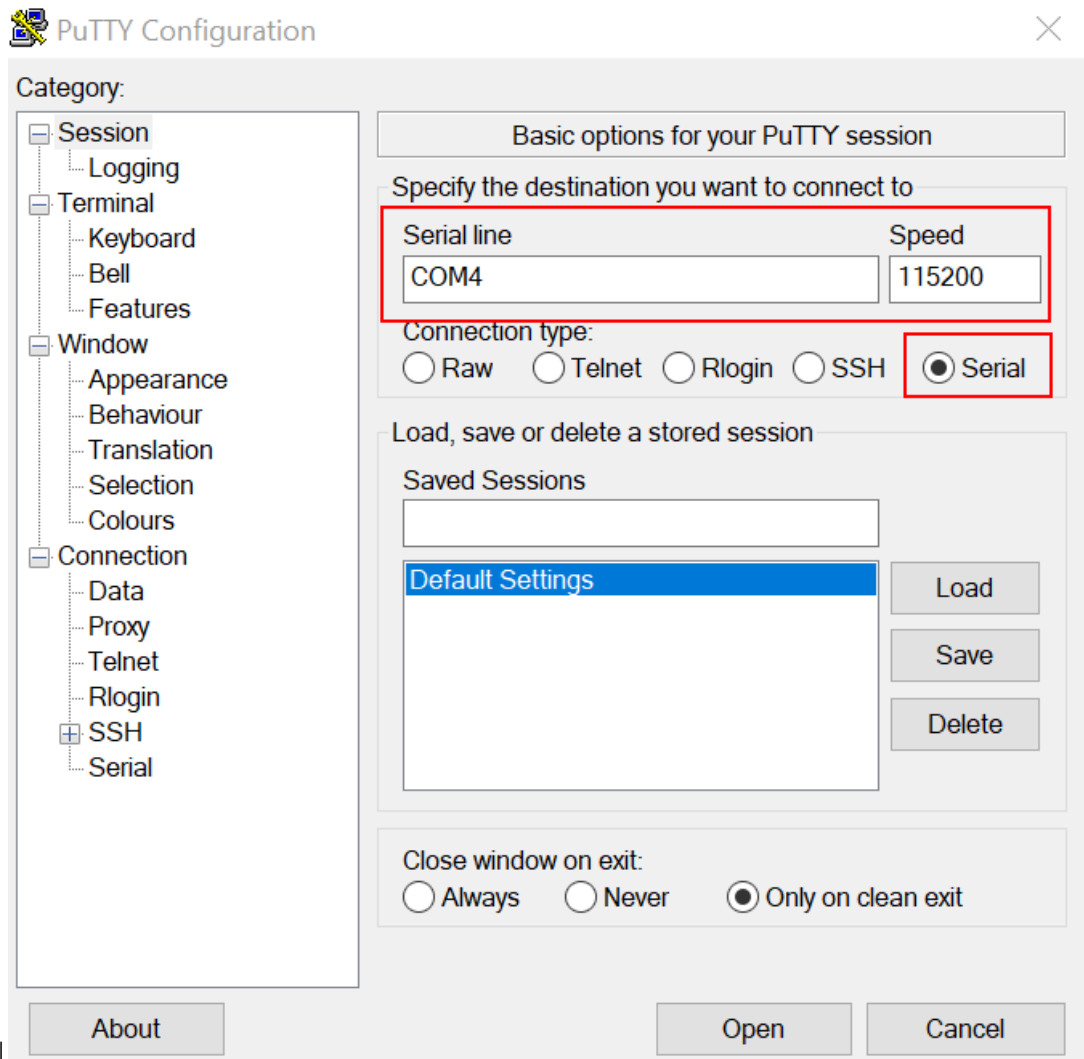
```

Parent topic: [Run a demo using Arm GCC](#)

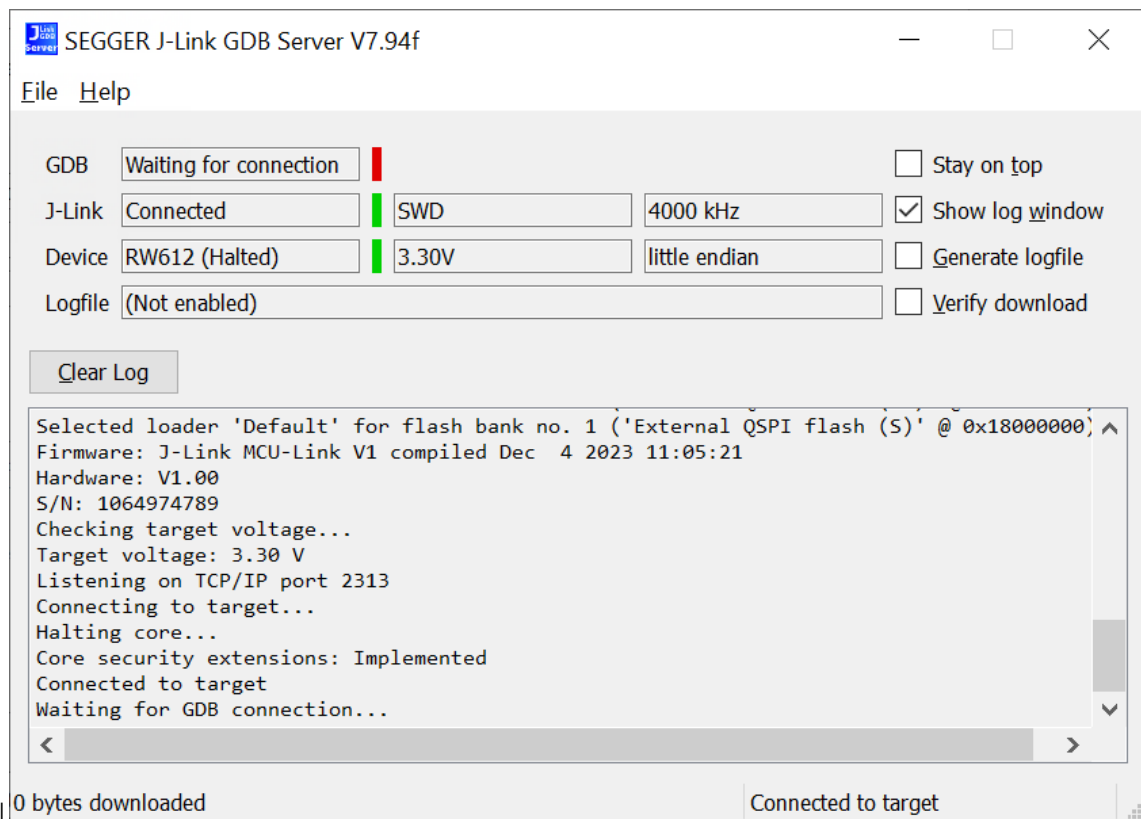
Run an example application This section describes steps to run a demo application using J-Link GDB Server application. To update the on-board LPC-Link2 debugger to Jlink firmware, see [Updating debugger firmware](#).

After the J-Link interface is configured and connected, follow these steps to download and run the demo applications:

1. Connect the development platform to your PC via USB cable between the LPC-Link2 USB connector and the PC USB connector. If using a standalone J-Link debug pod, connect it to the SWD/JTAG connector of the board.
2. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in board.h file)
 2. No parity
 3. 8 data bits
 4. 1 stop bit



3. Open the J-Link GDB Server application.
4. Select “RW612” as the target device and “SWD” as the target interface.
5. After it is connected, the screen appears as in Figure 2.



6. If not already running, open a GCC Arm Embedded tool chain command window. To launch the window, from the Windows operating system Start menu, go to **Programs > GNU Tools Arm Embedded <version>** and select **GCC Command Prompt**.



7. Change to the directory that contains the example application output. The output can be found in using one of these paths, depending on the build target selected:

```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc/debug
```

```
<install_dir>/boards/<board_name>/<example_type>/<application_name>/armgcc/release
```

For this example, the path is:

```
<install_dir>/boards/frdmrw612/demo_apps/hello_world/armgcc/debug
```

8. Run the `arm-none-eabi-gdb.exe <application_name>.elf` command. For this example, it is `arm-none-eabi-gdb.exe hello_world.elf`.

```

C:\SDK_2.16.000_RW612\boards\frdmrw612\demo_apps\hello_world\armgcc\debug>arm-none-eabi-gdb.exe hello_world.elf
GNU gdb (Arm GNU Toolchain 12.3.Rel1 (Build arm-12.35)) 13.2.90.20230627-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=i686-w64-mingw32 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.linaro.org/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello_world.elf...
(gdb)

```

9. Run these commands:

1. target remote localhost:2313
2. monitor reset
3. load
4. monitor reset

10. The application is now downloaded and halted at the watch point. Execute the monitor go command to start the demo application.

The hello_world application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.

```

COM4 - PuTTY
hello world.

```

Parent topic: [Run a demo using Arm GCC](#)

Run a demo using Keil MDK/μVision

This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

Install CMSIS device pack After the MDK tools are installed, Cortex Microcontroller Software Interface Standard (CMSIS) device packs must be installed to fully support the device from a debug perspective. These packs include things such as memory map information, register definitions, and flash programming algorithms. Follow these steps to install the RW612 CMSIS pack.

1. Download the RW612 pack.
2. After downloading the DFP, double click to install it.

Parent topic: [Run a demo using Keil MDK/μVision](#)

Build an example application

1. Open the desired example application workspace in:

```
<install_dir>/boards/<board_name>/*<example\_type>*/<application_name>/mdk
```

The workspace file is named as <demo_name>.uvmpw. For this specific example, the actual path is:

```
<install_dir>/boards/frdmrw612/demo_apps/hello_world/mdk/hello_world.uvmpw
```

2. To build the demo project, select **Rebuild**, highlighted in red.

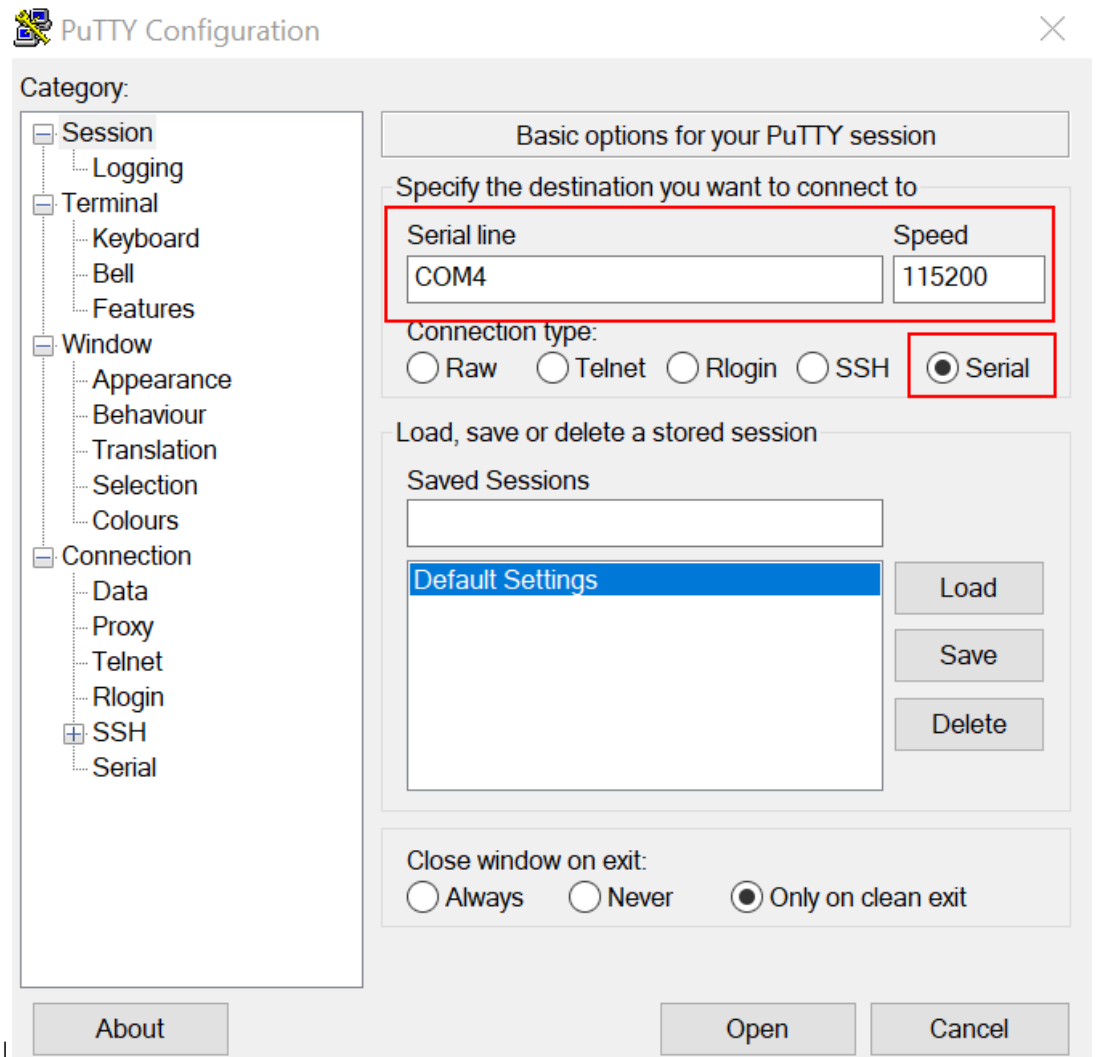


3. The build completes without errors.

Parent topic: [Run a demo using Keil MDK/μVision](#)

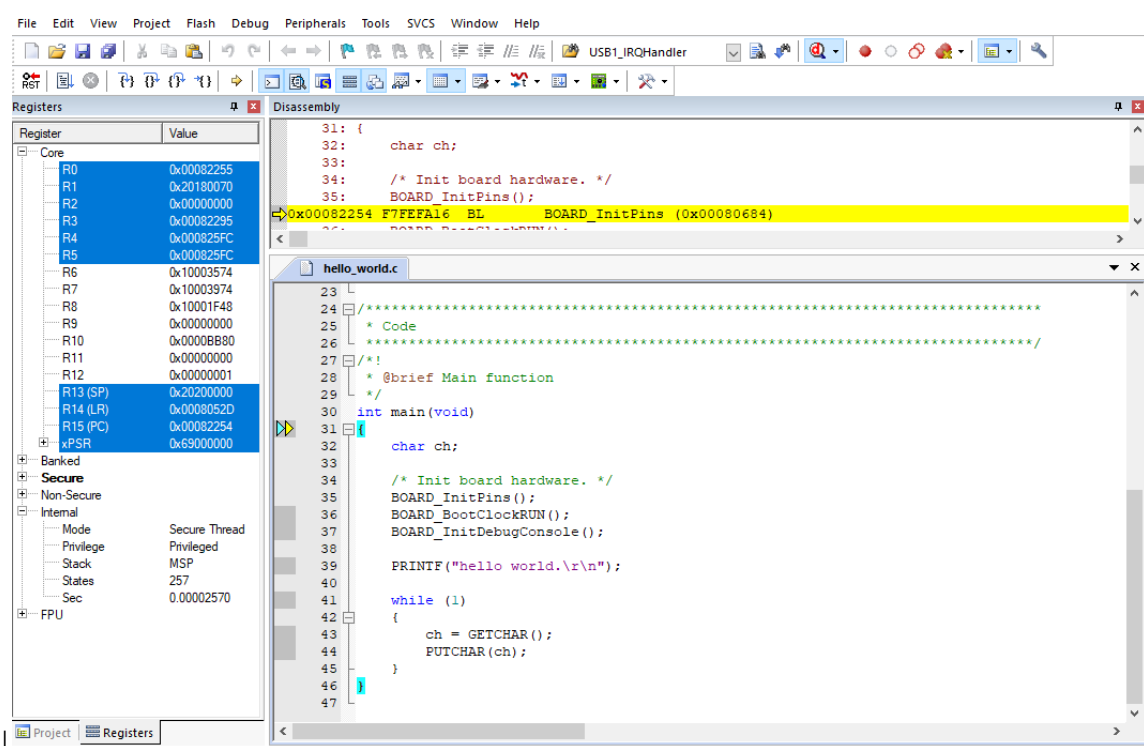
Run an example application To download and run the application, perform these steps:

1. Reference the table in [Default debug interfaces](#) to determine the debug interface that comes loaded on your specific hardware platform.
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits

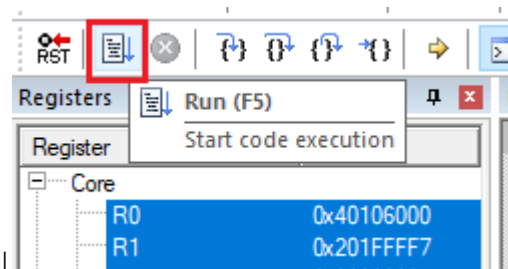


4. 1 stop bit |

4. To debug the application, click **load** (or press the F8 key). Then, click the **Start/Stop Debug Session** button.



5. Run the code by clicking **Run** to start the application.



The `hello\_world` application is now running and a banner is displayed on the terminal. If this is not true, [check your terminal settings and connections](#).

Parent topic: [Run a demo using Keil MDK/μVision](#)

MCUXpresso IDE New Project Wizard

MCUXpresso IDE features a new project wizard. The wizard provides functionality for the user to create new projects from the installed SDKs (and from pre-installed part support). It offers user the flexibility to select and change multiple builds. The wizard also includes a library and provides source code options. The source code is organized as software components, categorized as drivers, utilities, and middleware.

To use the wizard, start the MCUXpresso IDE. This is located in the **QuickStart Panel** at the bottom left of the MCUXpresso IDE window. Select **New project**, as shown in *Figure 1*.



For more details and usage of new project wizard, see the *MCUXpresso_IDE_User_Guide.pdf* in the MCUXpresso IDE installation folder.

How to determine COM port

This section describes the steps necessary to determine the debug COM port number of your NXP hardware development platform.

1. **Linux:** The serial port can be determined by running the following command after the USB Serial is connected to the host:

```
$ dmesg | grep "ttyUSB"
[503175.307873] usb 3-12: cp210x converter now attached to ttyUSB0
[503175.309372] usb 3-12: cp210x converter now attached to ttyUSB1
```

There are two ports, one is Cortex-A core debug console and the other is for Cortex M4.

2. **Windows:** To determine the COM port open Device Manager in the Windows operating system. Click on the **Start** menu and type **Device Manager** in the search bar.
3. In the Device Manager, expand the **Ports (COM & LPT)** section to view the available ports. The COM port names will be different for all the NXP boards.

How to define IRQ handler in CPP files

With MCUXpresso SDK, users could define their own IRQ handler in application level to override the default IRQ handler. For example, to override the default PIT_IRQHandler define in startup_DEVICE.s, application code like app.c can be implement like:

```
c
void PIT_IRQHandler(void)
{
```

(continues on next page)

(continued from previous page)

```
// Your code
}
```

When application file is CPP file, like app.cpp, then `extern "C"` should be used to ensure the function prototype alignment.

```
cpp
extern "C" {
    void PIT_IRQHandler(void);
}
void PIT_IRQHandler(void)
{
    // Your code
}
```

Default debug interfaces

The MCUXpresso SDK supports various hardware platforms that come loaded with a variety of factory programmed debug interface configurations. *Table 1* lists the hardware platforms supported by the MCUXpresso SDK, their default debug interface, and any version information that helps differentiate a specific interface configuration.

Note: The *OpenSDA details* column in *Table 1* is not applicable to LPC.

Hardware platform	Default interface	OpenSDA details
EVK-MC56F83000	P&E Micro OSJTAG	N/A
EVK-MIMXRT595	CMSIS-DAP	N/A
EVK-MIMXRT685	CMSIS-DAP	N/A
FRDM-K22F	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.1
FRDM-K28F	DAPLink	OpenSDA v2.1
FRDM-K32L2A4S	CMSIS-DAP	OpenSDA v2.1
FRDM-K32L2B	CMSIS-DAP	OpenSDA v2.1
FRDM-K32W042	CMSIS-DAP	N/A
FRDM-K64F	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.0
FRDM-K66F	J-Link OpenSDA	OpenSDA v2.1
FRDM-K82F	CMSIS-DAP	OpenSDA v2.1
FRDM-KE15Z	DAPLink	OpenSDA v2.1
FRDM-KE16Z	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.2
FRDM-KL02Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL03Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL25Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL26Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL27Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL28Z	P&E Micro OpenSDA	OpenSDA v2.1
FRDM-KL43Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL46Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KL81Z	CMSIS-DAP	OpenSDA v2.0
FRDM-KL82Z	CMSIS-DAP	OpenSDA v2.0
FRDM-KV10Z	CMSIS-DAP	OpenSDA v2.1
FRDM-KV11Z	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KV31F	P&E Micro OpenSDA	OpenSDA v1.0
FRDM-KW24	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.1
FRDM-KW36	DAPLink	OpenSDA v2.2
FRDM-KW41Z	CMSIS-DAP/DAPLink	OpenSDA v2.1 or greater
Hexiwear	CMSIS-DAP/mbd/DAPLink	OpenSDA v2.0

continues on next page

Table 1 – continued from previous page

Hardware platform	Default interface	OpenSDA details
HVP-KE18F	DAPLink	OpenSDA v2.2
HVP-KV46F150M	P&E Micro OpenSDA	OpenSDA v1
HVP-KV11Z75M	CMSIS-DAP	OpenSDA v2.1
HVP-KV58F	CMSIS-DAP	OpenSDA v2.1
HVP-KV31F120M	P&E Micro OpenSDA	OpenSDA v1
JN5189DK6	CMSIS-DAP	N/A
LPC54018 IoT Module	N/A	N/A
LPCXpresso54018	CMSIS-DAP	N/A
LPCXpresso54102	CMSIS-DAP	N/A
LPCXpresso54114	CMSIS-DAP	N/A
LPCXpresso51U68	CMSIS-DAP	N/A
LPCXpresso54608	CMSIS-DAP	N/A
LPCXpresso54618	CMSIS-DAP	N/A
LPCXpresso54628	CMSIS-DAP	N/A
LPCXpresso54S018M	CMSIS-DAP	N/A
LPCXpresso55s16	CMSIS-DAP	N/A
LPCXpresso55s28	CMSIS-DAP	N/A
LPCXpresso55s69	CMSIS-DAP	N/A
MAPS-KS22	J-Link OpenSDA	OpenSDA v2.0
MIMXRT1170-EVK	CMSIS-DAP	N/A
RD-RW612-BGA	CMSIS-DAP	N/A
RD-RW612-QFN	CMSIS-DAP	N/A
TWR-K21D50M	P&E Micro OSJTAG	N/A
TWR-K21F120M	P&E Micro OSJTAG	N/A
TWR-K22F120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K24F120M	CMSIS-DAP/mbed	OpenSDA v2.1
TWR-K60D100M	P&E Micro OSJTAG	N/A
TWR-K64D120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K64F120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K65D180M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K65D180M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV10Z32	P&E Micro OpenSDA	OpenSDA v1.0
TWR-K80F150M	CMSIS-DAP	OpenSDA v2.1
TWR-K81F150M	CMSIS-DAP	OpenSDA v2.1
TWR-KE18F	DAPLink	OpenSDA v2.1
TWR-KL28Z72M	P&E Micro OpenSDA	OpenSDA v2.1
TWR-KL43Z48M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KL81Z72M	CMSIS-DAP	OpenSDA v2.0
TWR-KL82Z72M	CMSIS-DAP	OpenSDA v2.0
TWR-KM34Z75M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KM35Z75M	DAPLink	OpenSDA v2.2
TWR-KV10Z32	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV11Z75M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV31F120M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV46F150M	P&E Micro OpenSDA	OpenSDA v1.0
TWR-KV58F220M	CMSIS-DAP	OpenSDA v2.1
TWR-KW24D512	P&E Micro OpenSDA	OpenSDA v1.0
USB-KW24D512	N/A External probe	N/A
USB-KW41Z	CMSIS-DAP\DAPILink	OpenSDA v2.1 or greater

Updating debugger firmware

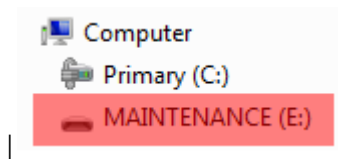
Updating OpenSDA firmware Any NXP hardware platform that comes with an OpenSDA-compatible debug interface has the ability to update the OpenSDA firmware. For reference,

OpenSDA firmware files can be found at the links below:

- J-Link: Download appropriate image from www.segger.com/opensda.html. Choose the appropriate J-Link binary based on the table in *Table 1*. Any OpenSDA v1.0 interface should use the standard OpenSDA download (in other words, the one with no version). For OpenSDA 2.0 or 2.1, select the corresponding binary.
- P&E Micro: Downloading P&E Micro OpenSDA firmware images requires registration with P&E Micro (www.pemicro.com).

Perform the following steps to update the OpenSDA firmware on your board for Windows and Linux OS users:

1. Unplug the board's USB cable.
2. Press the **Reset** button on the board. While still holding the button, plug the USB cable back into the board.
3. When the board re-enumerates, it shows up as a disk drive called **MAINTENANCE**.



4. Drag and drop the new firmware image onto the MAINTENANCE drive.

Note: If for any reason the firmware update fails, the board can always re-enter maintenance mode by holding down **Reset** button and power cycling.

These steps show how to update the OpenSDA firmware on your board for Mac OS users.

1. Unplug the board's USB cable.
2. Press the **Reset** button of the board. While still holding the button, plug the USB cable back into the board.
3. For boards with OpenSDA v2.0 or v2.1, it shows up as a disk drive called **BOOTLOADER** in **Finder**. Boards with OpenSDA v1.0 may or may not show up depending on the bootloader version. If you see the drive in **Finder**, proceed to the next step. If you do not see the drive in **Finder**, use a PC with Windows OS 7 or an earlier version to either update the OpenSDA firmware, or update the OpenSDA bootloader to version 1.11 or later. The bootloader update instructions and image can be obtained from P&E Microcomputer website.
4. For OpenSDA v2.1 and OpenSDA v1.0 (with bootloader 1.11 or later) users, drag the new firmware image onto the BOOTLOADER drive in **Finder**.
5. For OpenSDA v2.0 users, type these commands in a Terminal window:

```
> sudo mount -u -w -o sync /Volumes/BOOTLOADER
> cp -X <path to update file> /Volumes/BOOTLOADER
```

Note: If for any reason the firmware update fails, the board can always re-enter bootloader mode by holding down the **Reset** button and power cycling.

Parent topic: [Updating debugger firmware](#)

Updating MCU-Link firmware The LPCXpresso and RW61X hardware platform comes with a CMSIS-DAP-compatible debug interface (known as LPC-Link2). This firmware in this debug interface may be updated using the host computer utility called MCU-Link. This typically used when switching between the default debugger protocol (CMSIS-DAP) to SEGGER J-Link, or for updating this firmware with new releases of these. This section contains the steps to re-program the debug probe firmware.

Note: If MCUXpresso IDE is used and the jumper making DFUlink is installed on the board (JP5 on some boards, but consult the board user manual or schematic for specific jumper number), LPC-Link2 debug probe boots to DFU mode, and MCUXpresso IDE automatically downloads the CMSIS-DAP firmware to the probe before flash memory programming (after clicking **Debug**). Using DFU mode ensures most up-to-date/compatible firmware is used with MCUXpresso IDE.

NXP provides the MCU-Link utility, which is the recommended tool for programming the latest versions of CMSIS-DAP and J-Link firmware onto FRDM-RW612 or LPCXpresso boards. The utility can be downloaded from <https://www.nxp.com/design/microcontrollers-developer-resources/mcu-link-debug-probe:MCU-LINK>.

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in the MCU-Link user guide (<https://www.nxp.com/design/microcontrollers-developer-resources/mcu-link-debug-probe:MCU-LINK>, select the documentation tab).

1. Install the MCU-Link utility.
2. Unplug the board's USB cable.
3. Make the DFU link (install the jumper labelled DFUlink, JP10 on FRDM-RW612-BGA board).
4. Connect the probe to the host via USB (use Link USB connector).
5. Open a command shell and call the appropriate script located in the MCU-Link installation directory (<MCU-Link install dir>).
 1. To program CMSIS-DAP debug firmware: <LPCScript install dir>/scripts/program_CMSIS
 2. To program J-Link debug firmware: <LPCScript install dir>/scripts/program_JLINK
6. Remove DFU link (remove the jumper installed in Step 3).
7. Re-power the board by removing the USB cable and plugging it in again.

Parent topic: [Updating debugger firmware](#)

1.3 Getting Started with MCUXpresso SDK GitHub

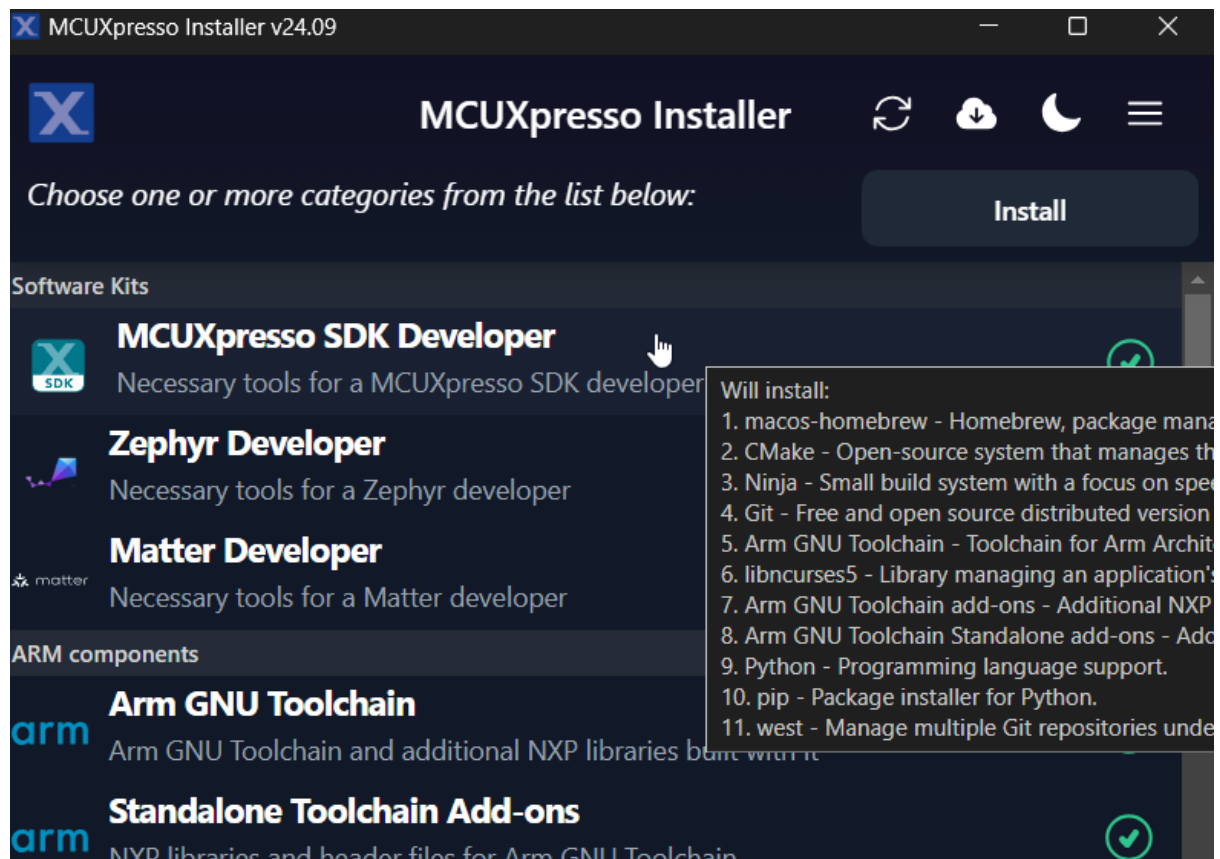
1.3.1 Getting Started with MCUXpresso SDK Repository

Installation

NOTE

If the installation instruction asks/selects whether to have the tool installation path added to the PATH variable, agree/select the choice. This option ensures that the tool can be used in any terminal in any path. [Verify the installation](#) after each tool installation.

Install Prerequisites with MCUXpresso Installer The MCUXpresso Installer offers a quick and easy way to install the basic tools needed. The MCUXpresso Installer can be obtained from <https://github.com/nxp-mcuxpresso/vscode-for-mcux/wiki/Dependency-Installation>. The MCUXpresso Installer is an automated installation process, simply select MCUXpresso SDK Developer from the menu and click install. If you prefer to install the basic tools manually, refer to the next section.



Alternative: Manual Installation

Basic tools

Git Git is a free and open source distributed version control system. Git is designed to handle everything from small to large projects with speed and efficiency. To install Git, visit the official [Git website](#). Download the appropriate version (you may use the latest one) for your operating system (Windows, macOS, Linux). Then run the installer and follow the installation instructions.

User `git --version` to check the version if you have a version installed.

Then configure your username and email using the commands:

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

Python Install python 3.10 or latest. Follow the [Python Download](#) guide.

Use `python --version` to check the version if you have a version installed.

West Please use the west version equal or greater than 1.2.0

```
# Note: you can add option '--default-timeout=1000' if you meet connection issue. Or you may set a different
↪source using option '-i'.
# for example, in China you could try: pip install -U west -i https://pypi.tuna.tsinghua.edu.cn/simple
pip install -U west
```

Build And Configuration System

CMake It is strongly recommended to use CMake version equal or later than 3.30.0. You can get latest CMake distributions from [the official CMake download page](#).

For Windows, you can directly use the .msi installer like [cmake-3.31.4-windows-x86_64.msi](#) to install.

For Linux, CMake can be installed using the system package manager or by getting binaries from [the official CMake download page](#).

After installation, you can use `cmake --version` to check the version.

Ninja Please use the ninja version equal or later than 1.12.1.

By default, Windows comes with the Ninja program. If the default Ninja version is too old, you can directly download the [ninja binary](#) and register the ninja executor location path into your system path variable to work.

For Linux, you can use your [system package manager](#) or you can directly download the [ninja binary](#) to work.

After installation, you can use `ninja --version` to check the version.

Kconfig MCUXpresso SDK uses Kconfig python implementation. We customize it based on our needs and integrate it into our build and configuration system. The Kconfiglib sources are placed under `mcuxsdk/scripts/kconfig` folder.

Please make sure [python](#) environment is setup ready then you can use the Kconfig.

Ruby Our build system supports IDE project generation for iar, mdk, codewarrior and xtensa to provide OOB from build to debug. This feature is implemented with ruby. You can follow the guide ruby environment setup to setup the ruby environment. Since we provide a built-in portable ruby, it is just a simple one cmd installation.

If you only work with CLI, you can skip this step.

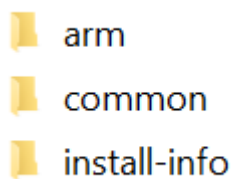
Toolchain MCUXpresso SDK supports all mainstream toolchains for embedded development. You can install your used or interested toolchains following the guides.

Toolchain	Download and Installation Guide	Note
Armgcc	Arm GNU Toolchain Install Guide	ARMGCC is default toolchain
IAR	IAR Installation and Licensing quick reference guide	
MDK	MDK Installation	
Armclang	Installing Arm Compiler for Embedded	
Zephyr	Zephyr SDK	
Codewarrior	NXP CodeWarrior	
Xtensa	Tensilica Tools	
NXP S32Compiler RISC-V Zen-V	NXP Website	

After you have installed the toolchains, register them in the system environment variables. This will allow the west build to recognize them:

Toolchain	Environment Variable	Example	Cmd Line Argument
Armgcc	AR-MGCC_DIR	C:\armgcc for windows/usr for Linux. Typically arm-none-eabi-* is installed under /usr/bin	– toolchain armgcc
IAR	IAR_DIR	C:\iar\ewarm-9.60.3 for Windows/opt/iarsystems/bxarm-9.60.3 for Linux	– toolchain iar
MDK	MDK_DIR	C:\Keil_v5 for Windows.MDK IDE is not officially supported with Linux.	– toolchain mdk
Armclang	ARM-CLANG_DIR	C:\ArmCompilerforEmbedded6.22 for Windows/opt/ArmCompilerforEmbedded6.21 for Linux	– toolchain mdk
Zephyr	ZEPHYR_DIR	c:\NXP\zephyr-sdk-<version> for windows/opt/zephyr-sdk-<version> for Linux	– toolchain zephyr
CodeWarrior	CW_DIR	C:\Freescalar\CW MCU v11.2 for windowsCodeWarrior is not supported with Linux	– toolchain code-warrior
Xtensa	XCC_DIR	C:\xtensa\XtDevTools\install\tools\RI-2023.11-win32\XtensaTools for windows/opt/xtensa/XtDevTools/install/tools/RI-2023.11-Linux/XtensaTools for Linux	– toolchain xtensa
NXP S32Compiler RISC-V Zen-V	RISCV-LVM_DIR	C:\riscv-llvm-win32_b298_b298_2024.08.12 for Windows/opt/riscv-llvm-Linux-x64_b298_b298_2024.08.12 for Linux	– toolchain riscv-llvm

- The <toolchain>_DIR is the root installation folder, not the binary location folder. For IAR, it is directory containing following installation folders:



- MDK IDE using armclang toolchain only officially supports Windows. In Linux, please directly use armclang toolchain by setting ARMCLANG_DIR. In Windows, since most Keil users will install MDK IDE instead of standalone armclang toolchain, the MDK_DIR has higher priority than ARMCLANG_DIR.
- For Xtensa toolchain, please set the XTENSA_CORE environment variable. Here's an example list:

Device Core	XTENSA_CORE
RT500 fusion1	nxp_rt500_RI23_11_newlib
RT600 hifi4	nxp_rt600_RI23_11_newlib
RT700 hifi1	rt700_hifi1_RI23_11_nlib
RT700 hifi4	t700_hifi4_RI23_11_nlib
i.MX8ULP fusion1	fusion_nxp02_dsp_prod

- In Windows, the short path is used in environment variables. If any toolchain is using the long path, you can open a command window from the toolchain folder and use below command to get the short path: `for %i in (.) do echo %~fsi`

Tool installation check Once installed, open a terminal or command prompt and type the associated command to verify the installation.

If you see the version number, you have successfully installed the tool. Else, check whether the tool's installation path is added into the PATH variable. You can add the installation path to the PATH with the commands below:

- Windows: Open command prompt or powershell, run below command to show the user PATH variable.

```
reg query HKEY_CURRENT_USER\Environment /v PATH
```

The tool installation path should be `C:\Users\xxx\AppData\Local\Programs\Git\cmd`. If the path is not seen in the output from above, append the path value to the PATH variable with the command below:

```
reg add HKEY_CURRENT_USER\Environment /v PATH /d "%PATH%;C:\Users\xxx\AppData\
↪Local\Programs\Git\cmd"
```

Then close the command prompt or powershell and verify the tool command again.

- Linux:
 1. Open the `$HOME/.bashrc` file using a text editor, such as `vim`.
 2. Go to the end of the file.
 3. Add the line which appends the tool installation path to the PATH variable and export PATH at the end of the file. For example, `export PATH="/Directory1:$PATH"`.
 4. Save and exit.
 5. Execute the script with `source .bashrc` or reboot the system to make the changes live. To verify the changes, run `echo $PATH`.
- macOS:
 1. Open the `$HOME/.bash_profile` file using a text editor, such as `nano`.
 2. Go to the end of the file.
 3. Add the line which appends the tool installation path to the PATH variable and export PATH at the end of the file. For example, `export PATH="/Directory1:$PATH"`.
 4. Save and exit.
 5. Execute the script with `source .bash_profile` or reboot the system to make the changes live. To verify the changes, run `echo $PATH`.

Get MCUXpresso SDK Repo

Establish SDK Workspace To get the MCUXpresso SDK repository, use the `west` tool to clone the manifest repository and checkout all the west projects.

```
# Initialize west with the manifest repository
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests/ mcuxpresso-sdk

# Update the west projects
cd mcuxpresso-sdk
west update

# Allow the usage of west extensions provided by MCUXpresso SDK
west config commands.allow_extensions true
```

Install Python Dependency(If do tool installation manually) To create a Python virtual environment in the west workspace core repo directory `mcuxsdk`, follow these steps:

1. Navigate to the core directory:

```
cd mcuxsdk
```

2. [Optional] Create and activate the virtual environment: If you don't want to use the python virtual environment, skip this step. **We strongly suggest you use `venv` to avoid conflicts with other projects using python.**

```
python -m venv .venv

# For Linux/MacOS
source .venv/bin/activate

# For Windows
.\.venv\Scripts\activate
# If you are using powershell and see the issue that the activate script cannot be run.
# You may fix the issue by opening the powershell as administrator and run below command:
powershell Set-ExecutionPolicy RemoteSigned
# then run above activate command again.
```

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

Remember to activate the virtual environment every time you start working in this directory. If you are using some modern shell like `zsh`, there are some powerful plugins to help you auto switch `venv` among workspaces. For example, `zsh-autoswitch-virtualenv`.

3. Install the required Python packages:

```
# Note: you can add option '--default-timeout=1000' if you meet connection issue. Or you may set a
↪ different source using option '-i'.
# for example, in China you could try: pip3 install -r mcuxsdk/scripts/requirements.txt -i https://pypi.
↪ tuna.tsinghua.edu.cn/simple
pip install -r scripts/requirements.txt
```

Explore Contents

This section helps you build basic understanding of current fundamental project content and guides you how to build and run the provided example project in whole SDK delivery.

Folder View The whole MCUXpresso SDK project, after you have done the `west init` and `west update` operations follow the guideline at [Getting Started Guide](#), have below folder structure:

Folder	Description
manifests	Manifest repo, contains the manifest file to initialize and update the west workspace.
mcuxsdk	The MCUXpresso SDK source code, examples, middleware integration and script files.

All the projects record in the [Manifest repo](#) are checked out to the folder `mcuxsdk/`, the layout of `mcuxsdk` folder is shown as below:

Folder	Description
arch	Arch related files such as ARM CMSIS core files, RISC-V files and the build files related to the architecture.
cmake	The cmake modules, files which organize the build system.
components	Software components.
devices	Device support package which categorized by device series. For each device, header file, feature file, startup file and linker files are provided, also device specific drivers are included.
docs	Documentation source and build configuration for this sphinx built online documentation.
drivers	Peripheral drivers.
examples	Various demos and examples, support files on different supported boards. For each board support, there are board configuration files.
middleware	Middleware components integrated into SDK.
rtos	Rtos components integrated into SDK.
scripts	Script files for the west extension command and build system support.
svd	Svd files for devices, this is optional because of large size. Customers run <code>west manifest config group.filter +optional</code> and <code>west update mcux-soc-svd</code> to get this folder.

Examples Project The examples project is part of the whole SDK delivery, and locates in the folder `mcuxsdk/examples` of west workspace.

Examples files are placed in folder of `<example_category>`, these examples include (but are not limited to)

- `demo_apps`: Basic demo set to start using SDK, including `hello_world` and `led_blinky`.
- `driver_examples`: Simple applications that show how to use the peripheral drivers for a single use case. These applications typically only use a single peripheral but there are cases where multiple peripherals are used (for example, SPI transfer using DMA).

Board porting layers are placed in folder of `_boards/<board_name>` which aims at providing the board specific parts for examples code mentioned above.

Run a demo using MCUXpresso for VS Code

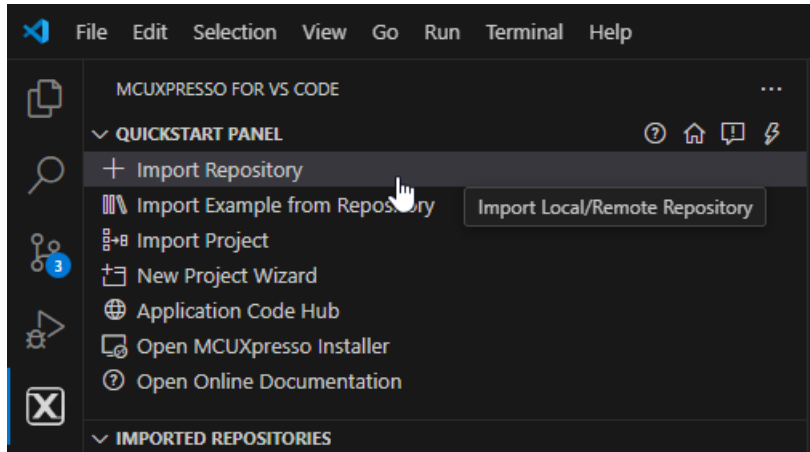
This section explains how to configure MCUXpresso for VS Code to build, run, and debug example applications. This guide uses the `hello_world` demo application as an example. However, these

steps can be applied to any example application in the MCUXpresso SDK.

Build an example application This section assumes that the user has already obtained the SDK as outlined in [Get MCUXpresso SDK Repo](#).

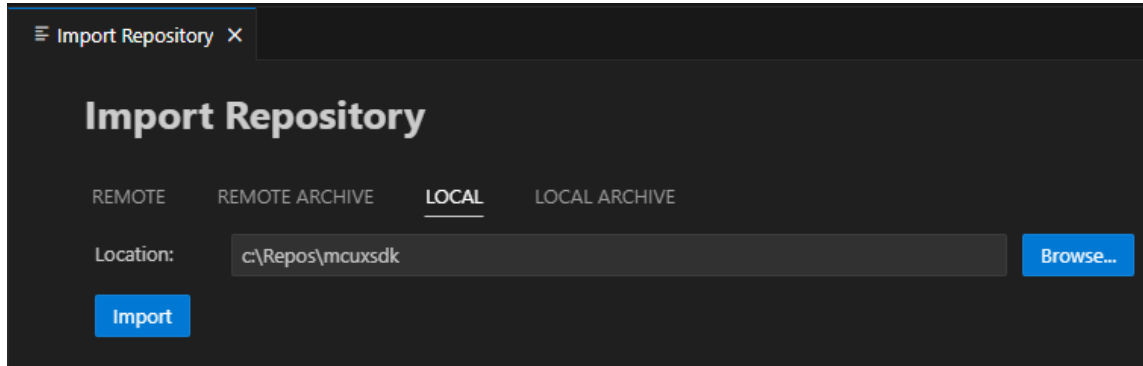
To build an example application:

1. Import the SDK into your workspace. Click **Import Repository** from the **QUICKSTART PANEL**.

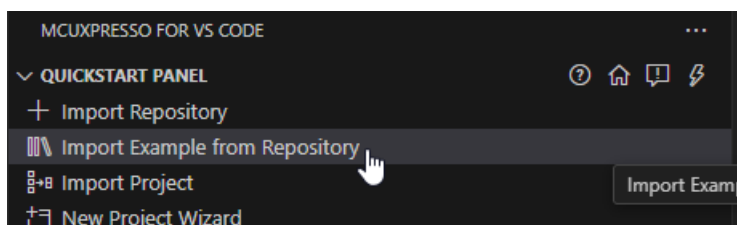


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

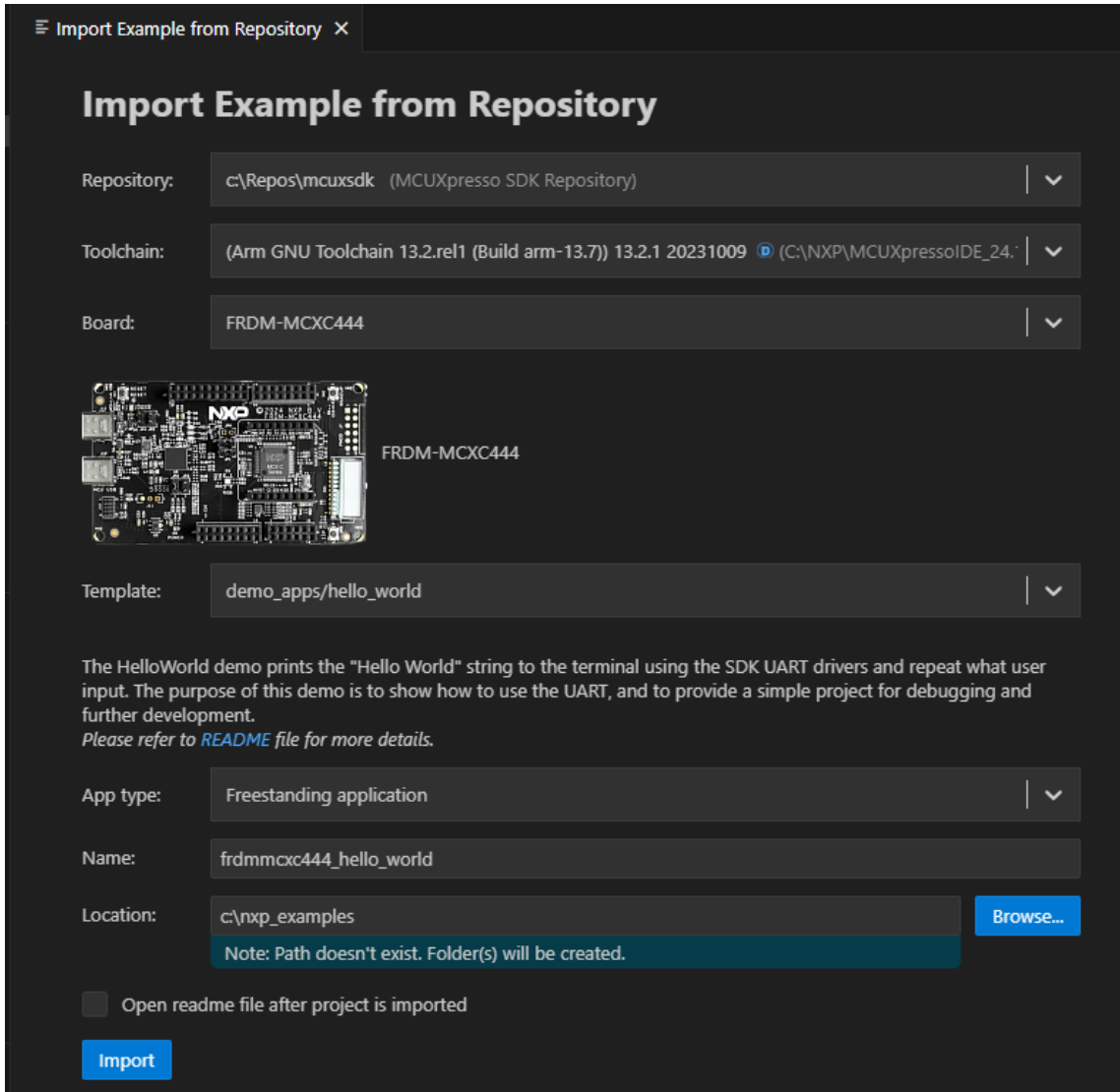
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



2. Click **Import Example from Repository** from the **QUICKSTART PANEL**.



In the dropdown menu, select the MCUXpresso SDK, the Arm GNU Toolchain, your board, template, and application type. Click **Import**.



Import Example from Repository

Repository: c:\Repos\mcuxsdk (MCUXpresso SDK Repository) | v

Toolchain: (Arm GNU Toolchain 13.2.rel1 (Build arm-13.7)) 13.2.1 20231009 | v

Board: FRDM-MCXC444 | v

 FRDM-MCXC444

Template: demo_apps/hello_world | v

The HelloWorld demo prints the "Hello World" string to the terminal using the SDK UART drivers and repeat what user input. The purpose of this demo is to show how to use the UART, and to provide a simple project for debugging and further development.
Please refer to [README](#) file for more details.

App type: Freestanding application | v

Name: frdmmcxc444_hello_world

Location: c:\nxp_examples [Browse...](#)

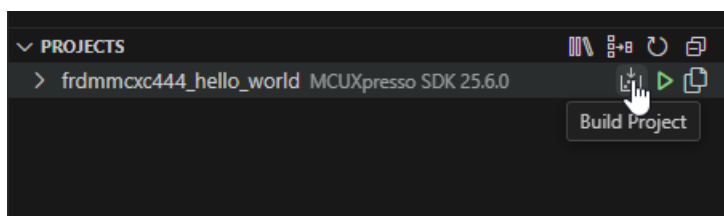
Note: Path doesn't exist. Folder(s) will be created.

☐ Open readme file after project is imported

[Import](#)

Note: The MCUXpresso SDK projects can be imported as **Repository applications** or **Freestanding applications**. The difference between the two is the import location. Projects imported as Repository examples will be located inside the MCUXpresso SDK, whereas Freestanding examples can be imported to a user-defined location. Select between these by designating your selection in the **App type** dropdown menu.

- VS Code will prompt you to confirm if the imported files are trusted. Click **Yes**.
- Navigate to the **PROJECTS** view. Find your project and click the **Build Project** icon.



The integrated terminal will open at the bottom and will display the build output.

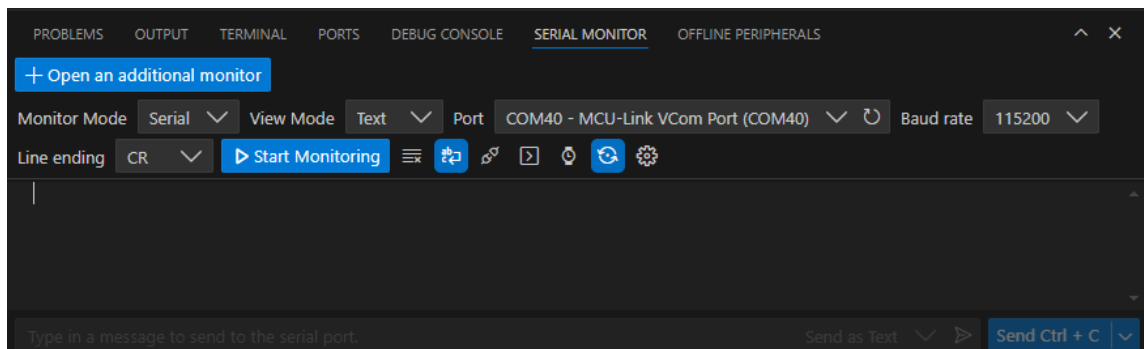
```

[17/21] Building C object CMakeFiles/hello_world.dir/C:/Repos/mcuxsdk/mcuxsdk/components/debug_console_lite/fsl_debug_console.c.obj
[18/21] Building C object CMakeFiles/hello_world.dir/C:/Repos/mcuxsdk/mcuxsdk/devices/MCXC/MCXC444/drivers/fsl_clock.c.obj
[19/21] Building C object CMakeFiles/hello_world.dir/C:/Repos/mcuxsdk/mcuxsdk/drivers/lpuart/fsl_lpuart.c.obj
[20/21] Building C object CMakeFiles/hello_world.dir/C:/Repos/mcuxsdk/mcuxsdk/drivers/uart/fsl_uart.c.obj
[21/21] Linking C executable hello_world.elf
Memory region      Used Size  Region Size  %age Used
  m_interrupts:     192 B    512 B      37.50%
  m_flash_config:    16 B     16 B     100.00%
  m_text:          7892 B   261104 B     3.02%
  m_data:          2128 B    32 KB     6.49%
build finished successfully.
Terminal will be reused by tasks, press any key to close it.

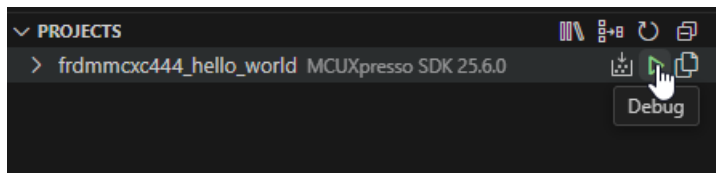
```

Run an example application **Note:** for full details on MCUXpresso for VS Code debug probe support, see [MCUXpresso for VS Code Wiki](#).

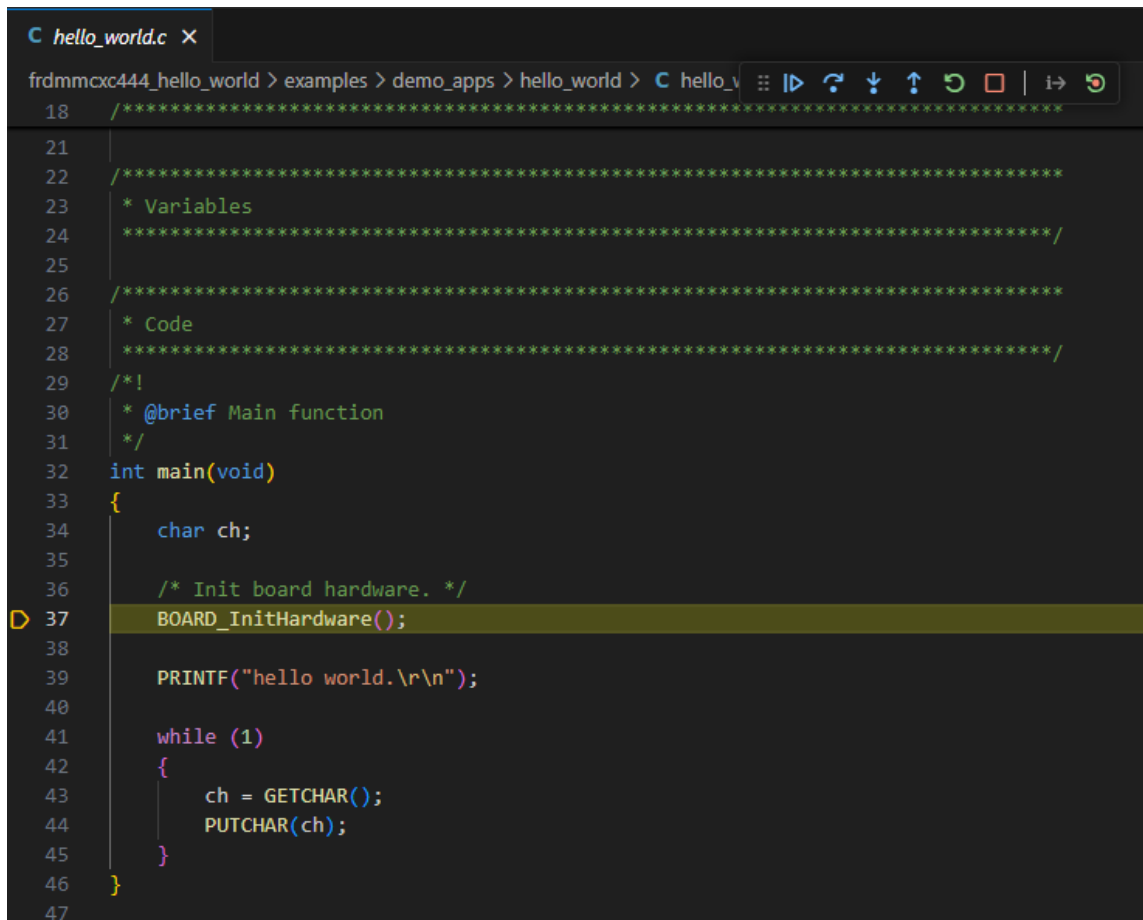
1. Open the **Serial Monitor** from the VS Code's integrated terminal. Select the VCom Port for your device and set the baud rate to 115200.



2. Navigate to the **PROJECTS** view and click the play button to initiate a debug session.



The debug session will begin. The debug controls are initially at the top.

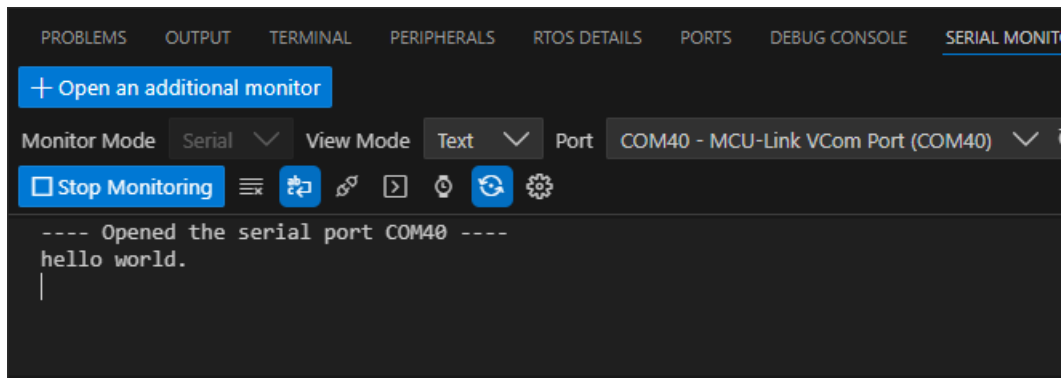


```

18  /*****
21
22  /*****
23  * Variables
24  *****/
25
26  /*****
27  * Code
28  *****/
29  /*!
30  * @brief Main function
31  */
32  int main(void)
33  {
34      char ch;
35
36      /* Init board hardware. */
37      BOARD_InitHardware();
38
39      PRINTF("hello world.\r\n");
40
41      while (1)
42      {
43          ch = GETCHAR();
44          PUTCHAR(ch);
45      }
46  }
47

```

3. Click **Continue** on the debug controls to resume execution of the code. Observe the output on the **Serial Monitor**.



```

PROBLEMS  OUTPUT  TERMINAL  PERIPHERALS  RTOS DETAILS  PORTS  DEBUG CONSOLE  SERIAL MONITOR
+ Open an additional monitor
Monitor Mode: Serial View Mode: Text Port: COM40 - MCU-Link VCom Port (COM40)
[Stop Monitoring] [Icons]
---- Opened the serial port COM40 ----
hello world.
|

```

Running a demo using ARMGCC CLI/IAR/MDK

Supported Boards Use the west extension `west list_project` to understand the board support scope for a specified example. All supported build command will be listed in output:

```
west list_project -p examples/demo_apps/hello_world [-t armgcc]
```

```
INFO: [ 1][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪evk9mimx8ulp -Dcore_id=cm33]
```

```
INFO: [ 2][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪evkbimxrt1050]
```

```
INFO: [ 3][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
```

(continues on next page)

(continued from previous page)

```

↪ evkbnimxrt1060]
INFO: [ 4][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt1170 -Dcore_id=cm4]
INFO: [ 5][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt1170 -Dcore_id=cm7]
INFO: [ 6][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt1060]
INFO: [ 7][west build -p always examples/demo_apps/hello_world --toolchain armgcc --config release -b_
↪ evkbnimxrt7ulp]
...

```

The supported toolchains and build targets for an example are decided by the example-self example.yml and board example.yml, please refer Example Toolchains and Targets for more details.

Build the project Use west build -h to see help information for west build command. Compared to zephyr's west build, MCUXpresso SDK's west build command provides following additional options for mcux examples:

- --toolchain: specify the toolchain for this build, default armgcc.
- --config: value for CMAKE_BUILD_TYPE. If not provided, build system will get all the example supported build targets and use the first debug target as the default one. Please refer Example Toolchains and Targets for more details about example supported build targets.

Here are some typical usages for generating a SDK example:

```

# Generate example with default settings, default used device is the mainset MK22F51212
west build -b frdmk22f examples/demo_apps/hello_world

# Just print cmake commands, do not execute it
west build -b frdmk22f examples/demo_apps/hello_world --dry-run

# Generate example with other toolchain like iar, default armgcc
west build -b frdmk22f examples/demo_apps/hello_world --toolchain iar

# Generate example with other config type
west build -b frdmk22f examples/demo_apps/hello_world --config release

# Generate example with other devices with --device
west build -b frdmk22f examples/demo_apps/hello_world --device MK22F12810 --config release

```

For multicore devices, you shall specify the corresponding core id by passing the command line argument -Dcore_id. For example

```

west build -b evkbnimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config_
↪ flexspi_nor_debug

```

For shield, please use the --shield to specify the shield to run, like

```

west build -b mimxrt700evk --shield a8974 examples/issdk_examples/sensors/fxls8974cf/fxls8974cf_poll -
↪ Dcore_id=cm33_core0

```

Sysbuild(System build) To support multicore project building, we ported Sysbuild from Zephyr. It supports combine multiple projects for compilation. You can build all projects by adding --sysbuild for main application. For example:

```

west build -b evkbnimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪ id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always

```

For more details, please refer to System build.

Config a Project Example in MCUXpresso SDK is configured and tested with pre-defined configuration. You can follow steps blow to change the configuration.

1. Run cmake configuration

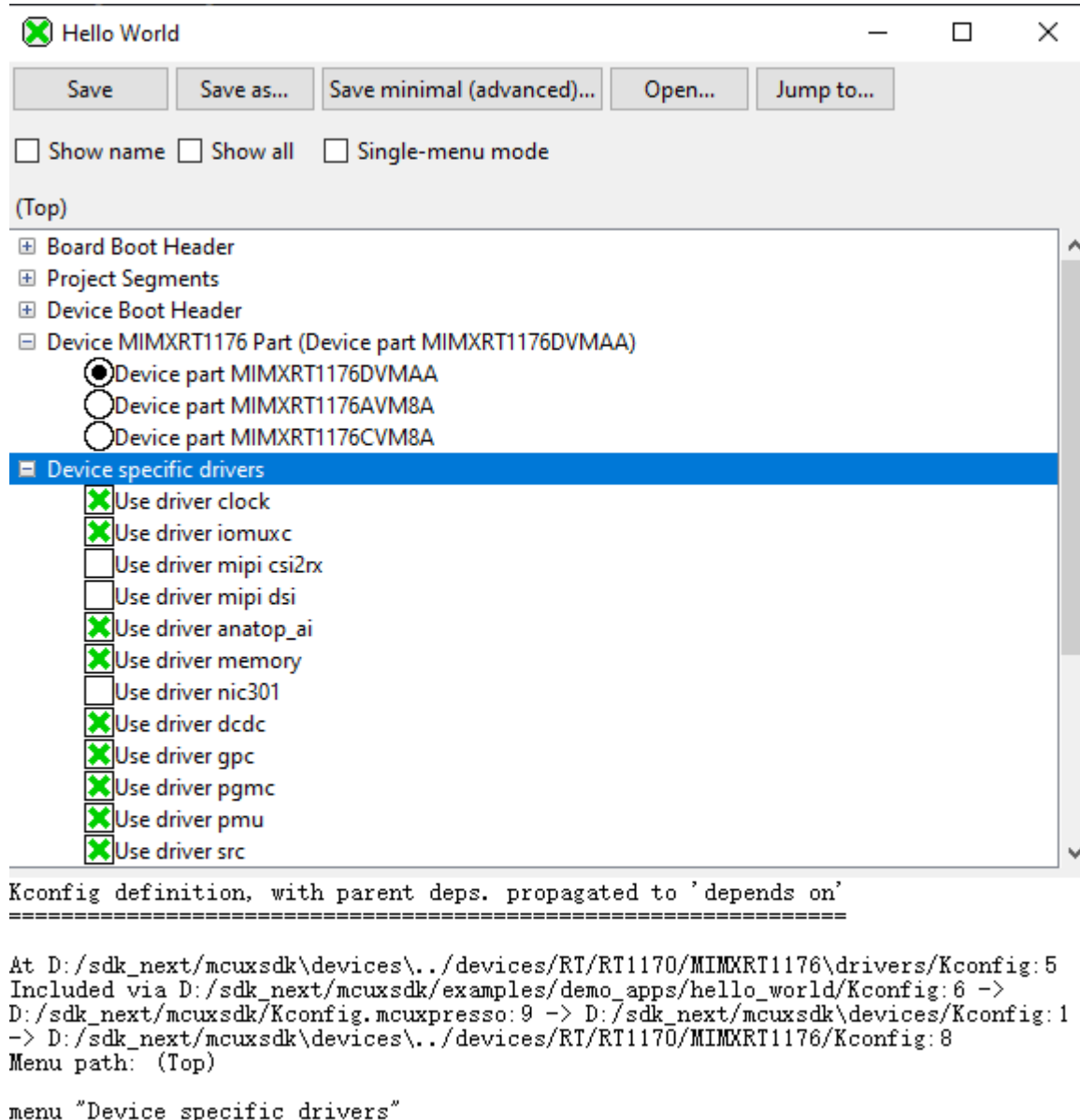
```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

Please note the project will be built without --cmake-only parameter.

2. Run guiconfig target

```
west build -t guiconfig
```

Then you will get the Kconfig GUI launched, like



You can reconfigure the project by selecting/deselecting Kconfig options.

After saving and closing the Kconfig GUI, you can directly run `west build` to build with the new configuration.

Flash *Note:* Please refer Flash and Debug The Example to enable west flash/debug support.
Flash the hello_world example:

```
west flash -r linkserver
```

Debug Start a gdb interface by following command:

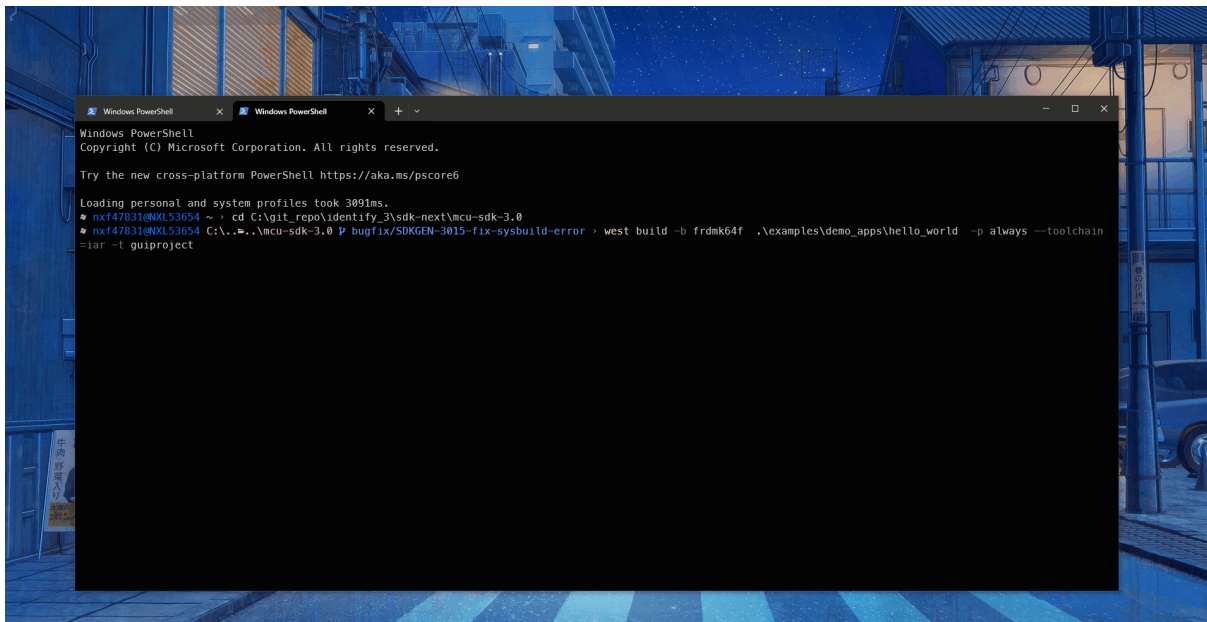
```
west debug -r linkserver
```

Work with IDE Project The above build functionalities are all with CLI. If you want to use the toolchain IDE to work to enjoy the better user experience especially for debugging or you are already used to develop with IDEs like IAR, MDK, Xtensa and CodeWarrior in the embedded world, you can play with our IDE project generation functionality.

This is the cmd to generate the evkbmimxrt1170 hello_world IAR IDE project files.

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config_↵  
↵flexspi_nor_debug -p always -t guiproject
```

By default, the IDE project files are generated in mcuxsdk/build/<toolchain> folder, you can open the project file with the IDE tool to work:



Note, please follow the [Installation](#) to setup the environment especially make sure that [ruby](#) has been installed.

1.4 Release Notes

1.4.1 MCUXpresso SDK Release Notes

Overview

The MCUXpresso SDK is a comprehensive software enablement package designed to simplify and accelerate application development with Arm Cortex-M-based devices from NXP, including its general purpose, crossover and Bluetooth-enabled MCUs. MCUXpresso SW and Tools for DSC

further extends the SDK support to current 32-bit Digital Signal Controllers. The MCUXpresso SDK includes production-grade software with integrated RTOS (optional), integrated enabling software technologies (stacks and middleware), reference software, and more.

In addition to working seamlessly with the MCUXpresso IDE, the MCUXpresso SDK also supports and provides example projects for various toolchains. The Development tools chapter in the associated Release Notes provides details about toolchain support for your board. Support for the MCUXpresso Config Tools allows easy cloning of existing SDK examples and demos, allowing users to leverage the existing software examples provided by the SDK for their own projects.

Underscoring our commitment to high quality, the MCUXpresso SDK is MISRA compliant and checked with Coverity static analysis tools. For details on MCUXpresso SDK, see [MCUXpresso-SDK: Software Development Kit for MCUXpresso](#).

MCUXpresso SDK

As part of the MCUXpresso software and tools, MCUXpresso SDK is the evolution of Kinetis SDK, includes support for LPC, DSC, PN76, and i.MX System-on-Chip (SoC). The same drivers, APIs, and middleware are still available with support for Kinetis, LPC, DSC, and i.MX silicon. The MCUXpresso SDK adds support for the MCUXpresso IDE, an Eclipse-based toolchain that works with all MCUXpresso SDKs. Easily import your SDK into the new toolchain to access to all of the available components, examples, and demos for your target silicon. In addition to the MCUXpresso IDE, support for the MCUXpresso Config Tools allows easy cloning of existing SDK examples and demos, allowing users to leverage the existing software examples provided by the SDK for their own projects.

In order to maintain compatibility with legacy Freescale code, the filenames and source code in MCUXpresso SDK containing the legacy Freescale prefix FSL has been left as is. The FSL prefix has been redefined as the NXP Foundation Software Library.

Development tools

The MCUXpresso SDK was tested with following development tools. Same versions or above are recommended.

- MCUXpresso IDE, Rev. 25.06.xx
- IAR Embedded Workbench for Arm, version is 9.60.4
- Keil MDK, version is 5.41
- MCUXpresso for VS Code v25.06
- GCC Arm Embedded Toolchain 14.2.x

Supported development systems

This release supports board and devices listed in following table. The board and devices in bold were tested in this release.

Development boards	MCU devices
FRDM-RW612	RW610ETA2I, RW610HNA2I, RW610UKA2I, RW612ETA2I , RW612HNA2I, RW612UKA2I

MCUXpresso SDK release package

The MCUXpresso SDK release package content is aligned with the silicon subfamily it supports. This includes the boards, CMSIS, devices, middleware, and RTOS support.

Device support The device folder contains the whole software enablement available for the specific System-on-Chip (SoC) subfamily. This folder includes clock-specific implementation, device register header files, device register feature header files, and the system configuration source files. Included with the standard SoC support are folders containing peripheral drivers, toolchain support, and a standard debug console. The device-specific header files provide a direct access to the microcontroller peripheral registers. The device header file provides an overall SoC memory mapped register definition. The folder also includes the feature header file for each peripheral on the microcontroller. The toolchain folder contains the startup code and linker files for each supported toolchain. The startup code efficiently transfers the code execution to the `main()` function.

Board support The boards folder provides the board-specific demo applications, driver examples, and middleware examples.

Demo application and other examples The demo applications demonstrate the usage of the peripheral drivers to achieve a system level solution. Each demo application contains a readme file that describes the operation of the demo and required setup steps. The driver examples demonstrate the capabilities of the peripheral drivers. Each example implements a common use case to help demonstrate the driver functionality.

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

CMSIS DSP Library The MCUXpresso SDK is shipped with the standard CMSIS development pack, including the prebuilt libraries.

memfault-firmware-sdk memfault-firmware-sdk

Wireless Connectivity Framework The Connectivity Framework is a software component that provides hardware abstraction modules to the upper layer connectivity stacks and components. It also provides a list of services and APIs, such as, Low power, Over the Air (OTA) Firmware update, File System, Security, Sensors, Serial Connectivity Interface (FSCI), and others. The Connectivity Framework modules are located in the *middleware\wireless\framework SDK* folder.

wpa_supplicant-rtos NXP Wi-Fi WPA Supplicant

Wireless EdgeFast Bluetooth PAL For more information, see the MCUXpresso SDK EdgeFast Bluetooth Protocol Abstraction Layer User's Guide.

Ethermind BT/BLE Stack nxp_bt_ble_stack

coreHTTP coreHTTP

NXP Wi-Fi The MCUXpresso SDK provides driver for NXP Wi-Fi external modules. The Wi-Fi driver is integrated with LWIP TCPIP stack and demonstrated with several network applications (iperf and AWS IoT).

For more information, see Getting Started with NXP based Wireless Modules and i.MX RT Platform Running on RTOS (document: UM11441).

USB Type-C PD Stack See the *MCUXpresso SDK USB Type-C PD Stack User's Guide* (document MCUXSDKUSBDUG) for more information

USB Host, Device, OTG Stack See the *MCUXpresso SDK USB Stack User's Guide* (document MCUXSDKUSBSUG) for more information.

TinyCBOR Concise Binary Object Representation (CBOR) Library

TF-M Trusted Firmware - M Library

PSA Test Suite Arm Platform Security Architecture Test Suite

Mbed Crypto Mbed Crypto library

PKCS#11 The PKCS#11 standard specifies an application programming interface (API), called “Cryptoki,” for devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a “cryptographic token”.

NXP IoT Agent NXP IoT Agent

MCU Boot Open source MCU Bootloader.

mbedTLS mbedtls SSL/TLS library v3.x

mbedTLS mbedtls SSL/TLS library v2.x

Voice Seeker (no AEC) VoiceSeeker is a multi-microphone voice control audio front-end signal processing solution. VoiceSeeker is not featuring acoustic echo cancellation (AEC).

lwIP The lwIP TCP/IP stack is pre-integrated with MCUXpresso SDK and runs on top of the MCUXpresso SDK Ethernet driver with Ethernet-capable devices/boards.

For details, see the *lwIP TCPIP Stack and MCUXpresso SDK Integration User's Guide* (document MCUXSDKLWIPUG).

lwIP is a small independent implementation of the TCP/IP protocol suite.

LVGL LVGL Open Source Graphics Library

llhttp HTTP parser llhttp

LittleFS LittleFS filesystem stack

FreeMASTER FreeMASTER communication driver for 32-bit platforms.

File systemFatfs The FatFs file system is integrated with the MCUXpresso SDK and can be used to access either the SD card or the USB memory stick when the SD card driver or the USB Mass Storage Device class implementation is used.

cJSON Ultralightweight JSON parser in ANSI C

AWS IoT Amazon Web Service (AWS) IoT Core SDK.

NXP PSA CRYPTO DRIVER PSA crypto driver for crypto library integration via driver wrappers

NXP ELS PKC ELS PKC crypto library

Release contents

Provides an overview of the MCUXpresso SDK release package contents and locations.

Deliverable	Location
Boards	INSTALL_DIR/boards
Demo Applications	INSTALL_DIR/boards/<board_name>/demo_apps
Driver Examples	INSTALL_DIR/boards/<board_name>/driver_examples
eIQ examples	INSTALL_DIR/boards/<board_name>/eIQ_examples
Board Project Template for MCUXpresso IDE NPW	INSTALL_DIR/boards/<board_name>/project_template
Driver, SoC header files, extension header files and feature header files, utilities	INSTALL_DIR/devices/<device_name>
CMSIS drivers	INSTALL_DIR/devices/<device_name>/cmsis_drivers
Peripheral drivers	INSTALL_DIR/devices/<device_name>/drivers
Toolchain linker files and startup code	INSTALL_DIR/devices/<device_name>/<toolchain_name>
Utilities such as debug console	INSTALL_DIR/devices/<device_name>/utilities
Device Project Template for MCUXpresso IDE NPW	INSTALL_DIR/devices/<device_name>/project_template
CMSIS Arm Cortex-M header files, DSP library source	INSTALL_DIR/CMSIS
Components and board device drivers	INSTALL_DIR/components
RTOS	INSTALL_DIR/rtos
Release Notes, Getting Started Document and other documents	INSTALL_DIR/docs
Tools such as shared cmake files	INSTALL_DIR/tools
Middleware	INSTALL_DIR/middleware

Known issues

This section lists the known issues, limitations, and/or workarounds.

SDK does not fully support the MCUX Config Tool

Currently, the pin/clock config files in the MCUXpresso SDK are not generated from the MCUXpresso Config Tool. As a result, there is an error reported during project import/clone from the SDK package. However, the Config Tool itself can work without any issue

Cloned project cannot build issue

Some cloned projects cannot build due to missing dependent component configuration. To fix the issue, you must correct the configuration manually in the cloned project.

The failed projects include nbu_bb_bm, wifi_cli_over_ble_wu, and ncp_device.

Romapi_flexspi example not working

The romapi_flexspi example fails to work with armgcc flash_debug target.

No UART simulated for hello_world_virtual_com

No UART peripheral is simulated for hello_world_virtual_com after downloading.

Limitations when creating a new FreeRTOS-based C/C++ project

Due to the missing component dependencies definition, there are several limitations when creating a new FreeRTOS-based C/C++ project in MCUXpresso IDE. When the **FreeRTOS kernel** component is selected (under Operating Systems/RTOS/Core menu), you must manually select the **FreeRTOS cm33 non trustzone port** component (under Middleware/RTOS menu) for projects without TrustZone. For FreeRTOS TrustZone projects creation, the support is not ready.

GPIO toggle operation not working on NCP SPI slave side with flash release build

Due to IDE compile optimization for the flash release target, the GPIO toggle operation on the NCP SPI slave side to indicate RX DMA ready or slave to master transfer is not working. The issue occurs because the GPIO toggle time is too short for the host to detect it.

El2go_agent_demo_wifi issue

The `el2go_agent_demo_wifi` project does not work on MCUXpresso for VSCode. However, Arm GCC standalone, MCUXpresso IDE, MDK, or IAR have no limitations.

Note: The above issue is not really a limitation of the SDK release, as it gets fixed by a toolchain update. The issue also partially applies to all other examples using TF-M.

When using MCUX IDE, do not specify a custom prefix or suffix while importing the example.

However, if it is a custom prefix or suffix is added, you must adapt the names of the library projects `el2go_mbedtls_lib` and `el2go_tfm_psa_ns_lib` in the `el2go_agent_demo_wifi_ns` project. To change the linker import paths to the right name, select **Project settings > C/C++ Build > Settings > MCU linker > Miscellaneous > Other Objects**.

Note: This limitation stays as it depends on the SDK generator fixes

Example mbedtls_benchmark may hang on some targets on devices with ELS acceleration

Some targets of ELS accelerated devices may experience runtime issues when run with the default configuration of the `mbedtls_benchmark` application.

Examples: `mbedtls_benchmark`

Affected toolchains: All

The freertos_lpuart example does not complete successfully

The example hangs after console output 'FreeRTOS LPUART driver example'.

Examples: `freertos_lpuart`

Affected toolchains: All

The example does not perform as expected (Ticks do not printed on the console or the application does not wake up from the sleep mode).

Examples: `freertos_tickless`

Affected toolchains: All

TF-M secure and EL2GO examples incorrect path in “Download extra image” with iar and mdk IDEs with Kex package

TF-M secure and EL2GO examples are missing the target path for ns binary in “extra download image” with iar and mdk IDEs

Examples: tfm_demo_s, tfm_psatest_s, tfm_regression_s, tfm_secureboot_s, el2go_agent_s, el2go_blob_test_s, el2go_import_blob_s, el2go_mqtt_demo_s **Affected toolchains:** mdk, iar **Affected platforms:** mcxn5xxevk, frdm-mcxn947, mcxn9xxevk, rdrw612bga, frdm-rw612 **Workaround:** There are two ways 1.) Flash secure and non secure bins via Jlink or SPSDK after the build with IDE and providing with correct paths of secure and non-secure binaries. or 2.) Add {target} debug/release in path of “Download extra image” for iar and for MDK in xxx_flashdownload.ini file.

1.5 ChangeLog

1.5.1 MCUXpresso SDK Changelog

Board Support Files

board

[25.06.00]

- Initial version

clock_config

[25.06.00]

- Initial version

pin_mux

[25.06.00]

- Initial version
-

CNS_ACOMP

[2.0.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

CNS_ADC

[2.2.1]

- Improvements
 - Fixed CERT-C issues.

[2.2.0]

- Improvements
 - Updated cns adc trigger sources.
 - Migrated cns adc trigger sources enumeration from cns_adc.h to device.h
 - Reserved single-end mode channel 15, differential mode channel 5, and channel 15.

[2.1.0]

- Bug Fixes
 - Fixed temperature measurement error, and provided ‘enableChop’ member to control the ADC chop.

[2.0.2]

- Bug Fixes
 - Fixed ADC scan channel misconfiguration issue.
 - Fixed violation of MISRA C-2012 rule 10.1 and rule 10.4.
- Improvements
 - Added new member “enableInputBufferChop” into “adc_config_t” to enable/disable input buffer chopper.

[2.0.1]

- Bug Fixes
 - Fixed MISRA-2012 rules.
 - * Rule 14.2, rule 10.3.

[2.0.0]

- Initial version.
-

CACHE64

[2.0.11]

- Bug Fixes
 - Fixed CERT INT30-C violations: check and guarantee address plus size is equal or smaller than UINT32_MAX.

[2.0.10]

- Improvements
 - Updated `CACHE64_InvalidateCacheByRange()`, `CACHE64_CleanCacheByRange()` and `CACHE64_CleanInvalidateCacheByRange()` to support some platforms that multiple regions in the memory map are remapped to create a continuous address space.

[2.0.9]

- Improvements
 - Removed `assert(false)` in `CACHE64_GetInstanceByAddr`.

[2.0.8]

- Improvements
 - Updated function `CACHE64_GetInstanceByAddr()` to support some devices that provide alias of cacheable memory section.

[2.0.7]

- Improvements
 - Check input parameter “size_byte” must be larger than 0.

[2.0.6]

- Bug Fixes
 - Fixed overflow for `CACHE64_GetInstanceByAddr()`/`CACHE64_CleanCacheByRange()`/`CACHE64_InvalidateCacheByRange()` APIs.

[2.0.5]

- Improvement
 - Made use of `FSL_FEATURE_CACHE64_CTRL_HAS_NO_WRITE_BUF` feature

[2.0.4]

- Improvement
 - Disable cache policy feature on SoC without `CACHE64_POLSEL` IP.
- Bug Fixes
 - Fixed doxygen issue.

[2.0.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.3.

[2.0.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3, 10.4 and 14.4.
 - Fixed doxygen issue.

[2.0.1]

- Improvements
 - Moved CLCR register configuration out of the while loop, it's unnecessary to repeat this operation.

[2.0.0]

- Initial version.
-

CDOG

[2.1.3]

- Re-design multiple instance IRQs and Clocks
- Add fix for RESTART command errata

[2.1.2]

- Support multiple IRQs
- Fix default CONTROL values

[2.1.1]

- Remove bit CONTROL[CONTROL_CTRL].

[2.1.0]

- Rename CWT to CDOG.

[2.0.2]

- Fix MISRA-2012 issues.

[2.0.1]

- Fix doxygen issues.

[2.0.0]

- Initial version.
-
-

COMMON

[2.6.0]

- Bug Fixes
 - Fix CERT-C violations.

[2.5.0]

- New Features
 - Added new APIs `InitCriticalSectionMeasurementContext`, `DisableGlobalIRQEx` and `EnableGlobalIRQEx` so that user can measure the execution time of the protected sections.

[2.4.3]

- Improvements
 - Enable irq's that mount under `irqsteer` interrupt extender.

[2.4.2]

- Improvements
 - Add the macros to convert peripheral address to secure address or non-secure address.

[2.4.1]

- Improvements
 - Improve for the macro redefinition error when integrated with `zephyr`.

[2.4.0]

- New Features
 - Added `EnableIRQWithPriority`, `IRQ_SetPriority`, and `IRQ_ClearPendingIRQ` for ARM.
 - Added `MSDK_EnableCpuCycleCounter`, `MSDK_GetCpuCycleCount` for ARM.

[2.3.3]

- New Features
 - Added `NETC` into status group.

[2.3.2]

- Improvements
 - Make driver `aarch64` compatible

[2.3.1]

- Bug Fixes
 - Fixed `MAKE_VERSION` overflow on 16-bit platforms.

[2.3.0]

- Improvements
 - Split the driver to common part and CPU architecture related part.

[2.2.10]

- Bug Fixes
 - Fixed the ATOMIC macros build error in cpp files.

[2.2.9]

- Bug Fixes
 - Fixed MISRA C-2012 issue, 5.6, 5.8, 8.4, 8.5, 8.6, 10.1, 10.4, 17.7, 21.3.
 - Fixed SDK_Malloc issue that not allocate memory with required size.

[2.2.8]

- Improvements
 - Included stddef.h header file for MDK tool chain.
- New Features:
 - Added atomic modification macros.

[2.2.7]

- Other Change
 - Added MECC status group definition.

[2.2.6]

- Other Change
 - Added more status group definition.
- Bug Fixes
 - Undef __VECTOR_TABLE to avoid duplicate definition in cmsis_clang.h

[2.2.5]

- Bug Fixes
 - Fixed MISRA C-2012 rule-15.5.

[2.2.4]

- Bug Fixes
 - Fixed MISRA C-2012 rule-10.4.

[2.2.3]

- New Features
 - Provided better accuracy of SDK_DelayAtLeastUs with DWT, use macro SDK_DELAY_USE_DWT to enable this feature.
 - Modified the Cortex-M7 delay count divisor based on latest tests on RT series boards, this setting lets result be closer to actual delay time.

[2.2.2]

- New Features
 - Added include RTE_Components.h for CMSIS pack RTE.

[2.2.1]

- Bug Fixes
 - Fixed violation of MISRA C-2012 Rule 3.1, 10.1, 10.3, 10.4, 11.6, 11.9.

[2.2.0]

- New Features
 - Moved SDK_DelayAtLeastUs function from clock driver to common driver.

[2.1.4]

- New Features
 - Added OTFAD into status group.

[2.1.3]

- Bug Fixes
 - MISRA C-2012 issue fixed.
 - * Fixed the rule: rule-10.3.

[2.1.2]

- Improvements
 - Add SUPPRESS_FALL_THROUGH_WARNING() macro for the usage of suppressing fallthrough warning.

[2.1.1]

- Bug Fixes
 - Deleted and optimized repeated macro.

[2.1.0]

- New Features
 - Added IRQ operation for XCC toolchain.
 - Added group IDs for newly supported drivers.

[2.0.2]

- Bug Fixes
 - MISRA C-2012 issue fixed.
 - * Fixed the rule: rule-10.4.

[2.0.1]

- Improvements
 - Removed the implementation of LPC8XX Enable/DisableDeepSleepIRQ() function.
 - Added new feature macro switch “FSL_FEATURE_HAS_NO_NONCACHEABLE_SECTION” for specific SoCs which have no noncacheable sections, that helps avoid an unnecessary complex in link file and the startup file.
 - Updated the align(x) to **attribute**(aligned(x)) to support MDK v6 armclang compiler.

[2.0.0]

- Initial version.
-

CRC

[2.1.1]

- Fix MISRA issue.

[2.1.0]

- Add CRC_WriteSeed function.

[2.0.2]

- Fix MISRA issue.

[2.0.1]

- Fixed KPSDK-13362. MDK compiler issue when writing to WR_DATA with -O3 optimize for time.

[2.0.0]

- Initial version.
-

CTIMER

[2.3.3]

- Bug Fixes
 - Fix CERT INT30-C INT31-C issue.
 - Make API CTIMER_SetupPwm and CTIMER_UpdatePwmDutycycle return fail if pulse width register overflow.

[2.3.2]

- Bug Fixes
 - Clear unexpected DMA request generated by RESET_PeripheralReset in API CTIMER_Init to avoid trigger DMA by mistake.

[2.3.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.7 and 12.2.

[2.3.0]

- Improvements
 - Added the CTIMER_SetPrescale(), CTIMER_GetCaptureValue(), CTIMER_EnableResetMatchChannel(), CTIMER_EnableStopMatchChannel(), CTIMER_EnableRisingEdgeCapture(), CTIMER_EnableFallingEdgeCapture(), CTIMER_SetShadowValue(), APIs Interface to reduce code complexity.

[2.2.2]

- Bug Fixes
 - Fixed SetupPwm() API only can use match 3 as period channel issue.

[2.2.1]

- Bug Fixes
 - Fixed use specified channel to setting the PWM period in SetupPwmPeriod() API.
 - Fixed Coverity Out-of-bounds issue.

[2.2.0]

- Improvements
 - Updated three API Interface to support Users to flexibly configure the PWM period and PWM output.
- Bug Fixes
 - MISRA C-2012 issue fixed: rule 8.4.

[2.1.0]

- Improvements
 - Added the CTIMER_GetOutputMatchStatus() API Interface.
 - Added feature macro for FSL_FEATURE_CTIMER_HAS_NO_CCR_CAP2 and FSL_FEATURE_CTIMER_HAS_NO_IR_CR2INT.

[2.0.3]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3, 10.4, 10.6, 10.7 and 11.9.

[2.0.2]

- New Features
 - Added new API “CTIMER_GetTimerCountValue” to get the current timer count value.
 - Added a control macro to enable/disable the RESET and CLOCK code in current driver.
 - Added a new feature macro to update the API of CTimer driver for lpc8n04.

[2.0.1]

- Improvements
 - API Interface Change
 - * Changed API interface by adding CTIMER_SetupPwmPeriod API and CTIMER_UpdatePwmPulsePeriod API, which both can set up the right PWM with high resolution.

[2.0.0]

- Initial version.
-

CNS_DAC

[2.1.1]

- Improvements
 - Fixed CERT-C issues.

[2.1.0]

- Improvements
 - Updated cns dac trigger sources.
 - Migrated cns dac trigger sources enumeration from cns_dac.h to device.h

[2.0.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules.

[2.0.0]

- Initial version.
-

LPC_DMA**[2.5.3]**

- Improvements
 - Add assert in DMA_SetChannelXferConfig to prevent XFERCOUNT value overflow.

[2.5.2]

- Bug Fixes
 - Use separate “SET” and “CLR” registers to modify shared registers for all channels, in case of thread-safe issue.

[2.5.1]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rule 11.6.

[2.5.0]

- Improvements
 - Added a new api DMA_SetChannelXferConfig to set DMA xfer config.

[2.4.4]

- Bug Fixes
 - Fixed the issue that DMA_IRQHandle might generate redundant callbacks.
 - Fixed the issue that DMA driver cannot support channel bigger then 32.
 - Fixed violation of the MISRA C-2012 rule 13.5.

[2.4.3]

- Improvements
 - Added features FSL_FEATURE_DMA_DESCRIPTOR_ALIGN_SIZEn/FSL_FEATURE_DMA0_DESCRIPTOR_ALIGN_SIZEn to support the descriptor align size not constant in the two instances.

[2.4.2]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rule 8.4.

[2.4.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 5.7, 8.3.

[2.4.0]

- Improvements
 - Added new APIs DMA_LoadChannelDescriptor/DMA_ChannelIsBusy to support polling transfer case.
- Bug Fixes
 - Added address alignment check for descriptor source and destination address.
 - Added DMA_ALLOCATE_DATA_TRANSFER_BUFFER for application buffer allocation.
 - Fixed the sign-compare warning.
 - Fixed violations of the MISRA C-2012 rules 18.1, 10.4, 11.6, 10.7, 14.4, 16.3, 20.7, 10.8, 16.1, 17.7, 10.3, 3.1, 18.1.

[2.3.0]

- Bug Fixes
 - Removed DMA_HandleIRQ prototype definition from header file.
 - Added DMA_IRQHandle prototype definition in header file.

[2.2.5]

- Improvements
 - Added new API DMA_SetupChannelDescriptor to support configuring wrap descriptor.
 - Added wrap support in function DMA_SubmitChannelTransfer.

[2.2.4]

- Bug Fixes
 - Fixed the issue that macro DMA_CHANNEL_CFER used wrong parameter to calculate DSTINC.

[2.2.3]

- Bug Fixes
 - Improved DMA driver Deinit function for correct logic order.
- Improvements
 - Added API DMA_SubmitChannelTransferParameter to support creating head descriptor directly.
 - Added API DMA_SubmitChannelDescriptor to support ping pong transfer.
 - Added macro DMA_ALLOCATE_HEAD_DESCRIPTOR/DMA_ALLOCATE_LINK_DESCRIPTOR to simplify DMA descriptor allocation.

[2.2.2]

- Bug Fixes
 - Do not use software trigger when hardware trigger is enabled.

[2.2.1]

- Bug Fixes
 - Fixed Coverity issue.

[2.2.0]

- Improvements
 - Changed API DMA_SetupDMADescriptor to non-static.
 - Marked APIs below as deprecated.
 - * DMA_PrepareTransfer.
 - * DMA_Submit transfer.
 - Added new APIs as below:
 - * DMA_SetChannelConfig.
 - * DMA_PrepareChannelTransfer.
 - * DMA_InstallDescriptorMemory.
 - * DMA_SubmitChannelTransfer.
 - * DMA_SetChannelConfigValid.
 - * DMA_DoChannelSoftwareTrigger.
 - * DMA_LoadChannelTransferConfig.

[2.0.1]

- Improvements
 - Added volatile for DMA descriptor member xfercfg to avoid optimization.

[2.0.0]

- Initial version.
-

DMIC**[2.3.3]**

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8.

[2.3.2]

- New Features
 - Supported 4 channels in driver.

[2.3.1]

- Bug Fixes
 - Fixed the issue that DMIC_EnableChannelDma and DMIC_EnableChannelFifo did not clean relevant bits.

[2.3.0]

- Improvements
 - Added new apis DMIC_ResetChannelDecimator/DMIC_EnableChannelGlobalSync/DMIC_DisableChannelGlobalSync

[2.2.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 14.4, 17.7, 10.4, 10.3, 10.8, 14.3.

[2.2.0]

- Bug Fixes
 - Corrected the usage of feature FSL_FEATURE_DMIC_IO_HAS_NO_BYPASS.

[2.1.1]

- Improvements
 - Added feature FSL_FEATURE_DMIC_HAS_NO_IOCFG for IOCFG register.

[2.1.0]

- New Features
 - Added API DMIC_EnableChannelInterrupt/DMIC_EnableChannelDma to replace API DMIC_SetOperationMode.
 - Added API DMIC_SetIOCFG and marked DMIC_ConfigIO as deprecated.
 - Added API DMIC_EnableChannelSignExtend to support sign extend feature.

[2.0.5]

- Improvements
 - Changed some parameters' value of DMIC_FifoChannel API, such as enable, resetn, and trig_level. This is not possible for the current code logic, so it improves the DMIC_FifoChannel logic and fixes incorrect math logic.

[2.0.4]

- Bug Fixes
 - Fixed the issue that DMIC DMA driver(ver2.0.3) did not support calling DMIC_TransferReceiveDMA in DMA callback as it did before version 2.0.3. But calling DMIC_TransferReceiveDMA in callback is not recommended.

[2.0.3]

- New Features
- Supported linked transfer in DMIC DMA driver.
- Added new API DMIC_EnableChannelFifo/DMIC_DoFifoReset/DMIC_InstallDMADescriptor.

[2.0.2]

- New Features
 - Supported more channels in driver.

[2.0.1]

- New Features
 - Added a control macro to enable/disable the RESET and CLOCK code in current driver.

[2.0.0]

- Initial version.
-

DMIC_DMA**[2.4.1]**

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8.

[2.4.0]

- Bug Fixes
 - Fixed the issue that DMIC_TransferAbortReceiveDMA can not disable dmic and dma request issue.

[2.3.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3.

[2.3.0]

- Refer DMIC driver change log 2.0.1 to 2.3.0
-

ENET

[2.9.2]

- Bug Fixes
 - RGMII mode is (temporarily) disabled before selecting between 10/100-Mbit/s and 1000-Mbit/s modes of operation. The bit RGMII_EN of RCR register must not be set while changing ECR register's speed bit, otherwise there is a possibility of ENET IP ending in an incorrect state.

[2.9.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 8.4, 10.4.

[2.9.0]

- Bug Fixes
 - Enabled collection of transfer statistics, so the function ENET_GetStatistics does not always return zeroes.
- New Features
 - Added new function ENET_EnableStatistics to enable/disable collection of transfer statistics.
 - Added new function ENET_ResetStatistics to reset transfer statistics.
- Improvements
 - Renamed the function ENET_ResetHareware to ENET_ResetHardware.

[2.8.0]

- New Features
 - Added the function to reset hardware on certain devices.

[2.7.1]

- Bug Fixes
 - Fixed the issue that free wrong buffer address when one frame stores in multiple buffers and memory pool is not enough to allocate these buffers to receive one complete frame.

[2.7.0]

- Improvements
 - Deleted deprecated zero copy Tx/Rx functions and set callback function which can be configured in ENET_Init.
 - Moved the Rx zero copy buffer allocation to Rx BD initialization function to reduce unnecessary looping code.
- Bug Fixes

- Fixed the issue that predefined Rx buffers which should not be used when enabling Rx zero copy are still be handled by cache operation, it causes hardfault on some platforms.
- Fixed the issue that zero-copy Rx function doesn't check Rx length of 0 in the BD with EMPTY bit is 0, it may occur in the corner case reported by customer. Not sure how it turns out, consider it as an ENET IP issue and drop this abnormal BD.

[2.6.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 11.6.

[2.6.2]

- Improvements
 - Changed ENET1_MAC0_Rx_Tx_Done0_DriverIRQHandler/ENET1_MAC0_Rx_Tx_Done1_DriverIRQHandler to ENET1_MAC0_Rx_Tx_Done1_DriverIRQHandler/ENET1_MAC0_Rx_Tx_Done2_DriverIRQHandler which represent ring 1 and ring 2.

[2.6.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3, 10.4, 10.7, 11.6, 11.8.

[2.6.0]

- Improvements
 - Added MDIO access wrapper APIs for ease of use.
 - Fixed the build warning introduced by 64-bit compatibility patch.

[2.5.4]

- Improvements
 - Made the driver compatible with 64-bit platforms.

[2.5.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 11.6.

[2.5.2]

- Improvements
 - Updated the TXIC/RXIC register handling code according to the new header file.

[2.5.1]

- Bug Fixes
 - Fixed document typo.

[2.5.0]

- Bug Fixes
 - Fixed the SendFrame/SendFrameZeroCopy functions issue with scattered buffers.
 - Updated the formula of MDC calculation.
 - Used a feature macro to distinguish the old IP design from the new design, because old IP design always reads a value zero from ATCR->CAPTURE bit. For old IP, driver calculates and wait the necessary delay cycles after setting ATCR->CAPTURE then gets the timestamp value.
- New Features
 - Added new zero copy Tx/Rx function.
 - New zero copy Tx function combines scattered and contiguous Tx buffer in one API, it also supports more Tx features which buffer descriptor supports but previous Tx function doesn't support.
 - New zero copy Rx function use dynamic buffer mechanism and simpler interface.
- Improvements
 - Corrected the interrupt handler for PTP timestamp IRQ and PTP1588 event IRQ since platform difference.
 - Added missing IRQ handlers for PTP1588 events on some platforms.
 - Corrected the max Tx frame length verification, it will not depend on a fixed macro. The ENET_FRAME_MAX_FRAMELEN is only an default value for driver, application can configure it. Driver calculates the limitation with the max frame length in register which may takes extended 4 or 8 bytes VLAN tag if VLAN/SVLAN enables.
 - Deleted deprecated Clause 45 read/write legacy APIs.

[2.4.3]

- Improvements
 - Aligned the IRQ handler name with header file.

[2.4.2]

- Bug Fixes
 - Fixed the MISRA issue of speculative out-of-bounds access.

[2.4.1]

- Bug Fixes
 - Fixed the PTP time capture issue.

[2.4.0]

- Improvements
 - Exposed API ENET_ReclaimTxDescriptor for user application to reclaim tx descriptors in their application.
 - Added counter to record multicast hash conflict in struct _enet_handle, improved the situation that one multicast group could be left by other conflict multicast address left operation.

- Improved concurrent usage of relaim and send frame operation.

[2.3.4]

- Bug Fixes
 - Fixed the issue that interrupt handler only checks the interrupt event flag but not checks interrupt mask flag.

[2.3.3]

- Bug Fixes
 - Fixed the issue that some compilers may choose the memcpy with 4-bit aligned address limitation due to the type of address pointer is 'unsigned int *', the data address doesn't have to be 4-bit aligned.

[2.3.2]

- New Features
 - Added the feature that ENET driver can be used in the platform which integrates both 10/100M and 1G ENET IP.
 - Deleted duplicated code about ARM errata 838869 in first/second level IRQ handler.

[2.3.1]

- Improvements
 - Added function pointer checking in IRQ handler to make sure code can be used even it runs into the interrupt when the second level interrupt handler is NULL.

[2.3.0]

- Bug Fixes
 - Fixed the issue that clause 45 MDIO read/write API doesn't check the transmission over status between two transmissions.
 - Fixed violations of the MISRA C-2012 rules 2.2,10.3,10.4,10.7,11.6,11.8,13.5,14.4,15.7,17.7.
- New Features
 - Added APIs to support send/receive frame with Zero-Copy.
- Improvements
 - Separated the clock configuration from module configuration when init and deinit.
 - Added functions to set second level interrupt handler.
 - Provided new function to get 1588 timer count without disabling interrupt.
 - Improved timestamp controlling, deleted all old timestamp management APIs and data structures.
 - Merged the single/multiple ring(s) APIs, now these APIs can handle both.
 - Used base and index to control buffer descriptor, aligned with qos and lpc enet driver.

[2.2.6]

- Bug Fixes
 - Updated MII speed formula referring to the manual.

[2.2.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1, 10.3, 10.4, 10.6, 10.7, 11.6, 11.9, 13.5, 14.4, 16.4, 17.7, 21.15, 3.1, 8.4.
 - Changed to use `ARRAY_SIZE(s_enetBases)` as the array size for `s_ENETHandle`, fixed the hardfault issue for using some ENET instance when `ARRAY_SIZE(s_enetBases)` is not same as `FSL_FEATURE_SOC_ENET_COUNT`.

[2.2.4]

- Improvements
 - Added call to Data Synchronization Barrier instruction before activating Tx/Rx buffer descriptor to ensure previous data update is completed.
 - Improved `ENET_TransmitIRQHandler` to store timestamps for multiple transmit buffer descriptors.
 - Bug Fixes
 - Fixed the issue that `ENET_Ptp1588GetTimer` did not handle the timer wrap situation.

[2.2.3]

- Improvements
 - Improved data buffer cache maintenance in the ENET driver.

[2.2.2]

- New Features
 - Added APIs for extended multi-ring support.
 - Added the AVB configure API for extended AVB feature support.

[2.2.1]

- Improvements
 - Changed the input data pointer attribute to `const` in `ENET_SendFrame()`.

[2.1.1]

- New Features
 - Added the extended MDIO IEEE802.3 Clause 45 MDIO format SMI command APIs.
 - Added the extended interrupt coalescing feature.
- Improvements
 - Combined all storage operations in the `ENET_Init` to `ENET_SetHandler` API.

[2.0.1]

- Bug Fixes
 - Used direct transmit busy check when doing data transmit.
- Miscellaneous Changes
 - Updated IRQ handler work flow.
 - Changed the TX/RX interrupt macro from kENET_RxByteInterrupt to kENET_RxBufferInterrupt, from kENET_TxByteInterrupt to kENET_TxBufferInterrupt.
 - Deleted unnecessary parameters in ENET handler.

[2.0.0]

- Initial version.
-

FLEXCOMM**[2.0.2]**

- Bug Fixes
 - Fixed typos in FLEXCOMM15_DriverIRQHandler().
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.
- Improvements
 - Added instance calculation in FLEXCOMM16_DriverIRQHandler() to align with Flexcomm 14 and 15.

[2.0.1]

- Improvements
 - Added more IRQHandler code in drivers to adapt new devices.

[2.0.0]

- Initial version.
-

FLEXSPI**[2.7.0]**

- New Features
 - Added new API to support address remapping.

[2.6.4]

- Improvements
 - Added new macro “FSL_SDK_ENABLE_FLEXSPI_RESET_CONTROL” to support driver level reset control.

[2.6.3]

- Bug Fixes
 - Fixed an issue which cause IPCR1[IPAREN] cleared by mistake.

[2.6.2]

- Bug Fixes
 - Wait Bus IDLE before operation of FLEXSPI_SoftwareReset(), FLEXSPI_TransferBlocking() and FLEXSPI_TransferNonBlocking().

[2.6.1]

- Bug Fixes
 - Updated code of reset peripheral.
 - Updated FLEXSPI_UpdateLUT() to check if input lut address is not in Flexspi AMBA region.
 - Updated FLEXSPI_Init() to check if input AHB buffer size exceeded maximum AHB size.

[2.6.0]

- New Features
 - Added new API to set AHB memory-mapped flash base address.
 - Added support of DLLxCR[REFPHASEGAP] bit field, it is recommended to set it as 0x2 if DLL calibration is enabled.

[2.5.1]

- Bugfixes
 - Fixed handling of W1C bits in the INTR register
 - Removed FIFO resets from FLEXSPI_CheckAndClearError
 - FLEXSPI_TransferBlocking is observing IPCMDDONE and then fetches the final status of the transfer
 - Fixed issue that FLEXSPI2_DriverIRQHandler not defined.

[2.5.0]

- Improvements
 - Supported word un-aligned access for write/read blocking/non-blocking API functions.
 - Fixed dead loop issue in DLL update function when using FRO clock source.
 - Fixed violations of the MISRA C-2012 Rule 10.3.

[2.4.0]

- Improvements
 - Isolated IP command parallel mode and AHB command parallel mode using feature MACRO.
 - Supported new column address shift feature for external memory.

[2.3.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 14.2.

[2.3.4]

- Bug Fixes
 - Updated `flexspi_config_t` structure and `FlexSPI_Init` to support new feature `FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_COMBINATION`.

[2.3.3]

- Bug Fixes
 - Removed feature `FSL_FEATURE_FLEXSPI_DQS_DELAY_PS` for DLL delay setting. Changed to use feature `FSL_FEATURE_FLEXSPI_DQS_DELAY_MIN` to set slave delay target as 0 for DLL enable and clock frequency higher than 100MHz.

[2.3.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 8.4, 8.5, 10.1, 10.3, 10.4, 11.6 and 14.4.

[2.3.1]

- Bug Fixes
 - Wait for bus to be idle before using it as access to external flash with new setting in `FLEXSPI_SetFlashConfig()` API.
 - Fixed the potential buffer overread and Tx FIFO overwrite issue in `FLEXSPI_WriteBlocking`.

[2.3.0]

- New Features
 - Added new API `FLEXSPI_UpdateDllValue` for users to update DLL value after updating `flexspi` root clock.
 - Corrected grammatical issues for comments.
 - Added support for new feature `FSL_FEATURE_FLEXSPI_DQS_DELAY_PS` in DLL configuration.

[2.2.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3 and 10.4.
 - Updated `_flexspi_command` from named enumerator into anonymous enumerator.

[2.2.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3, 10.4, 10.8, 11.9, 14.4, 15.7, 16.4, 17.7, 7.3.
 - Fixed IAR build warning Pe167.
 - Fixed the potential buffer overwrite and Rx FIFO overread issue in FLEXSPI_ReadBlocking.

[2.2.0]

- Bug Fixes
 - Fixed flag name typos: kFLEXSPI_IpTxFifoWatermarkEmpltyFlag to kFLEXSPI_IpTxFifoWatermarkEmptyFlag; kFLEXSPI_IpCommandExcutionDoneFlag to kFLEXSPI_IpCommandExecutionDoneFlag.
 - Fixed comments typos such as sequencen->sequence, levle->level.
 - Fixed FLSHCR2[ARDSEQID] field clean issue.
 - Updated flexspi_config_t structure and FlexSPI_Init to support new feature FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_ATDFEN and FSL_FEATURE_FLEXSPI_HAS_NO_MCR0_ARDFEN.
 - Updated flexspi_flags_t structure to support new feature FSL_FEATURE_FLEXSPI_HAS_INTEN_AHBBUSERROREN.

[2.1.1]

- Improvements
 - Defaulted enable prefetch for AHB RX buffer configuration in FLEXSPI_GetDefaultConfig, which is align with the reset value in AHBRXBUFxCR0.
 - Added software workaround for ERR011377 in FLEXSPI_SetFlashConfig; added some delay after DLL lock status set to ensure correct data read/write.

[2.1.0]

- New Features
 - Added new API FLEXSPI_UpdateRxSampleClock for users to update read sample clock source after initialization.
 - Added reset peripheral operation in FLEXSPI_Init if required.

[2.0.5]

- Bug Fixes
 - Fixed FLEXSPI_UpdateLUT cannot do partial update issue.

[2.0.4]

- Bug Fixes
 - Reset flash size to zero for all ports in FLEXSPI_Init; fixed the possible out-of-range flash access with no error reported.

[2.0.3]

- Bug Fixes
 - Fixed AHB receive buffer size configuration issue. The FLEXSPI_AHBRXBUFFCRO_BUFSZ field should configure 64 bits size, and currently the AHB receive buffer size is in bytes which means 8-bit, so the correct configuration should be `config->ahbConfig.buffer[i].bufferSize / 8`.

[2.0.2]

- New Features
 - Supported DQS write mask enable/disable feature during set FLEXSPI configuration.
 - Provided new API FLEXSPI_TransferUpdateSizeEDMA for users to update eDMA transfer size(SSIZE/DSIZE) per DMA transfer.
- Bug Fixes
 - Fixed invalid operation of FLEXSPI_Init to enable AHB bus Read Access to IP RX FIFO.
 - Fixed incorrect operation of FLEXSPI_Init to configure IP TX FIFO watermark.

[2.0.1]

- Bug Fixes
 - Fixed the flag clear issue and AHB read Command index configuration issue in FLEXSPI_SetFlashConfig.
 - Updated FLEXSPI_UpdateLUT function to update LUT table from any index instead of previous command index.
 - Added bus idle wait in FLEXSPI_SetFlashConfig and FLEXSPI_UpdateLUT to ensure bus is idle before any change to FlexSPI controller.
 - Updated interrupt API FLEXSPI_TransferNonBlocking and interrupt handle flow FLEXSPI_TransferHandleIRQ.
 - Updated eDMA API FLEXSPI_TransferEDMA.

[2.0.0]

- Initial version.
-

FLEXSPI DMA Driver

[2.2.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3, 10.4, 10.8.

[2.2.0]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 Rule 10.1, 10.3.
- New Features
 - Updated name of FLEXSPI_TransferGetTransferCountDMA API.

[2.1.1]

- New Features
 - Updated driver to support feature FSL_FEATURE_FLEXSPI_DMA_MULTIPLE_DES.

[2.1.0]

- Bug Fixes
 - Updated enumeration flexspi_dma_transfer_nsize_t and remove the unsupported items.
- New Features
 - Updated driver for deprecating the multiple linked descriptors inside FLEXSPI_TransferDMA, only up to one linked descriptor is needed according to hardware update.

[2.0.0]

- Initial version.
-

FMEAS

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issues fixed: rule 10.4, rule 10.8.

[2.1.0]

- Updated “FMEAS_GetFrequency”, “FMEAS_StartMeasure”, “FMEAS_IsMeasureComplete” API and add definition to match ASYNC_SYSCON.

[2.0.0]

- Initial version ported from LPCOpen.
-

GDMA

[2.0.3]

- Bug Fixes
 - Fixed MISRA C-2012 violation.

[2.0.2]

- Improvements
 - Changed to use FSL_FEATURE_GDMA_CHANNEL_NUM defined by feature header.

[2.0.1]

- Bug Fixes
 - Fixed MISRA C-2012 violation.

[2.0.0]

- Initial version.
-

GPIO**[2.1.7]**

- Improvements
 - Enhanced GPIO_PinInit to enable clock internally.

[2.1.6]

- Bug Fixes
 - Clear bit before set it within GPIO_SetPinInterruptConfig() API.

[2.1.5]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 3.1, 10.6, 10.7, 17.7.

[2.1.4]

- Improvements
 - Added API GPIO_PortGetInterruptStatus to retrieve interrupt status for whole port.
 - Corrected typos in header file.

[2.1.3]

- Improvements
 - Updated “GPIO_PinInit” API. If it has DIRCLR and DIRSET registers, use them at set 1 or clean 0.

[2.1.2]

- Improvements
 - Removed deprecated APIs.

[2.1.1]

- Improvements
 - API interface changes:
 - * Refined naming of APIs while keeping all original APIs, marking them as deprecated. Original APIs will be removed in next release. The main change is updating APIs with prefix of `_PinXXX()` and `_PortXXX`

[2.1.0]

- New Features
 - Added GPIO initialize API.

[2.0.0]

- Initial version.
-

I2C

[2.3.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.1.
 - Fixed issue that if master only sends address without data during I2C interrupt transfer, address nack cannot be detected.

[2.3.2]

- Improvement
 - Enable or disable timeout option according to `enableTimeout`.
- Bug Fixes
 - Fixed timeout value calculation error.
 - Fixed bug that the interrupt transfer cannot recover from the timeout error.

[2.3.1]

- Improvement
 - Before master transfer with transactional APIs, enable master function while disable slave function and vice versa for slave transfer to avoid the one affecting the other.
- Bug Fixes
 - Fixed bug in `I2C_SlaveEnable` that the slave enable/disable should not affect the other register bits.

[2.3.0]

- Improvement
 - Added new return codes `kStatus_I2C_EventTimeout` and `kStatus_I2C_SclLowTimeout`, and added the check for event timeout and SCL timeout in I2C master transfer.
 - Fixed bug in slave transfer that the address match event should be invoked before not after slave transmit/receive event.

[2.2.0]

- New Features
 - Added enumeration `_i2c_status_flags` to include all previous master and slave status flags, and added missing status flags.
 - Modified `I2C_GetStatusFlags` to get all I2C flags.
 - Added API `I2C_ClearStatusFlags` to clear all clearable flags not just master flags.
 - Modified master transactional APIs to enable bus event timeout interrupt during transfer, to avoid glitch on bus causing transfer hangs indefinitely.
- Bug Fixes
 - Fixed bug that status flags and interrupt enable masks share the same enumerations by adding enumeration `_i2c_interrupt_enable` for all master and slave interrupt sources.

[2.1.0]

- Bug Fixes
 - Fixed bug that during master transfer, when master is nacked during slave probing or sending subaddress, the return status should be `kStatus_I2C_Addr_Nak` rather than `kStatus_I2C_Nak`.
- Bug Fixes
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.4, 13.5.
- New Features
 - Added macro `I2C_MASTER_TRANSMIT_IGNORE_LAST_NACK`, so that user can configure whether to ignore the last byte being nacked by slave during master transfer.

[2.0.8]

- Bug Fixes
 - Fixed `I2C_MasterSetBaudRate` issue that `MSTSCLOW` and `MSTSCLHIGH` are incorrect when `MSTTIME` is odd.

[2.0.7]

- Bug Fixes
 - Two dividers, `CLKDIV` and `MSTTIME` are used to configure baudrate. According to reference manual, in order to generate 400kHz baudrate, the clock frequency after `CLKDIV` must be less than 2mHz. Fixed the bug that, the clock frequency after `CLKDIV` may be larger than 2mHz using the previous calculation method.
 - Fixed MISRA 10.1 issues.

- Fixed wrong baudrate calculation when feature FSL_FEATURE_I2C_PREPCLKFRG_8MHZ is enabled.

[2.0.6]

- New Features
 - Added master timeout self-recovery support for feature FSL_FEATURE_I2C_TIMEOUT_RECOVERY.
- Bug Fixes
 - Eliminated IAR Pa082 warning.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.

[2.0.5]

- Bug Fixes
 - Fixed wrong assignment for datasize in I2C_InitTransferStateMachineDMA.
 - Fixed wrong working flow in I2C_RunTransferStateMachineDMA to ensure master can work in no start flag and no stop flag mode.
 - Fixed wrong working flow in I2C_RunTransferStateMachine and added kReceive-DataBeginState in _i2c_transfer_states to ensure master can work in no start flag and no stop flag mode.
 - Fixed wrong handle state in I2C_MasterTransferDMAHandleIRQ. After all the data has been transferred or nak is returned, handle state should be changed to idle.
- Improvements
 - Rounded up the calculated divider value in I2C_MasterSetBaudRate.

[2.0.4]

- Improvements
 - Updated the I2C_WATI_TIMEOUT macro to unified name I2C_RETRY_TIMES
 - Updated the “I2C_MasterSetBaudRate” API to support baudrate configuration for feature QN9090.
- Bug Fixes
 - Fixed build warnning caused by uninitialized variable.
 - Fixed COVERITY issue of unchecked return value in I2C_RTOS_Transfer.

[2.0.3]

- Improvements
 - Unified the component full name to FLEXCOMM I2C(DMA/FREERTOS) driver.

[2.0.2]

- Improvements
 - In slave IRQ:
 1. Changed slave receive process to first set the I2C_SLVCTL_SLVCONTINUE_MASK to acknowledge the received data, then do data receive.
 2. Improved slave transmit process to set the I2C_SLVCTL_SLVCONTINUE_MASK immediately after writing the data.

[2.0.1]

- Improvements
 - Added I2C_WATI_TIMEOUT macro to allow users to specify the timeout times for waiting flags in functional API and blocking transfer API.

[2.0.0]

- Initial version.
-

I2S**[2.3.2]**

- Bug Fixes
 - Fixed warning for comparison between pointer and integer.

[2.3.1]

- Bug Fixes
 - Updated the value of TX/RX software transfer state machine after transfer contents are submitted to avoid race condition.

[2.3.0]

- Improvements
 - Added api I2S_InstallDMADescriptorMemory/I2S_TransferSendLoopDMA/I2S_TransferReceiveLoopDMA to support loop transfer.
 - Added api I2S_EmptyTxFifo to support blocking flush tx fifo.
 - Updated api I2S_TransferAbortDMA by removed the blocking flush tx fifo from this function.
- Bug Fixes
 - Removed the while loop in abort transfer function to fix the dead loop issue under specific user case.

[2.2.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 8.4.

[2.2.1]

- Improvements
 - Added feature FSL_FEATURE_FLEXCOMM_INSTANCE_I2S_SUPPORT_SECONDARY_CHANNELn for the SOC has parts of instance support secondary channel.
- Bug Fixes
 - Added volatile statement for the state variable of i2s_handle and enable the mainline channel pair before enable interrupt to avoid the issue of code execution reordering which may cause the interrupt generated unexpectedly.

[2.2.0]

- Improvements
 - Added 8/16/24 bits mono data format transfer support in I2S driver.
 - Added new apis I2S_SetBitClockRate.
- Bug Fixes
 - Fixed the PA082 build warning.
 - Fixed the sign-compare warning.
 - Fixed violations of the MISRA C-2012 rules 10.4, 10.8, 11.9, 10.1, 11.3, 13.5, 11.8, 10.3, 10.7.
 - Fixed the Operand don't affect result Coverity issue.

[2.1.0]

- Improvements
 - Added a feature for the FLEXCOMM which supports I2S and has interconnection with DMIC.
 - Used a feature to control PDMDATA instead of I2S_CFG1_PDMDATA.
 - Added member bytesPerFrame in i2s_dma_handle_t, used for DMA transfer width configure, instead of using sizeof(uint32_t) hardcode.
 - Used the macro provided by DMA driver to define the I2S DMA descriptor.
- Bug Fixes
 - Fixed the issue that I2S DMA driver always generated duplicate callback.

[2.0.3]

- New Features
 - Added a feature to remove configuration for the second channel on LPC51U68.

[2.0.2]

- New Features
 - Added ENABLE_IRQ handle after register I2S interrupt handle.

[2.0.1]

- Improvements
 - Unified the component full name to FLEXCOMM I2S (DMA) driver.

[2.0.0]

- Initial version.
-
-

I2S_DMA**[2.3.3]**

- Bug Fixes
 - Fixed data size limit does not match the macro DMA_MAX_TRANSFER_BYTES issue.

[2.3.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3.

[2.3.1]

- Refer I2S driver change log 2.0.1 to 2.3.1
-

IMU**[2.2.0]**

- New Features
 - Added IMU_BUSY_POLL_COUNT parameter to prevent infinite polling loops in IMU operations.
 - Added timeout mechanism to all polling loops in IMU driver code.
- Improvements
 - Enhanced error handling in blocking functions to return timeout status.
 - Updated documentation to clarify timeout behavior and return values.
 - Added IMU_ERR_TIMEOUT error code for timeout conditions.

[2.1.1]

- Bug Fixes
 - Fix MISRA C-2012 violations.
 - Fixed IMU_GetStatusFlags bug that returns wrong RX FIFO status.

[2.1.0]

- Improvements:
 - Updated API prototype, remove CIU2_Type from parameters.

[2.0.0]

- Initial version.
-

INPUTMUX

[2.0.9]

- Improvements
 - Use INPUTMUX_CLOCKS to initialize the inputmux module clock to adapt to multiple inputmux instances.
 - Modify the API base type from INPUTMUX_Type to void.

[2.0.8]

- Improvements
 - Updated a feature macro usage for function INPUTMUX_EnableSignal.

[2.0.7]

- Improvements
 - Release peripheral from reset if necessary in init function.

[2.0.6]

- Bug Fixes
 - Fixed the documentation wrong in API INPUTMUX_AttachSignal.

[2.0.5]

- Bug Fixes
 - Fixed build error because some devices has no sct.

[2.0.4]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rule 10.4, 12.2 in INPUTMUX_EnableSignal() function.

[2.0.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.4, 10.7, 12.2.

[2.0.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.4, 12.2.

[2.0.1]

- Support channel mux setting in INPUTMUX_EnableSignal().

[2.0.0]

- Initial version.
-
-
-

ITRC**[2.0.0]**

- Initial version.
-

LCDIC**[2.2.0]**

- New Features
 - Add software timeout when waiting for CMD done.

[2.1.0]

- New Features
 - Add separate APIs for send and receive data in non-blocking way.
- Others
 - Return error status when sending or receiving data larger than 0x40000, current driver doesn't support this.

[2.0.3]

- Bug Fixes
 - Fixed potential issue that clock may not be send out when sending data array.

[2.0.2]

- Bug Fixes
 - Fixed build error with MDK 5.37.

[2.0.1]

- Bug Fixes
 - Added delay after setting LCDIC_EN to make sure LCDIC is out of reset.

[2.0.0]

- Initial version.
-

LCDIC_DMA

[2.1.0]

- New Features
 - Add separate APIs for send and receive data.
- Others
 - Return error status when sending or receiving data larger than 0x40000, current driver doesn't support this.

[2.0.0]

- Initial version.
-

MRT

[2.0.5]

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.0.4]

- Improvements
 - Don't reset MRT when there is not system level MRT reset functions.

[2.0.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.1 and 10.4.
 - Fixed the wrong count value assertion in MRT_StartTimer API.

[2.0.2]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rule 10.4.

[2.0.1]

- Added control macro to enable/disable the RESET and CLOCK code in current driver.

[2.0.0]

- Initial version.
-
-

OSTIMER**[2.2.4]**

- Bug Fixes
 - Fixed CERT INT31-C violations.

[2.2.3]

- Improvements
 - Disable and clear pending interrupts before disabling the OSTIMER clock to avoid interrupts being executed when the clock is already disabled.

[2.2.2]

- Improvements
 - Support devices with different OSTIMER instance name.

[2.2.1]

- Improvements
 - Release peripheral from reset if necessary in init function.

[2.2.0]

- Improvements
 - Move the PMC operation out of the OSTIMER driver to board specific files.
 - Added low level APIs to control OSTIMER MATCH and interrupt.

[2.1.2]

- Bug Fixes
 - Fixed MISRA-2012 rule 10.8.

[2.1.1]

- Bug Fixes
 - removes the suffix 'n' for some register names and bit fields' names
- Improvements
 - Added HW CODE GRAY feature supported by CODE GRAY in SYSCTRL register group.

[2.1.0]

- Bug Fixes
 - Added a workaround to fix the issue that no interrupt was reported when user set smaller period.
 - Fixed violation of MISRA C-2012 rule 10.3 and 11.9.
- Improvements
 - Added return value for the two APIs to set match value.
 - * OSTIMER_SetMatchRawValue
 - * OSTIMER_SetMatchValue

[2.0.3]

- Bug Fixes
 - Fixed violation of MISRA C-2012 rule 10.3, 14.4, 17.7.

[2.0.2]

- Improvements
 - Added support for OSTIMER0

[2.0.1]

- Improvements
 - Removed the software reset function out of the initialization API.
 - Enabled interrupt directly instead of enabling deep sleep interrupt. Users need to enable the deep sleep interrupt in application code if needed.

[2.0.0]

- Initial version.
-

PINT

[2.2.0]

- Fixed
 - Fixed the issue that clear interrupt flag when it's not handled. This causes events to be lost.
- Changed
 - Used one callback for one PINT instance. It's unnecessary to provide different callbacks for all PINT events.

[2.1.13]

- Improvements
 - Added instance array for PINT to adapt more devices.
 - Used release reset instead of reset PINT which may clear other related registers out of PINT.

[2.1.12]

- Bug Fixes
 - Fixed coverity issue.

[2.1.11]

- Bug Fixes
 - Fixed MISRA C-2012 rule 10.7 violation.

[2.1.10]

- New Features
 - Added the driver support for MCXN10 platform with combined interrupt handler.

[2.1.9]

- Bug Fixes
 - Fixed MISRA-2012 rule 8.4.

[2.1.8]

- Bug Fixes
 - Fixed MISRA-2012 rule 10.1 rule 10.4 rule 10.8 rule 18.1 rule 20.9.

[2.1.7]

- Improvements
 - Added fully support for the SECPINT, making it can be used just like PINT.

[2.1.6]

- Bug Fixes
 - Fixed the bug of not enabling common pint clock when enabling security pint clock.

[2.1.5]

- Bug Fixes
 - Fixed issue for MISRA-2012 check.
 - * Fixed rule 10.1 rule 10.3 rule 10.4 rule 10.8 rule 14.4.
 - Changed interrupt init order to make pin interrupt configuration more reasonable.

[2.1.4]

- Improvements
 - Added feature to control distinguish PINT/SECPINT relevant interrupt/clock configurations for PINT_Init and PINT_Deinit API.
 - Swapped the order of clearing PIN interrupt status flag and clearing pending NVIC interrupt in PINT_EnableCallback and PINT_EnableCallbackByIndex function.
 - Bug Fixes
 - * Fixed build issue caused by incorrect macro definitions.

[2.1.3]

- Bug fix:
 - Updated PINT_PinInterruptClrStatus to clear PINT interrupt status when the bit is asserted and check whether was triggered by edge-sensitive mode.
 - Write 1 to IST corresponding bit will clear interrupt status only in edge-sensitive mode and will switch the active level for this pin in level-sensitive mode.
 - Fixed MISRA c-2012 rule 10.1, rule 10.6, rule 10.7.
 - Added FSL_FEATURE_SECPINT_NUMBER_OF_CONNECTED_OUTPUTS to distinguish IRQ relevant array definitions for SECPINT/PINT on lpc55s69 board.
 - Fixed PINT driver c++ build error and remove index offset operation.

[2.1.2]

- Improvement:
 - Improved way of initialization for SECPINT/PINT in PINT_Init API.

[2.1.1]

- Improvement:
 - Enabled secure pint interrupt and add secure interrupt handle.

[2.1.0]

- Added PINT_EnableCallbackByIndex/PINT_DisableCallbackByIndex APIs to enable/disable callback by index.

[2.0.2]

- Added control macro to enable/disable the RESET and CLOCK code in current driver.

[2.0.1]

- Bug fix:
 - Updated PINT driver to clear interrupt only in Edge sensitive.

[2.0.0]

- Initial version.
-
-

POWERQUAD**[2.2.0]**

- New Features
 - Added new API PQ_Arctan2Fixed.

[2.1.1]

- Bug Fixes
 - Remove PQ_WaitDone from PQ_ArctanFixed and PQ_ArctanhFixed because it is unnecessary.

[2.1.0]

- Improvements
 - Fixed typo issue for biquad related function name.
 - Changed operator from “%” into “&” to reduce heavy cycle for biquad functions.

[2.0.5]

- Improvements
 - Added a note in driver for FIR that powerquad has a hardware limitation, when using it for FIR increment calculation, the address of pSrc needs to be a continuous address.

[2.0.4]

- Improvements
 - Supported the platforms which don't have PowerQuad clock and reset control.

[2.0.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 8.4, 10.1, 10.3, 10.4, 10.6, and so on.

[2.0.2]

- Bug Fixes
 - Fixed array size issue in fsl_powerquad_data.h file.
 - Fixed vector function pipeline issue.

[2.0.1]

- Bug Fixes
 - Fixed build error in C++ mode.

[2.0.0]

- Initial version.
-
-
-

RTC

[2.2.0]

- New Features
 - Created new APIs for the RTC driver.
 - * RTC_EnableSubsecCounter
 - * RTC_GetSubsecValue

[2.1.3]

- Bug Fixes
 - Fixed issue that RTC_GetWakeupCount may return wrong value.

[2.1.2]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.1, 10.4 and 10.7.

[2.1.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3 and 11.9.

[2.1.0]

- Bug Fixes
 - Created new APIs for the RTC driver.
 - * RTC_EnableTimer
 - * RTC_EnableWakeUpTimerInterruptFromDPD
 - * RTC_EnableAlarmTimerInterruptFromDPD
 - * RTC_EnableWakeupTimer
 - * RTC_GetEnabledWakeupTimer
 - * RTC_SetSecondsTimerMatch

- * RTC_GetSecondsTimerMatch
- * RTC_SetSecondsTimerCount
- * RTC_GetSecondsTimerCount
- deprecated legacy APIs for the RTC driver.
 - * RTC_StartTimer
 - * RTC_StopTimer
 - * RTC_EnableInterrupts
 - * RTC_DisableInterrupts
 - * RTC_GetEnabledInterrupts

[2.0.0]

- Initial version.
-

SCTIMER

[2.5.1]

- Bug Fixes
 - Fixed bug in SCTIMER_SetupCaptureAction: When kSCTIMER_Counter_H is selected, events 12-15 and capture registers 12-15 CAPn_H field can't be used.

[2.5.0]

- Improvements
 - Add SCTIMER_GetCaptureValue API to get capture value in capture registers.

[2.4.9]

- Improvements
 - Supported platforms which don't have system level SCTIMER reset.

[2.4.8]

- Bug Fixes
 - Fixed the issue that the SCTIMER_UpdatePwmDutycycle() can't writes MATCH_H bit and RELOADn_H.

[2.4.7]

- Bug Fixes
 - Fixed the issue that the SCTIMER_UpdatePwmDutycycle() can't configure 100% duty cycle PWM.

[2.4.6]

- Bug Fixes
 - Fixed the issue where the H register was not written as a word along with the L register.
 - Fixed the issue that the SCTIMER_SetCOUNTValue() is not configured with high 16 bits in unify mode.

[2.4.5]

- Bug Fixes
 - Fix SCT_EV_STATE_STATEMSKn macro build error.

[2.4.4]

- Bug Fixes
 - Fix MISRA C-2012 issue 10.8.

[2.4.3]

- Bug Fixes
 - Fixed the wrong way of writing CAPCTRL and REGMODE registers in SCTIMER_SetupCaptureAction.

[2.4.2]

- Bug Fixes
 - Fixed SCTIMER_SetupPwm 100% duty cycle issue.

[2.4.1]

- Bug Fixes
 - Fixed the issue that MATCHn_H bit and RELOADn_H bit could not be written.

[2.4.0]

[2.3.0]

- Bug Fixes
 - Fixed the potential overflow issue of pulseperiod variable in SCTIMER_SetupPwm/SCTIMER_UpdatePwmDutycycle API.
 - Fixed the issue of SCTIMER_CreateAndScheduleEvent API does not correctly work with 32 bit unified counter.
 - Fixed the issue of position of clear counter operation in SCTIMER_Init API.
- Improvements
 - Update SCTIMER_SetupPwm/SCTIMER_UpdatePwmDutycycle to support generate 0% and 100% PWM signal.
 - Add SCTIMER_SetupEventActiveDirection API to configure event activity direction.

- Update SCTIMER_StartTimer/SCTIMER_StopTimer API to support start/stop low counter and high counter at the same time.
- Add SCTIMER_SetCounterState/SCTIMER_GetCounterState API to write/read counter current state value.
- Update APIs to make it meaningful.
 - * SCTIMER_SetEventInState
 - * SCTIMER_ClearEventInState
 - * SCTIMER_GetEventInState

[2.2.0]

- Improvements
 - Updated for 16-bit register access.

[2.1.3]

- Bug Fixes
 - Fixed the issue of uninitialized variables in SCTIMER_SetupPwm.
 - Fixed the issue that the Low 16-bit and high 16-bit work independently in SCTIMER driver.
- Improvements
 - Added an enumerable macro of unify counter for user.
 - * kSCTIMER_Counter_U
 - Created new APIs for the RTC driver.
 - * SCTIMER_SetupStateLdMethodAction
 - * SCTIMER_SetupNextStateActionwithLdMethod
 - * SCTIMER_SetCOUNTValue
 - * SCTIMER_GetCOUNTValue
 - * SCTIMER_SetEventInState
 - * SCTIMER_ClearEventInState
 - * SCTIMER_GetEventInState
 - Deprecated legacy APIs for the RTC driver.
 - * SCTIMER_SetupNextStateAction

[2.1.2]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.3, 10.4, 10.6, 10.7, 11.9, 14.2 and 15.5.

[2.1.1]

- Improvements
 - Updated the register and macro names to align with the header of devices.

[2.1.0]

- Bug Fixes
 - Fixed issue where SCT application level Interrupt handler function is occupied by SCT driver.
 - Fixed issue where wrong value for INSYNC field inside SCTIMER_Init function.
 - Fixed issue to change Default value for INSYNC field inside SCTIMER_GetDefaultConfig.

[2.0.1]

- New Features
 - Added control macro to enable/disable the RESET and CLOCK code in current driver.

[2.0.0]

- Initial version.
-

SDU

[1.0.0]

- Initial version.
-

SMARTCARD

[2.3.0]

- New features:
 - Added support for USIM

[2.2.2]

- Bug fix:
 - Fixed MISRA C-2012 rule 10.4.

[2.2.1]

- Bug fix:
 - Fixed IAR warnings Pa082 in smartcard_emvsim
 - Fixed MISRA issues
 - Fixed rules 10.1, 10.3, 10.4, 10.6, 10.7, 10.8, 14.4, 16.1, 16.3, 16.4, 17.7

[2.2.0]

- New features:
 - Updated to use RX/TX FIFO

[2.1.2]

- Provided time delay function which works in microseconds.
- Bug fix:
 - Changed event to semaphore in RTOS driver (KPSDK-11634).
 - Added check if de-initialized variables are not null iSMARTCARD_RTOS_Deinit() (KPSDK-8788).
 - Changed deactivation sequence iSMARTCARD_PHY_TDA8035_Deactivate() to properly stop the clockPOSCR-35).
 - Fixed timing issue with VSEL0/1 signals in smartcard TDA803driver (KPSDK-10160)

[2.1.1]

- New features:
 - Added default phy interface selection into smartcard RTOS drivers (KPSDK-9063).
 - Replaced smartcard_phy_ncn8025 driver by smartcard_phy_tda8035.
- Bug fix:
 - Fixed protocol timers activation sequences in smartcard_emvsim and smartcard_phy_tda8035 drivers during emv11 pre-certification tests (KPSDK-9170, KPSDK-9556).

[2.1.0]

- Initial version.
-

SPI**[2.3.2]**

- Bug Fixes
 - Fixed the txData from void * to const void * in transmit API

[2.3.1]

- Improvements
 - Changed SPI_DUMMYDATA to 0x00.

[2.3.0]

- Update version.

[2.2.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules.

[2.2.1]

- Bug Fixes
 - Fixed MISRA 2012 10.4 issue.
 - Added code to clear FIFOs before transfer using DMA.

[2.2.0]

- Bug Fixes
 - Fixed bug that slave gets stuck during interrupt transfer.

[2.1.1]

- Improvements
 - Added timeout mechanism when waiting certain states in transfer driver.
- Bug Fixes
 - Fixed MISRA 10.1, 5.7 issues.

[2.1.0]

- Bug Fixes
 - Fixed Coverity issue of incrementing null pointer in SPI_TransferHandleIRQInternal.
 - Eliminated IAR Pa082 warnings.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.
- New Features
 - Modified the definition of SPI_SSELPOL_MASK to support the socs that have only 3 SSEL pins.

[2.0.4]

- Bug Fixes
 - Fixed the bug of using read only mode in DMA transfer. In DMA transfer mode, if transfer->txData is NULL, code attempts to read data from the address of 0x0 for configuring the last frame.
 - Fixed wrong assignment of handle->state. During transfer handle->state should be kSPI_Busy rather than kStatus_SPI_Busy.
- Improvements
 - Rounded up the calculated divider value in SPI_MasterSetBaud.

[2.0.3]

- Improvements
 - Added “SPI_FIFO_DEPTH(base)” with more definition.

[2.0.2]

- Improvements
 - Unified the component full name to FLEXCOMM SPI(DMA/FREERTOS) driver.

[2.0.1]

- Changed the data buffer from uint32_t to uint8_t which matches the real applications for SPI DMA driver.
- Added dummy data setup API to allow users to configure the dummy data to be transferred.
- Added new APIs for half-duplex transfer function. Users can not only send and receive data by one API in polling/interrupt/DMA way, but choose either to transmit first or to receive first. Besides, the PCS pin can be configured as assert status in transmission (between transmit and receive) by setting the isPcsAssertInTransfer to true.

[2.0.0]

- Initial version.
-

SPI_DMA**[2.2.1]**

- Bug Fixes
 - Fixed MISRA 2012 11.6 issue..

[2.2.0]

- Improvements
 - Supported dataSize larger than 1024 data transmit.
-

TRNG**[2.0.18]**

- Bug fix:
 - TRNG health checks now done in software on RT5xx and RT6xx.

[2.0.17]

- New features:
 - Add support for RT700.

[2.0.16]

- Improvements:
 - Added support for Dual oscillator mode.

[2.0.15]

- Other changes:
 - Changed TRNG_USER_CONFIG_DEFAULT_XXX values according to latest recommended by design team.

[2.0.14]

- New features:
 - Add support for RW610 and RW612.

[2.0.13]

- Bug fix:
 - After deepsleep it might return error, added clearing bits in TRNG_GetRandomData() and generating new entropy.
 - Modified reloading entropy in TRNG_GetRandomData(), for some data length it doesn't reloading entropy correctly.

[2.0.12]

- Bug fix:
 - For KW34A4_SERIES, KW35A4_SERIES, KW36A4_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv8.

[2.0.11]

- Bug fix:
 - Add clearing pending errors in TRNG_Init().

[2.0.10]

- Bug Fix:
 - Fixed doxygen issues.

[2.0.9]

- Bug Fix:
 - Fix HIS_CCM metrics issues.

[2.0.8]

- Bug fix:
 - For K32L2A41A_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv4.

[2.0.7]

- Bug fix:
 - Fix MISRA 2004 issue rule 12.5.

[2.0.6]

- Bug fix:
 - For KW35Z4_SERIES set TRNG_USER_CONFIG_DEFAULT_OSC_DIV to kTRNG_RingOscDiv8.

[2.0.5]

- Improvements:
 - For FRQMIN, FRQMAX and OSCDIV, add possibility to use device specific preprocessor macro to define default value in TRNG user configuration structure.

[2.0.4]

- Bug Fix:
 - Fix MISRA-2012 issues.
 - * Rule 10.1, rule 10.3, rule 13.5, rule 16.1.

[2.0.3]

- Improvements:
 - update TRNG_Init to restart new entropy generation.

[2.0.2]

- Improvements:
 - fix MISRA issues
 - * Rule 14.4.

[2.0.1]

- New features:
 - Set default OSCDIV for Kinetis devices KL8x and KL28Z.
- Other changes:
 - Changed default OSCDIV for K81 to divide by 2.

[2.0.0]

- Initial version.
-

USART**[2.8.5]**

- Bug Fixes
 - Fixed race condition during call of USART_EnableTxDMA and USART_EnableRxDMA.

[2.8.4]

- Bug Fixes
 - Fixed exclusive access in USART_TransferReceiveNonBlocking and USART_TransferSendNonBlocking.

[2.8.3]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 10.3, 11.8.

[2.8.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 14.2.

[2.8.1]

- Bug Fixes
 - Fixed the Baud Rate Generator(BRG) configuration in 32kHz mode.

[2.8.0]

- New Features
 - Added the rx timeout interrupts and status flags of bus status.
 - Added new rx timeout configuration item in usart_config_t.
 - Added API USART_SetRxTimeoutConfig for rx timeout configuration.
- Improvements
 - When the calculated baudrate cannot meet user's configuration, lower OSR value is allowed to use.

[2.7.0]

- New Features
 - Added the missing interrupts and status flags of bus status.
 - Added the check of tx error, noise error framing error and parity error in interrupt handler.

[2.6.0]

- Improvements
 - Used separate data for TX and RX in usart_transfer_t.
- Bug Fixes
 - Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling USART_TransferReceiveNonBlocking, the received data count returned by USART_TransferGetReceiveCount is wrong.
- New Features
 - Added missing API USART_TransferGetSendCountDMA get send count using DMA.

[2.5.0]

- New Features
 - Added APIs `USART_GetRxFifoCount/USART_GetTxFifoCount` to get rx/tx FIFO data count.
 - Added APIs `USART_SetRxFifoWatermark/USART_SetTxFifoWatermark` to set rx/tx FIFO water mark.
- Bug Fixes
 - Fixed DMA transfer blocking issue by enabling tx idle interrupt after DMA transmission finishes.

[2.4.0]

- New Features
 - Modified `usart_config_t`, `USART_Init` and `USART_GetDefaultConfig` APIs so that the hardware flow control can be enabled during module initialization.
- Bug Fixes
 - Fixed MISRA 10.4 violation.

[2.3.1]

- Bug Fixes
 - Fixed bug that operation on `INTENSET`, `INTENCLR`, `FIFOINTENSET` and `FIFOINTENCLR` should use bitwise operation not ‘or’ operation.
 - Fixed bug that if rx interrupt occurs before TX interrupt is enabled and after `txDataSize` is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.
- Improvements
 - Added check for baud rate’s accuracy that returns `kStatus_USART_BaudrateNotSupport` when the best achieved baud rate is not within 3% error of configured baud rate.

[2.3.0]

- New Features
 - Added APIs to configure 9-bit data mode, set slave address and send address.
 - Modified `USART_TransferReceiveNonBlocking` and `USART_TransferHandleIRQ` to use 9-bit mode in multi-slave system.

[2.2.0]

- New Features
 - Added the feature of supporting USART working at 32 kHz clocking mode.
- Improvements
 - Modified `USART_TransferHandleIRQ` so that `txState` will be set to idle only when all data has been sent out to bus.
 - Modified `USART_TransferGetSendCount` so that this API returns the real byte count that USART has sent out rather than the software buffer status.

- Added timeout mechanism when waiting for certain states in transfer driver.
- Bug Fixes
 - Fixed MISRA 10.1 issues.
 - Fixed bug that operation on INTENSET, INTENCLR, FIFOINTENSET and FIFOINTENCLR should use bitwise operation not ‘or’ operation.
 - Fixed bug that if rx interrupt occurs before TX interrupt is enabled and after txData-Size is configured, the data will be sent early by mistake, thus TX interrupt will be enabled after data is sent out.

[2.1.1]

- Improvements
 - Added check for transmitter idle in USART_TransferHandleIRQ and USART_TransferSendDMACallback to ensure all the data would be sent out to bus.
 - Modified USART_ReadBlocking so that if more than one receiver errors occur, all status flags will be cleared and the most severe error status will be returned.
- Bug Fixes
 - Eliminated IAR Pa082 warnings.
 - Fixed MISRA issues.
 - * Fixed rules 10.1, 10.3, 10.4, 10.7, 10.8, 11.3, 11.6, 11.8, 11.9, 13.5.

[2.1.0]

- New Features
 - Added features to allow users to configure the USART to synchronous transfer(master and slave) mode.
- Bug Fixes
 - Modified USART_SetBaudRate to get more accurate configuration.

[2.0.3]

- New Features
 - Added new APIs to allow users to enable the CTS which determines whether CTS is used for flow control.

[2.0.2]

- Bug Fixes
 - Fixed the bug where transfer abort APIs could not disable the interrupts. The FIFOINTENSET register should not be used to disable the interrupts, so use the FIFOINTENCLR register instead.

[2.0.1]

- Improvements
 - Unified the component full name to FLEXCOMM USART (DMA/FREERTOS) driver.

[2.0.0]

- Initial version.
-

USART_DMA**[2.6.0]**

- Refer USART driver change log 2.0.1 to 2.6.0
-

UTICK**[2.0.5]**

- Improvements
 - Improved for SOC RW610.

[2.0.4]

- Bug Fixes
 - Fixed compile fail issue of no-supporting PD configuration in utick driver.

[2.0.3]

- Bug Fixes
 - Fixed violations of MISRA C-2012 rules: 8.4, 14.4, 17.7

[2.0.2]

- Added new feature definition macro to enable/disable power control in drivers for some devices have no power control function.

[2.0.1]

- Added control macro to enable/disable the CLOCK code in current driver.

[2.0.0]

- Initial version.
-

WWDT**[2.1.9]**

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rule 10.4.

[2.1.8]

- Improvements
 - Updated the “WWDT_Init” API to add wait operation. Which can avoid the TV value read by CPU still be 0xFF (reset value) after WWDT_Init function returns.

[2.1.7]

- Bug Fixes
 - Fixed the issue that the watchdog reset event affected the system from PMC.
 - Fixed the issue of setting watchdog WDPROTECT field without considering the backwards compatibility.
 - Fixed the issue of clearing bit fields by mistake in the function of WWDT_ClearStatusFlags.

[2.1.5]

- Bug Fixes
 - deprecated a unusable API in WWWDWT driver.
 - * WWDT_Disable

[2.1.4]

- Bug Fixes
 - Fixed violation of the MISRA C-2012 rules Rule 10.1, 10.3, 10.4 and 11.9.
 - Fixed the issue of the inseparable process interrupted by other interrupt source.
 - * WWDT_Init

[2.1.3]

- Bug Fixes
 - Fixed legacy issue when initializing the MOD register.

[2.1.2]

- Improvements
 - Updated the “WWDT_ClearStatusFlags” API and “WWDT_GetStatusFlags” API to match QN9090. WDTOF is not set in case of WD reset. Get info from PMC instead.

[2.1.1]

- New Features
 - Added new feature definition macro for devices which have no LCOK control bit in MOD register.
 - Implemented delay/retry in WWDT driver.

[2.1.0]

- Improvements
 - Added new parameter in configuration when initializing WWDT module. This parameter, which must be set, allows the user to deliver the WWDT clock frequency.

[2.0.0]

- Initial version.
-

1.6 Driver API Reference Manual

This section provides a link to the Driver API RM, detailing available drivers and their usage to help you integrate hardware efficiently.

[RW612](#)

1.7 Middleware Documentation

Find links to detailed middleware documentation for key components. While not all onboard middleware is covered, this serves as a useful reference for configuration and development.

1.7.1 Wireless Connectivity Framework

[framework](#)

1.7.2 MCU Boot

[mcuboot_opensource](#)

1.7.3 FreeMASTER

[freemaster](#)

1.7.4 AWS IoT

[aws_iot](#)

1.7.5 NXP Wi-Fi

[Wi-Fi](#)

1.7.6 FreeRTOS

[FreeRTOS](#)

1.7.7 Wireless EdgeFast Bluetooth PAL

edgefast_bluetooth

1.7.8 lwIP

lwip

1.7.9 File systemFatfs

fatfs

Chapter 2

RW612

2.1 ACOMP: Analog Comparator

`void ACOMP_Init(ACOMP_Type *base, const acomp_config_t *config)`

Initializes the module, including warm up time, response mode, inactive value and so on.

Parameters

- `base` – ACOMP peripheral base address.
- `config` – The pointer to the structure `acomp_config_t`.

`void ACOMP_GetDefaultConfig(acomp_config_t *config)`

Gets the default configuration of ACOMP module.

```
config->id = kACOMP_Acomp0;
config->enable = false;
config->warmupTime = kACOMP_WarmUpTime1us;
config->responseMode = kACOMP_SlowResponseMode;
config->inactiveValue = kACOMP_ResultLogicLow;
config->intTrigType = kACOMP_HighLevelTrig;
config->edgeDetectTrigSrc = kACOMP_EdgePulseDis;
config->outPinMode = kACOMP_PinOutDisable;
config->posInput = NULL;
config->negInput = NULL;
```

Parameters

- `config` – The pointer to the structure `acomp_config_t`.

`void ACOMP_Deinit(ACOMP_Type *base)`

De-initializes the module.

Parameters

- `base` – ACOMP peripheral base address.

`void ACOMP_SetInputConfig(ACOMP_Type *base, acomp_comparator_id_t id, const acomp_positive_input_config_t *posInput, const acomp_negative_input_config_t *negInput)`

Configures selected comparator's inputs, including input channel and hysteresis level.

Parameters

- `base` – ACOMP peripheral base address.

- `id` – The selected acomp comparator's id, please refer to `acomp_comparator_id_t`.
- `posInput` – The configuration of selected comparator's positive input, please refer to `acomp_positive_input_config_t`.
- `negInput` – The configuration of selected comparator's negative input, please refer to `acomp_negative_input_config_t`.

static inline void ACOMP_DoSoftwareReset(ACOMP_Type *base, *acomp_comparator_id_t* id)
Does software reset to the selected ACOMP module.

Parameters

- `base` – ACOMP peripheral base address.
- `id` – The selected acomp comparator's id, please refer to `acomp_comparator_id_t`.

static inline void ACOMP_Enable(ACOMP_Type *base, *acomp_comparator_id_t* id, bool enable)
Enables/Disables ACOMP module.

Parameters

- `base` – ACOMP peripheral base address.
- `id` – The selected acomp comparator's id, please refer to `acomp_comparator_id_t`.
- `enable` – Used to enable/disable module.
 - **true** Enable comparator instance.
 - **false** Disable comparator instance.

static inline void ACOMP_ResetClockDivider(ACOMP_Type *base)
Resets clock divider.

Parameters

- `base` – ACOMP peripheral base address.

static inline *acomp_result_logic_status_t* ACOMP_GetResult(ACOMP_Type *base,
acomp_comparator_id_t id)

Gets the selected acomp conversion result.

Parameters

- `base` – ACOMP peripheral base address.
- `id` – The selected acomp comparator's id, please refer to `acomp_comparator_id_t`.

Returns

The result of the selected acomp instance.

static inline void ACOMP_EnableInterrupts(ACOMP_Type *base, uint32_t interruptMask)
ACOMP Interrupt Control Interfaces.

Enables interrupts, including acomp0 asynchronized interrupt, acomp0 synchronized interrupt, acomp1 asynchronized interrupt, and acomp1 synchronized interrupt.

Parameters

- `base` – ACOMP peripheral base address.
- `interruptMask` – The OR'ed value of the interrupts to be enabled, please refer to `_acomp_interrupt_enable`.

```
static inline void ACOMP_DisableInterrupt(ACOMP_Type *base, uint32_t interruptMask)
```

Disables interrupts, including acomp0 asynchronized interrupt, acomp0 synchronized interrupt, acomp1 asynchronized interrupt, and acomp1 synchronized interrupt.

Parameters

- base – ACOMP peripheral base address.
- interruptMask – The OR'ed value of the interrupts to be disabled, please refer to `_accomp_interrupt_enable`.

```
uint32_t ACOMP_GetStatusFlags(ACOMP_Type *base)
```

ACOMP Status Flag Interfaces.

Gets status flags, such as ACOMP0 active status flags, ACOMP1 active status flags, and so on.

Parameters

- base – ACOMP peripheral base address.

Returns

The OR'ed value ACOMP status flags, please refer to `_accomp_status_flags` for details.

```
static inline void ACOMP_ClearStatusFlags(ACOMP_Type *base, uint32_t statusFlagMask)
```

Clears status flags that can be cleared by software.

Note: Only `kACOMP_Acomp0OutInterruptFlag`, `kACOMP_Acomp0OutAInterruptFlag`, `kACOMP_Acomp1OutInterruptFlag`, and `kACOMP_Acomp1OutAInterruptFlag` can be cleared by software.

Parameters

- base – ACOMP peripheral base address.
- statusFlagMask – The OR'ed value of the status flags that can be cleared.

```
enum _accomp_interrupt_enable
```

The enumeration of interrupts, including ACOMP0 synchrnized output interrupt, ACOMP0 asynchrnized output interrupt, ACOMP1 synchrnized output interrupt, and ACOMP1 asynchrnized output interrupt.

Values:

enumerator `kACOMP_Out0InterruptEnable`

ACOMP0 synchrnized output interrupt enable.

enumerator `kACOMP_OutA0InterruptEnable`

ACOMP0 asynchrnized output interrupt enable.

enumerator `kACOMP_Out1InterruptEnable`

ACOMP1 synchrnized output interrupt enable.

enumerator `kACOMP_OutA1InterruptEnable`

ACOMP1 asynchrnized output interrupt enable.

```
enum _accomp_status_flags
```

The enumeration of status flags, including ACOMP0 active staus flag, ACOMP1 active status flag, and so on.

Values:

enumerator `kACOMP_Acomp0ActiveFlag`

ACOMP0 active status flag, if this flag is set it means the ACOMP0 is active.

enumerator kACOMP__Acomp0OutInterruptFlag

ACOMP0 Synchronized output interrupt flags, this flag is set when ACOMP0 synchronized output changes from 0 to 1 and the corresponding interrupt is enabled.

enumerator kACOMP__Acomp0OutAInterruptFlag

ACOMP0 Asynchronized output interrupt flags, this flag is set when ACOMP0 asynchronized output changes from 0 to 1 and the corresponding interrupt is enabled.

enumerator kACOMP__Acomp0RawOutInterruptFlag

ACOMP0 raw synchroized output interrrupt flags.

enumerator kACOMP__Acomp0RawOutAInterruptFlag

ACOMP0 raw asynchroized output interrupt flags.

enumerator kACOMP__Acomp1ActiveFlag

ACOMP1 active status flag, if this flag is set it means the ACOMP0 is active.

enumerator kACOMP__Acomp1OutInterruptFlag

ACOMP1 Synchronized output interrupt flags, this flag is set when ACOMP1 synchroized output changes from 0 to 1 and the corresponding interrupt is enabled.

enumerator kACOMP__Acomp1OutAInterruptFlag

ACOMP1 Asynchronized output interrupt flags, this flag is set when ACOMP1 asynchronized output changes from 0 to 1 and the corresponding interrupt is enabled.

enumerator kACOMP__Acomp1RawOutInterruptFlag

ACOMP1 raw synchroized output interrrupt flags.

enumerator kACOMP__Acomp1RawOutAInterruptFlag

ACOMP1 raw asynchroized output interrupt flags.

enum _acomp_result_logic_status

ACOMP result logical status Type definition.

Values:

enumerator kACOMP__ResultLogicLow

The comparsion result is high logic.

enumerator kACOMP__ResultLogicHigh

The comparsion result is low logic.

enum _acomp_comparator_id

ACOMP comparator id.

Values:

enumerator kACOMP__Acomp0

Index for ACOMP0

enumerator kACOMP__Acomp1

Index for ACOMP1

enum _acomp_warm_up_time

The enumeration of wave up time.

Values:

enumerator kACOMP__WarmUpTime1us

Set wave-up time as 1us.

enumerator kACOMP__WarmUpTime2us

Set wave-up time as 2us.

enumerator kACOMP_WarmUpTime4us
Set wave-up time as 4us.

enumerator kACOMP_WarmUpTime8us
Set wave-up time as 8us.

enum _acomp_response_mode

The enumeration of response mode. The response mode will affect the delay from input to output.

Values:

enumerator kACOMP_SlowResponseMode
Slow response mode also called power mode 1.

enumerator kACOMP_MediumResponseMode
Medium response mode also called power mode 2.

enumerator kACOMP_FastResponseMode
Fast response mode also called power mode 3.

enum _acomp_interrupt_trigger_type

ACOMP interrupt trigger type definition.

Values:

enumerator kACOMP_LowLevelTrig
Low level trigger interrupt.

enumerator kACOMP_HighLevelTrig
High level trigger interrupt.

enumerator kACOMP_FallingEdgeTrig
Falling edge trigger interrupt.

enumerator kACOMP_RisingEdgeTrig
Rising edge trigger interrupt.

enum _acomp_edge_pulse_trig_source

ACOMP edge pulse trigger source type definition.

Values:

enumerator kACOMP_EdgePulseDis
edge pulse function is disable

enumerator kACOMP_EdgePulseRising
Rising edge can trigger edge pulse

enumerator kACOMP_EdgePulseFalling
Falling edge can trigger edge pulse

enumerator kACOMP_EdgePulseBothEdge
Both edge can trigger edge pulse

enum _acomp_pin_out_type

ACOMP synchronous/asynchronous output type to pin.

Values:

enumerator kACOMP_PinOutSyn
Enable ACOMP synchronous pin output

enumerator kACOMP__PinOutAsyn

Enable ACOMP asynchronous pin output

enumerator kACOMP__PinOutSynInverted

Enable ACOMP inverted synchronous pin output

enumerator kACOMP__PinOutAsynInverted

Enable ACOMP inverted asynchronous pin output

enumerator kACOMP__PinOutDisable

Disable ACOMP pin output

enum __acomp__positive__channel

ACOMP positive channel enumeration.

Values:

enumerator kACOMP__PosCh0

ACOMP Channel0 selection

enumerator kACOMP__PosCh1

ACOMP Channel1 selection

enumerator kACOMP__PosCh2

ACOMP Channel2 selection

enumerator kACOMP__PosCh3

ACOMP Channel3 selection

enumerator kACOMP__PosCh4

ACOMP Channel4 selection

enumerator kACOMP__PosCh5

ACOMP Channel5 selection

enumerator kACOMP__PosCh6

ACOMP Channel6 selection

enumerator kACOMP__PosCh7

ACOMP Channel7 selection

enumerator kACOMP__PosChDACA

DACA selection

enumerator kACOMP__PosChDACB

DACB selection

enum __acomp__negative__channel

ACOMP negative channel enumeration.

Values:

enumerator kACOMP__NegCh0

ACOMP Channel0 selection

enumerator kACOMP__NegCh1

ACOMP Channel1 selection

enumerator kACOMP__NegCh2

ACOMP Channel2 selection

enumerator kACOMP__NegCh3

ACOMP Channel3 selection

enumerator kACOMP_NegCh4
ACOMP Channel4 selection

enumerator kACOMP_NegCh5
ACOMP Channel5 selection

enumerator kACOMP_NegCh6
ACOMP Channel6 selection

enumerator kACOMP_NegCh7
ACOMP Channel7 selection

enumerator kACOMP_NegChDACA
DACA selection

enumerator kACOMP_NegChDACB
DACB selection

enumerator kACOMP_NegChVREF1P2
Vref1p2 selection

enumerator kACOMP_NegChAVSS
AVSS selection

enumerator kACOMP_NegChVIO_0P25
VIO Scaling factor 0.25

enumerator kACOMP_NegChVIO_0P50
VIO Scaling factor 0.50

enumerator kACOMP_NegChVIO_0P75
VIO Scaling factor 0.75

enumerator kACOMP_NegChVIO_1P00
VIO Scaling factor 1.00

enum _acomp_input_hysteresis
ACOMP hysteresis level enumeration.

Values:

enumerator kACOMP_Hyster0MV
Hysteresis level = 0mv

enumerator kACOMP_Hyster10MV
Hysteresis level = 10mv

enumerator kACOMP_Hyster20MV
Hysteresis level = 20mv

enumerator kACOMP_Hyster30MV
Hysteresis level = 30mv

enumerator kACOMP_Hyster40MV
Hysteresis level = 40mv

enumerator kACOMP_Hyster50MV
Hysteresis level = 50mv

enumerator kACOMP_Hyster60MV
Hysteresis level = 60mv

enumerator kACOMP_Hyster70MV

Hysteresis level = 70mv

typedef enum *_acomp_result_logic_status* acomp_result_logic_status_t

ACOMP result logical status Type definition.

typedef enum *_acomp_comparator_id* acomp_comparator_id_t

ACOMP comparator id.

typedef enum *_acomp_warm_up_time* acomp_warm_up_time_t

The enumeration of wave up time.

typedef enum *_acomp_response_mode* acomp_response_mode_t

The enumeration of response mode. The response mode will affect the delay from input to output.

typedef enum *_acomp_interrupt_trigger_type* acomp_interrupt_trigger_type_t

ACOMP interrupt trigger type definition.

typedef enum *_acomp_edge_pulse_trig_source* acomp_edge_pulse_trig_source_t

ACOMP edge pulse trigger source type definition.

typedef enum *_acomp_pin_out_type* acomp_pin_out_type_t

ACOMP synchronous/asynchronous output type to pin.

typedef enum *_acomp_positive_channel* acomp_positive_channel_t

ACOMP positive channel enumeration.

typedef enum *_acomp_negative_channel* acomp_negative_channel_t

ACOMP negative channel enumeration.

typedef enum *_acomp_input_hysteresis* acomp_input_hysteresis_t

ACOMP hysteresis level enumeration.

typedef struct *_acomp_positive_input_config* acomp_positive_input_config_t

The configuration of positive input, including channel selection and hysteresis level.

typedef struct *_acomp_negative_input_config* acomp_negative_input_config_t

The configuration of negative input, including channel selection and hysteresis level.

typedef struct *_acomp_config* acomp_config_t

The configure structure of acomp, including warm up time, response mode and so on.

FSL_ACOMP_DRIVER_VERSION

ACOMP driver version.

Version 2.0.1.

ACOMP_REG_ADDR(startAddr, id)

The macro to get the address based on start address and acomp id.

ACOMP_REG_CONST_ADDR(startAddr, id)

ACOMP_GET_REG_VAL(startAddr, id)

The macro to get register value based on start address and acomp id.

ACOMP_GET_REG_CONST_VAL(startAddr, id)

ACOMP_SET_REG_BIT(startAddr, id, val)

Sets register's bit field.

ACOMP_CLEAR_REG_BIT(startAddr, id, val)

Clears register's bit field.

struct *_acomp_positive_input_config*

#include <fsl_acomp.h> The configuration of positive input, including channel selection and hysteresis level.

Public Members

acomp_positive_channel_t channel

Positive input channel selection, please refer to *acomp_positive_channel_t*.

acomp_input_hysteresis_t hysteresisLevel

Positive hysteresis voltage level selection, please refer to *acomp_input_hysteresis_t*.

struct *_acomp_negative_input_config*

#include <fsl_acomp.h> The configuration of negative input, including channel selection and hysteresis level.

Public Members

acomp_negative_channel_t channel

Negative input channel selection, please refer to *acomp_negative_channel_t*.

acomp_input_hysteresis_t hysteresisLevel

Negative hysteresis voltage level selection, please refer to *acomp_input_hysteresis_t*.

struct *_acomp_config*

#include <fsl_acomp.h> The configure structure of *acomp*, including warm up time, response mode and so on.

Public Members

acomp_comparator_id_t id

The id of comparator, please refer to *acomp_comparator_id_t*.

bool enable

Enable/Disable the selected ACOMP.

- **true** Enable the selected ACOMP.
- **false** Disable the selected ACOMP.

acomp_warm_up_time_t warmupTime

Configure warm-up time, please refer to *acomp_warm_up_time_t*.

acomp_response_mode_t responseMode

Configure response mode(power mode), please refer to *acomp_response_mode_t* for details.

acomp_interrupt_trigger_type_t intTrigType

Select interrupt trigger type, please refer to *acomp_interrupt_trigger_type_t*.

acomp_result_logic_status_t inactiveValue

Configure output value for inactive state.

acomp_edge_pulse_trig_source_t edgeDetectTrigSrc

Config edge detect trigger source, please refer to *acomp_edge_pulse_trig_source_t*.

acomp_pin_out_type_t outPinMode

Config the output pin mode, please refer to *acomp_pin_out_type_t* for details.

const *acomp_positive_input_config_t* *posInput

The pointer to the configuration structure of positive input, please refer to *acomp_positive_input_config_t*.

const *acomp_negative_input_config_t* *negInput

The pointer to the configuration structure of negative input, please refer to *acomp_positive_input_config_t*.

2.2 ADC: Analog Digital Converter

void ADC_Init(ADC_Type *base, const *adc_config_t* *config)

Initialize ADC module, including clock divider, power mode, and so on.

Parameters

- base – ADC peripheral base address.
- config – The pointer to the structure *adc_config_t*.

void ADC_GetDefaultConfig(*adc_config_t* *config)

Get default configuration.

```
config->clockDivider = kADC_ClockDivider1;
config->powerMode = kADC_PowerModeFullBiasingCurrent;
config->resolution = kADC_Resolution12Bit;
config->warmupTime = kADC_WarmUpTime16us;
config->vrefSource = kADC_Vref1P2V;
config->inputMode = kADC_InputSingleEnded;
config->conversionMode = kADC_ConversionContinuous;
config->scanLength = kADC_ScanLength_1;
config->averageLength = kADC_AverageNone;
config->triggerSource = kADC_TriggerSourceSoftware;
config->inputGain = kADC_InputGain1;
config->enableInputGainBuffer = false;
config->resultWidth = kADC_ResultWidth16;
config->fifoThreshold = kADC_FifoThresholdData1;
config->enableDMA = false;
config->enableADC = false;
```

Parameters

- config – The Pointer to the structure *adc_config_t*.

void ADC_Deinit(ADC_Type *base)

De-initialize the ADC module.

Parameters

- base – ADC peripheral base address.

static inline void ADC_DoSoftwareReset(ADC_Type *base)

Reset the whole ADC block.

Parameters

- base – ADC peripheral base address.

static inline void ADC_SelectAnalogPortionPowerMode(ADC_Type *base,
 adc_analog_portion_power_mode_t
 powerMode)

Select ADC analog portion power mode.

Parameters

- base – ADC peripheral base address.
- powerMode – The power mode to be set, please refer to `adc_analog_portion_power_mode_t`.

`status_t` ADC_DoAutoCalibration(ADC_Type *base, `adc_calibration_ref_t` calVref)

Do automatic calibration measurement.

Note: After auto calibrate successful, user can invoke `ADC_GetAutoCalibrationData()` to get self offset calibration value and self gain calibration value.

Parameters

- base – ADC peripheral base address.
- calVref – The input reference channel for gain calibration, please refer to `adc_calibration_ref_t` for details.

Return values

- `kStatus_Success` – Auto calibrate successfully.
- `kStatus_Fail` – Auto calibrate failure.

static inline void ADC_GetAutoCalibrationData(ADC_Type *base, uint16_t *offsetCal, uint16_t *gainCal)

Get the ADC automatic calibration data.

Parameters

- base – ADC peripheral base address.
- offsetCal – Self offset calibration data pointer, evaluate NULL if not required.
- gainCal – Self gain calibration data pointer, evaluate NULL if not required.

static inline void ADC_ResetAutoCalibrationData(ADC_Type *base)

Reset the automatic calibration data.

Parameters

- base – ADC peripheral base address.

static inline void ADC_DoUserCalibration(ADC_Type *base, uint16_t offsetCal, uint16_t gainCal)

Do user defined calibration.

Parameters

- base – ADC peripheral base address.
- offsetCal – User defined offset calibration data.
- gainCal – User defined gain calibration data.

static inline void ADC_EnableTemperatureSensor(ADC_Type *base, bool enable)

Enable/disable temperature sensor.

Note: This function is useful only when the channel source is temperature sensor.

Parameters

- base – ADC peripheral base address.

- **enable** – Used to enable/disable temperature sensor.
 - **true** Enable temperature sensor.
 - **false** Disable temperature sensor.

```
static inline void ADC_SetTemperatureSensorMode(ADC_Type *base,  
                                              adc_temperature_sensor_mode_t  
                                              tSensorMode)
```

Set temperature sensor mode, available selections are internal diode mode and external diode mode.

Parameters

- **base** – ADC peripheral base address.
- **tSensorMode** – The temperature sensor mode to be set, please refer to `adc_temperature_sensor_mode_t`.

```
static inline void ADC_EnableAudio(ADC_Type *base, bool enable)  
    Enable/disable audio PGA and decimation rate select.
```

Parameters

- **base** – ADC peripheral base address.
- **enable** – Used to enable/disable audio PGA and decimation rate select.
 - **true** Enable audio PGA and decimation rate select.
 - **false** Disable audio PGA and decimation rate select.

```
static inline void ADC_SetAudioPGAVoltageGain(ADC_Type *base,  
                                              adc_audio_pga_voltage_gain_t voltageGain)
```

Set audio PGA voltage gain.

Parameters

- **base** – ADC peripheral base address.
- **voltageGain** – The selected audio PGA voltage gain, please refer to `adc_audio_pga_voltage_gain_t`.

```
void ADC_ConfigAudioVoiceLevel(ADC_Type *base, bool enableDetect, adc_audio_voice_level_t  
                               voiceLevel)
```

Configure audio voice level.

Parameters

- **base** – ADC peripheral base address.
- **enableDetect** – Used to enable/disable voice level detection.
 - **true** Enable voice level detection.
 - **false** Disable voice level detection.
- **voiceLevel** – Selected voice level, please refer to `adc_audio_voice_level_t`.

```
void ADC_SetScanChannel(ADC_Type *base, adc_scan_channel_t scanChannel,  
                       adc_channel_source_t channelSource)
```

Set scan channel mux source.

Parameters

- **base** – ADC peripheral base address.
- **scanChannel** – The selected channel, please refer to `adc_scan_channel_t` for details.

- `channelSource` – The mux source to be set to the selected channel, please refer to `adc_channel_source_t` for details.

`static inline void ADC_DoSoftwareTrigger(ADC_Type *base)`

If trigger mode is selected as software trigger, invoking this function to start conversion.

Note: This API will also clear the FIFO.

Parameters

- `base` – ADC peripheral base address.

`static inline void ADC_StopConversion(ADC_Type *base)`

Invoke this function to stop conversion.

Parameters

- `base` – ADC peripheral base address.

`static inline uint32_t ADC_GetConversionResult(ADC_Type *base)`

Get the 32-bit width packed ADC conversion result.

Parameters

- `base` – ADC peripheral base address.

Returns

32-bit width packed ADC conversion result.

`static inline uint8_t ADC_GetFifoDataCount(ADC_Type *base)`

Get the ADC FIFO data count.

Parameters

- `base` – ADC peripheral base address.

Returns

ADC FIFO data count.

`static inline void ADC_EnableInterrupts(ADC_Type *base, uint32_t interruptMask)`

Enable interrupts, such as conversion data ready interrupt, gain correction saturation interrupt, FIFO under run interrupt, and so on.

Parameters

- `base` – ADC peripheral base address.
- `interruptMask` – The interrupts to be enabled, should be the OR'ed value of `_adc_interrupt_enable`.

`static inline void ADC_DisableInterrupts(ADC_Type *base, uint32_t interruptMask)`

Disable interrupts, such as conversion data ready interrupt, gain correction saturation interrupt, FIFO under run interrupt, and so on.

Parameters

- `base` – ADC peripheral base address.
- `interruptMask` – The interrupts to be disabled, should be the OR'ed value of `_adc_interrupt_enable`.

`uint32_t ADC_GetStatusFlags(ADC_Type *base)`

Get status flags, including interrupt flags, raw flags, and so on.

Parameters

- `base` – ADC peripheral base address.

Returns

The OR'ed value of ADC status flags, please refer to `_adc_status_flags` for details.

static inline void ADC_ClearStatusFlags(ADC_Type *base, uint32_t statusFlagsMask)
Clear status flags.

Note: Only interrupt flags and raw flags can be cleared.

Parameters

- `base` – ADC peripheral base address.
- `statusFlagsMask` – The OR'ed value of status flags to be cleared, please refer to `_adc_status_flags` for details.

enum `_adc_interrupt_enable`

The enumeration of interrupts, this enumeration can be used to enable/disable interrupts.

Values:

enumerator `kADC_DataReadyInterruptEnable`

Conversion data ready interrupt.

enumerator `kADC_GainSaturationInterruptEnable`

Gain correction saturation interrupt

enumerator `kADC_OffsetSaturationInterruptEnable`

Offset correction saturation interrupt enable.

enumerator `kADC_NegativeSaturationInterruptEnable`

ADC data negative side saturation interrupt enable.

enumerator `kADC_PositiveSaturationInterruptEnable`

ADC data positive side saturation interrupt enable.

enumerator `kADC_FifoOverrunInterruptEnable`

FIFO overrun interrupt enable.

enumerator `kADC_FifoUnderrunInterruptEnable`

FIFO underrun interrupt enable.

enum `_adc_status_flags`

The enumeration of adc status flags, including interrupt flags, raw flags, and so on.

Note: The raw flags will be captured regardless the interrupt mask. Both interrupt flags and raw flags can be cleared.

Values:

enumerator `kADC_DataReadyInterruptFlag`

Conversion Data Ready interrupt flag.

enumerator `kADC_GainSaturationInterruptFlag`

Gain correction saturation interrupt flag.

enumerator `kADC_OffsetSaturationInterruptFlag`

Offset correction saturation interrupt flag.

enumerator kADC_NegativeSaturationInterruptFlag
ADC data negative side saturation interrupt flag.

enumerator kADC_PositiveSaturationInterruptFlag
ADC data positive side saturation interrupt flag.

enumerator kADC_FifoOverrunInterruptFlag
FIFO overrun interrupt flag.

enumerator kADC_FifoUnderrunInterruptFlag
FIFO underrun interrupt flag.

enumerator kADC_DataReadyRawFlag
Conversion data ready raw flag, this flag will be captured regardless the interrupt mask.

enumerator kADC_GainSaturationRawFlag
Gain correction saturation raw flag, this flag will be captured regardless the interrupt mask.

enumerator kADC_OffsetSaturationRawFlag
Offset correction saturation raw flag, this flag will be captured regardless the interrupt mask.

enumerator kADC_NegativeSaturationRawFlag
ADC data negative side saturation raw flag, this flag will be captured regardless the interrupt mask.

enumerator kADC_PositiveSaturationRawFlag
ADC data positive side saturation raw flag, this flag will be captured regardless the interrupt mask.

enumerator kADC_FifoOverrunRawFlag
FIFO overrun raw flag, this flag will be captured regardless the interrupt mask.

enumerator kADC_FifoUnderrunRawFlag
FIFO underrun interrupt mask, this flag will be captured regardless the interrupt mask.

enumerator kADC_ActiveStatusFlag
ADC conversion active status flag.

enumerator kADC_FIFONotEmptyStatusFlag
FIFO not empty status flag.

enumerator kADC_FifoFullStatusFlag
FIFO full status flag.

enum _adc_clock_divider
ADC clock divider ratio type.
Values:

enumerator kADC_ClockDivider1
Clock divider ratio is 1

enumerator kADC_ClockDivider2
Clock divider ratio is 2

enumerator kADC_ClockDivider3
Clock divider ratio is 3

enumerator kADC_ClockDivider4
Clock divider ratio is 4

enumerator kADC_ClockDivider5
Clock divider ratio is 5

enumerator kADC_ClockDivider6
Clock divider ratio is 6

enumerator kADC_ClockDivider7
Clock divider ratio is 7

enumerator kADC_ClockDivider8
Clock divider ratio is 8

enumerator kADC_ClockDivider9
Clock divider ratio is 9

enumerator kADC_ClockDivider10
Clock divider ratio is 10

enumerator kADC_ClockDivider11
Clock divider ratio is 11

enumerator kADC_ClockDivider12
Clock divider ratio is 12

enumerator kADC_ClockDivider13
Clock divider ratio is 13

enumerator kADC_ClockDivider14
Clock divider ratio is 14

enumerator kADC_ClockDivider15
Clock divider ratio is 15

enumerator kADC_ClockDivider16
Clock divider ratio is 16

enumerator kADC_ClockDivider17
Clock divider ratio is 17

enumerator kADC_ClockDivider18
Clock divider ratio is 18

enumerator kADC_ClockDivider19
Clock divider ratio is 19

enumerator kADC_ClockDivider20
Clock divider ratio is 20

enumerator kADC_ClockDivider21
Clock divider ratio is 21

enumerator kADC_ClockDivider22
Clock divider ratio is 22

enumerator kADC_ClockDivider23
Clock divider ratio is 23

enumerator kADC_ClockDivider24
Clock divider ratio is 24

enumerator kADC_ClockDivider25
Clock divider ratio is 25

enumerator kADC_ClockDivider26
Clock divider ratio is 26

enumerator kADC_ClockDivider27
Clock divider ratio is 27

enumerator kADC_ClockDivider28
Clock divider ratio is 28

enumerator kADC_ClockDivider29
Clock divider ratio is 29

enumerator kADC_ClockDivider30
Clock divider ratio is 30

enumerator kADC_ClockDivider31
Clock divider ratio is 31

enumerator kADC_ClockDivider32
Clock divider ratio is 32

enum _adc_analog_portion_power_mode
ADC analog portion low-power mode selection.
Values:

enumerator kADC_PowerModeFullBiasingCurrent
Full biasing current.

enumerator kADC_PowerModeHalfBiasingCurrent
Half biasing current.

enum _adc_resolution
ADC resolution type.
Values:

enumerator kADC_Resolution12Bit
12-bit resolution

enumerator kADC_Resolution14Bit
14-bit resolution

enumerator kADC_Resolution16Bit
16-bit resolution

enumerator kADC_Resolution16BitAudio
16-bit resolution for audio application

enum _adc_warm_up_time
The enumeration of adc warm up time, the ADC warm-up state can also bypassed.
Values:

enumerator kADC_WarmUpTime1us
ADC warm-up time is 1 us.

enumerator kADC_WarmUpTime2us
ADC warm-up time is 2 us.

enumerator kADC_WarmUpTime3us
ADC warm-up time is 3 us.

enumerator kADC_WarmUpTime4us
ADC warm-up time is 4 us.

enumerator kADC_WarmUpTime5us
ADC warm-up time is 5 us.

enumerator kADC_WarmUpTime6us
ADC warm-up time is 6 us.

enumerator kADC_WarmUpTime7us
ADC warm-up time is 7 us.

enumerator kADC_WarmUpTime8us
ADC warm-up time is 8 us.

enumerator kADC_WarmUpTime9us
ADC warm-up time is 9 us.

enumerator kADC_WarmUpTime10us
ADC warm-up time is 10 us.

enumerator kADC_WarmUpTime11us
ADC warm-up time is 11 us.

enumerator kADC_WarmUpTime12us
ADC warm-up time is 12 us.

enumerator kADC_WarmUpTime13us
ADC warm-up time is 13 us.

enumerator kADC_WarmUpTime14us
ADC warm-up time is 14 us.

enumerator kADC_WarmUpTime15us
ADC warm-up time is 15 us.

enumerator kADC_WarmUpTime16us
ADC warm-up time is 16 us.

enumerator kADC_WarmUpTime17us
ADC warm-up time is 17 us.

enumerator kADC_WarmUpTime18us
ADC warm-up time is 18 us.

enumerator kADC_WarmUpTime19us
ADC warm-up time is 19 us.

enumerator kADC_WarmUpTime20us
ADC warm-up time is 20 us.

enumerator kADC_WarmUpTime21us
ADC warm-up time is 21 us.

enumerator kADC_WarmUpTime22us
ADC warm-up time is 22 us.

enumerator kADC_WarmUpTime23us
ADC warm-up time is 23 us.

enumerator kADC_WarmUpTime24us

ADC warm-up time is 24 us.

enumerator kADC_WarmUpTime25us

ADC warm-up time is 25 us.

enumerator kADC_WarmUpTime26us

ADC warm-up time is 26 us.

enumerator kADC_WarmUpTime27us

ADC warm-up time is 27 us.

enumerator kADC_WarmUpTime28us

ADC warm-up time is 28 us.

enumerator kADC_WarmUpTime29us

ADC warm-up time is 29 us.

enumerator kADC_WarmUpTime30us

ADC warm-up time is 30 us.

enumerator kADC_WarmUpTime31us

ADC warm-up time is 31 us.

enumerator kADC_WarmUpTime32us

ADC warm-up time is 32 us.

enumerator kADC_WarmUpStateBypass

ADC warm-up state bypassed.

enum _adc_vref_source

ADC voltage reference source type.

Values:

enumerator kADC_Vref1P8V

Internal 1.8V reference

enumerator kADC_Vref1P2V

Internal 1.2V reference

enumerator kADC_VrefExternal

External single-ended reference though ADC_CH3

enumerator kADC_VrefInternal1P2V

Internal 1.2V reference with cap filter though ADC_CH3

enum _adc_input_mode

ADC input mode type.

Values:

enumerator kADC_InputSingleEnded

Single-ended mode

enumerator kADC_InputDifferential

Differential mode

enum _adc_conversion_mode

ADC conversion mode type.

Values:

enumerator kADC_ConversionOneShot

One shot mode

enumerator kADC_ConversionContinuous

Continuous mode

enum _adc_scan_length

ADC scan length type.

Values:

enumerator kADC_ScanLength_1

Scan length is 1

enumerator kADC_ScanLength_2

Scan length is 2

enumerator kADC_ScanLength_3

Scan length is 3

enumerator kADC_ScanLength_4

Scan length is 4

enumerator kADC_ScanLength_5

Scan length is 5

enumerator kADC_ScanLength_6

Scan length is 6

enumerator kADC_ScanLength_7

Scan length is 7

enumerator kADC_ScanLength_8

Scan length is 8

enumerator kADC_ScanLength_9

Scan length is 9

enumerator kADC_ScanLength_10

Scan length is 10

enumerator kADC_ScanLength_11

Scan length is 11

enumerator kADC_ScanLength_12

Scan length is 12

enumerator kADC_ScanLength_13

Scan length is 13

enumerator kADC_ScanLength_14

Scan length is 14

enumerator kADC_ScanLength_15

Scan length is 15

enumerator kADC_ScanLength_16

Scan length is 16

enum _adc_average_length

ADC average length type.

Values:

enumerator kADC_AverageNone
Average length: no average

enumerator kADC_Average2
Average length: 2

enumerator kADC_Average4
Average length: 4

enumerator kADC_Average8
Average length: 8

enumerator kADC_Average16
Average length: 16

enum _adc_input_gain
ADC input buffer gain type.

Values:

enumerator kADC_InputGain0P5
Input buffer gain is 0.5

enumerator kADC_InputGain1
Input buffer gain is 1

enumerator kADC_InputGain2
Input buffer gain is 2

enum _adc_result_width
ADC result width type.

Values:

enumerator kADC_ResultWidth16
16-bit final result buffer width

enumerator kADC_ResultWidth32
32-bit final result buffer width

enum _adc_fifo_threshold
The threshold of FIFO.

Values:

enumerator kADC_FifoThresholdData1
FIFO Threshold is 1 data.

enumerator kADC_FifoThresholdData4
FIFO Threshold is 4 data.

enumerator kADC_FifoThresholdData8
FIFO Threshold is 8 data.

enumerator kADC_FifoThresholdData16
FIFO Threshold is 16 data.

enum _adc_calibration_ref
ADC calibration voltage reference type.

Values:

enumerator kADC_CalibrationVrefInternal
Internal vref as input for calibration

enumerator kADC_CalibrationVrefExternal

External vref as input for calibration

enum _adc_scan_channel

ADC scan channel type.

Values:

enumerator kADC_ScanChannel0

Scan channel 0

enumerator kADC_ScanChannel1

Scan channel 1

enumerator kADC_ScanChannel2

Scan channel 2

enumerator kADC_ScanChannel3

Scan channel 3

enumerator kADC_ScanChannel4

Scan channel 4

enumerator kADC_ScanChannel5

Scan channel 5

enumerator kADC_ScanChannel6

Scan channel 6

enumerator kADC_ScanChannel7

Scan channel 7

enumerator kADC_ScanChannel8

Scan channel 8

enumerator kADC_ScanChannel9

Scan channel 9

enumerator kADC_ScanChannel10

Scan channel 10

enumerator kADC_ScanChannel11

Scan channel 11

enumerator kADC_ScanChannel12

Scan channel 12

enumerator kADC_ScanChannel13

Scan channel 13

enumerator kADC_ScanChannel14

Scan channel 14

enumerator kADC_ScanChannel15

Scan channel 15

enum _adc_channel_source

ADC channel source type.

Values:

enumerator kADC_CH0

Single-ended mode, channel[0] and vssa

enumerator kADC_CH1

Single-ended mode, channel[1] and vssa

enumerator kADC_CH2

Single-ended mode, channel[2] and vssa

enumerator kADC_CH3

Single-ended mode, channel[3] and vssa

enumerator kADC_CH4

Single-ended mode, channel[4] and vssa

enumerator kADC_CH5

Single-ended mode, channel[5] and vssa

enumerator kADC_CH6

Single-ended mode, channel[6] and vssa

enumerator kADC_CH7

Single-ended mode, channel[7] and vssa

enumerator kADC_VBATS

Single-ended mode, vbat_s and vssa

enumerator kADC_VREF

Single-ended mode, vref_12 and vssa

enumerator kADC_DACA

Single-ended mode, daca and vssa

enumerator kADC_DACB

Single-ended mode, dacb and vssa

enumerator kADC_VSSA

Single-ended mode, vssa and vssa

enumerator kADC_CH0_CH1

Differential mode, channel[0] and channel[1]

enumerator kADC_CH2_CH3

Differential mode, channel[2] and channel[3]

enumerator kADC_CH4_CH5

Differential mode, channel[4] and channel[5]

enumerator kADC_CH6_CH7

Differential mode, channel[6] and channel[7]

enumerator kADC_DACA_DACB

Differential mode, daca and dacb

enum _adc_temperature_sensor_mode

Temperature sensor mode, including internal diode mode and external diode mode.

Values:

enumerator kADC_TSensorExternal

External diode mode.

enum _adc_audio_pga_voltage_gain

ADC audio pga gain type.

Values:

enumerator kADC_AudioGain4

Audio pga gain is 4

enumerator kADC_AudioGain8

Audio pga gain is 8

enumerator kADC_AudioGain16

Audio pga gain is 16

enumerator kADC_AudioGain32

Audio pga gain is 32

enum _adc_audio_voice_level

ADC audio voice level selection.

Values:

enumerator kADC_VoiceLevel0

Input voice level >+255LSB or <-256LSB

enumerator kADC_VoiceLevel1

Input voice level >+511LSB or <-512LSB

enumerator kADC_VoiceLevel2

Input voice level >+1023LSB or <-1024LSB

enumerator kADC_VoiceLevel3

Input voice level >+2047LSB or <-2048LSB

typedef enum _adc_clock_divider adc_clock_divider_t

ADC clock divider ratio type.

typedef enum _adc_analog_portion_power_mode adc_analog_portion_power_mode_t

ADC analog portion low-power mode selection.

typedef enum _adc_resolution adc_resolution_t

ADC resolution type.

typedef enum _adc_warm_up_time adc_warm_up_time_t

The enumeration of adc warm up time, the ADC warm-up state can also bypassed.

typedef enum _adc_vref_source adc_vref_source_t

ADC voltage reference source type.

typedef enum _adc_input_mode adc_input_mode_t

ADC input mode type.

typedef enum _adc_conversion_mode adc_conversion_mode_t

ADC conversion mode type.

typedef enum _adc_scan_length adc_scan_length_t

ADC scan length type.

typedef enum _adc_average_length adc_average_length_t

ADC average length type.

typedef enum _adc_input_gain adc_input_gain_t

ADC input buffer gain type.

typedef enum _adc_result_width adc_result_width_t

ADC result width type.

`typedef enum _adc_fifo_threshold` `adc_fifo_threshold_t`

The threshold of FIFO.

`typedef enum _adc_calibration_ref` `adc_calibration_ref_t`

ADC calibration voltage reference type.

`typedef enum _adc_scan_channel` `adc_scan_channel_t`

ADC scan channel type.

`typedef enum _adc_channel_source` `adc_channel_source_t`

ADC channel source type.

`typedef enum _adc_temperature_sensor_mode` `adc_temperature_sensor_mode_t`

Temperature sensor mode, including internal diode mode and external diode mode.

`typedef enum _adc_audio_pga_voltage_gain` `adc_audio_pga_voltage_gain_t`

ADC audio pga gain type.

`typedef enum _adc_audio_voice_level` `adc_audio_voice_level_t`

ADC audio voice level selection.

`typedef struct _adc_config` `adc_config_t`

The structure of adc options, including clock divider, power mode, and so on.

`FSL_ADC_DRIVER_VERSION`

ADC driver version.

Version 2.2.1.

`struct _adc_config`

`#include <fsl_adc.h>` The structure of adc options, including clock divider, power mode, and so on.

Public Members

`adc_clock_divider_t` `clockDivider`

Analog 64M clock division ratio, please refer to `adc_clock_divider_t`.

`adc_resolution_t` `resolution`

Configure ADC resolution, please refer to `adc_resolution_t`.

`adc_warm_up_time_t` `warmupTime`

Configure warm-up time.

`adc_vref_source_t` `vrefSource`

Configure voltage reference source, please refer to `adc_vref_source_t`.

`adc_input_mode_t` `inputMode`

Configure input mode, such as `kADC_InputSingleEnded` or `kADC_InputDifferential`.

`adc_conversion_mode_t` `conversionMode`

Configure convrsion mode, such as `kADC_ConversionOneShot` or `kADC_ConversionContinuous`.

`adc_scan_length_t` `scanLength`

Configure the length of scan, please refer to `adc_scan_length_t`.

`adc_average_length_t` `averageLength`

Configure hardware average number, please refer to `adc_average_length_t`

`adc_trigger_source_t` triggerSource

Configure trigger source, the trigger source can be divided into hardware trigger and software trigger, please refer to `adc_trigger_source_t` for details.

`adc_input_gain_t` inputGain

Configure ADC input buffer gain, please refer to `adc_input_gain_t`.

`bool` enableInputGainBuffer

Enable/Disable input gain buffer.

- **true** Enable input gain buffer.
- **false** Disable input gain buffer.

`bool` enableInputBufferChop

Enable/Disable input buffer chopper:

- **true** Enable input buffer chopper;
- **false** Disable input buffer chopper.

`bool` enableChop

Enable/Disable the ADC chopper:

- **true** Enable the chopper;
- **false** Disable the chopper.

`adc_result_width_t` resultWidth

Select result FIFO data packed format, please refer to `adc_result_width_t`.

`adc_fifo_threshold_t` fifoThreshold

Configure FIFO threshold, please refer to `adc_fifo_threshold_t`.

`bool` enableDMA

Enable/Disable DMA request.

- **true** Enable DMA request.
- **false** Disable DMA request.

`bool` enableADC

Enable/Disable ADC module.

- **true** Enable ADC module.
- **false** Disable ADC module.

2.3 CACHE: CACHE Memory Controller

`uint32_t` CACHE64_GetInstance(CACHE64_POLSEL_Type *base)

Returns an instance number given peripheral base address.

Parameters

- base – The peripheral base address.

Returns

CACHE64_POLSEL instance number starting from 0.

`uint32_t` CACHE64_GetInstanceByAddr(uint32_t address)

brief Returns an instance number given physical memory address.

param address The physical memory address.

Returns

CACHE64_CTRL instance number starting from 0.

status_t CACHE64_Init(CACHE64_POLSEL_Type *base, const *cache64_config_t* *config)

Initializes an CACHE64 instance with the user configuration structure.

This function configures the CACHE64 module with user-defined settings. Call the CACHE64_GetDefaultConfig() function to configure the configuration structure and get the default configuration.

Parameters

- base – CACHE64_POLSEL peripheral base address.
- config – Pointer to a user-defined configuration structure.

Return values

kStatus_Success – CACHE64 initialize succeed

void CACHE64_GetDefaultConfig(*cache64_config_t* *config)

Gets the default configuration structure.

This function initializes the CACHE64 configuration structure to a default value. The default values are first region covers whole cacheable area, and policy set to write back.

Parameters

- config – Pointer to a configuration structure.

void CACHE64_EnableCache(CACHE64_CTRL_Type *base)

Enables the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_DisableCache(CACHE64_CTRL_Type *base)

Disables the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_InvalidateCache(CACHE64_CTRL_Type *base)

Invalidates the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

void CACHE64_InvalidateCacheByRange(uint32_t address, uint32_t size_byte)

Invalidates cache by range.

Note: Address and size should be aligned to "CACHE64_LINESIZE_BYTE". The startAddr here will be forced to align to CACHE64_LINESIZE_BYTE if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address of cache.
- size_byte – size of the memory to be invalidated, should be larger than 0.

```
void CACHE64_CleanCache(CACHE64_CTRL_Type *base)
```

Cleans the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

```
void CACHE64_CleanCacheByRange(uint32_t address, uint32_t size_byte)
```

Cleans cache by range.

Note: Address and size should be aligned to “CACHE64_LINESIZE_BYTE”. The startAddr here will be forced to align to CACHE64_LINESIZE_BYTE if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address of cache.
- size_byte – size of the memory to be cleaned, should be larger than 0.

```
void CACHE64_CleanInvalidateCache(CACHE64_CTRL_Type *base)
```

Cleans and invalidates the cache.

Parameters

- base – CACHE64_CTRL peripheral base address.

```
void CACHE64_CleanInvalidateCacheByRange(uint32_t address, uint32_t size_byte)
```

Cleans and invalidate cache by range.

Note: Address and size should be aligned to “CACHE64_LINESIZE_BYTE”. The startAddr here will be forced to align to CACHE64_LINESIZE_BYTE if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address of cache.
- size_byte – size of the memory to be Cleaned and Invalidated, should be larger than 0.

```
void CACHE64_EnableWriteBuffer(CACHE64_CTRL_Type *base, bool enable)
```

Enables/disables the write buffer.

Parameters

- base – CACHE64_CTRL peripheral base address.
- enable – The enable or disable flag. true - enable the write buffer. false - disable the write buffer.

```
static inline void ICACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)
```

Invalidates instruction cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application

should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address.
- size_byte – size of the memory to be invalidated, should be larger than 0.

static inline void DCACHE_InvalidateByRange(uint32_t address, uint32_t size_byte)

Invalidates data cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address.
- size_byte – size of the memory to be invalidated, should be larger than 0.

static inline void DCACHE_CleanByRange(uint32_t address, uint32_t size_byte)

Clean data cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address.
- size_byte – size of the memory to be cleaned, should be larger than 0.

static inline void DCACHE_CleanInvalidateByRange(uint32_t address, uint32_t size_byte)

Cleans and Invalidates data cache by range.

Note: Address and size should be aligned to CACHE64_LINESIZE_BYTE due to the cache operation unit FSL_FEATURE_CACHE64_CTRL_LINESIZE_BYTE. The startAddr here will be forced to align to the cache line size if startAddr is not aligned. For the size_byte, application should make sure the alignment or make sure the right operation order if the size_byte is not aligned.

Parameters

- address – The physical address.
- size_byte – size of the memory to be Cleaned and Invalidated, should be larger than 0.

FSL_CACHE_DRIVER_VERSION

cache driver version.

`enum _cache64_policy`
Level 2 cache controller way size.
Values:
`enumerator kCACHE64_PolicyNonCacheable`
Non-cacheable
`enumerator kCACHE64_PolicyWriteThrough`
Write through
`enumerator kCACHE64_PolicyWriteBack`
Write back
`typedef enum _cache64_policy cache64_policy_t`
Level 2 cache controller way size.
`typedef struct _cache64_config cache64_config_t`
CACHE64 configuration structure.
`CACHE64_LINESIZE_BYTE`
cache line size.
`CACHE64_REGION_NUM`
cache region number.
`CACHE64_REGION_ALIGNMENT`
cache region alignment.
`struct _cache64_config`
#include <fsl_cache.h> CACHE64 configuration structure.

Public Members

`uint32_t boundaryAddr[(3U) - 1]`
< The cache controller can divide whole memory into 3 regions. Boundary address is the FlexSPI internal address (start from 0) instead of system address (start from FlexSPI AMBA base) to split adjacent regions and must be 1KB aligned. The boundary address itself locates in upper region. Cacheable policy for each region.

2.4 CDOG

`status_t CDOG_Init(CDOG_Type *base, cdog_config_t *conf)`
Initialize CDOG.

This function initializes CDOG block and setting.

Parameters

- `base` – CDOG peripheral base address
- `conf` – CDOG configuration structure

Returns

Status of the init operation

`void CDOG_Deinit(CDOG_Type *base)`
Deinitialize CDOG.

This function deinitializes CDOG secure counter.

Parameters

- base – CDOG peripheral base address

void CDOG_GetDefaultConfig(*cdog_config_t* *conf)

Sets the default configuration of CDOG.

This function initialize CDOG config structure to default values.

Parameters

- conf – CDOG configuration structure

void CDOG_Stop(CDOG_Type *base, uint32_t stop)

Stops secure counter and instruction timer.

This function stops instruction timer and secure counter. This also change state of CDOG to IDLE.

Parameters

- base – CDOG peripheral base address
- stop – expected value which will be compared with value of secure counter

void CDOG_Start(CDOG_Type *base, uint32_t reload, uint32_t start)

Sets secure counter and instruction timer values.

This function sets value in RELOAD and START registers for instruction timer and secure counter

Parameters

- base – CDOG peripheral base address
- reload – reload value
- start – start value

void CDOG_Check(CDOG_Type *base, uint32_t check)

Checks secure counter.

This function compares stop value in handler with secure counter value by writing to RELOAD register.

Parameters

- base – CDOG peripheral base address
- check – expected (stop) value

void CDOG_Set(CDOG_Type *base, uint32_t stop, uint32_t reload, uint32_t start)

Sets secure counter and instruction timer values.

This function sets value in STOP, RELOAD and START registers for instruction timer and secure counter.

Parameters

- base – CDOG peripheral base address
- stop – expected value which will be compared with value of secure counter
- reload – reload value for instruction timer
- start – start value for secure timer

void CDOG_Add(CDOG_Type *base, uint32_t add)

Add value to secure counter.

This function add specified value to secure counter.

Parameters

- base – CDOG peripheral base address.
- add – Value to be added.

void CDOG__Add1(CDOG_Type *base)

Add 1 to secure counter.

This function add 1 to secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG__Add16(CDOG_Type *base)

Add 16 to secure counter.

This function add 16 to secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG__Add256(CDOG_Type *base)

Add 256 to secure counter.

This function add 256 to secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG__Sub(CDOG_Type *base, uint32_t sub)

brief Subtract value to secure counter

This function subtract specified value to secure counter.

param base CDOG peripheral base address. param sub Value to be subtracted.

void CDOG__Sub1(CDOG_Type *base)

Subtract 1 from secure counter.

This function subtract specified 1 from secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG__Sub16(CDOG_Type *base)

Subtract 16 from secure counter.

This function subtract specified 16 from secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG__Sub256(CDOG_Type *base)

Subtract 256 from secure counter.

This function subtract specified 256 from secure counter.

Parameters

- base – CDOG peripheral base address.

void CDOG__WritePersistent(CDOG_Type *base, uint32_t value)

Set the CDOG persistent word.

Parameters

- base – CDOG peripheral base address.
- value – The value to be written.

uint32_t CDOG_ReadPersistent(CDOG_Type *base)

Get the CDOG persistent word.

Parameters

- base – CDOG peripheral base address.

Returns

The persistent word.

FSL_CDOG_DRIVER_VERSION

Defines CDOG driver version 2.1.3.

Change log:

- Version 2.1.3
 - Re-design multiple instance IRQs and Clocks
 - Add fix for RESTART command errata
- Version 2.1.2
 - Support multiple IRQs
 - Fix default CONTROL values
- Version 2.1.1
 - Remove bit CONTROL[CONTROL_CTRL]
- Version 2.1.0
 - Rename CWT to CDOG
- Version 2.0.2
 - Fix MISRA-2012 issues
- Version 2.0.1
 - Fix doxygen issues
- Version 2.0.0
 - initial version

enum __cdog_debug_Action_ctrl_enum

Values:

enumerator kCDOG_DebugHaltCtrl_Run

enumerator kCDOG_DebugHaltCtrl_Pause

enum __cdog_irq_pause_ctrl_enum

Values:

enumerator kCDOG_IrqPauseCtrl_Run

enumerator kCDOG_IrqPauseCtrl_Pause

enum __cdog_fault_ctrl_enum

Values:

enumerator kCDOG_FaultCtrl_EnableReset

enumerator kCDOG_FaultCtrl_EnableInterrupt

```
    enumerator kCDOG_FaultCtrl_NoAction
enum __code_lock_ctrl_enum
    Values:
    enumerator kCDOG_LockCtrl_Lock
    enumerator kCDOG_LockCtrl_Unlock
typedef uint32_t secure_counter_t
SC_ADD(add)
SC_ADD1
SC_ADD16
SC_ADD256
SC_SUB(sub)
SC_SUB1
SC_SUB16
SC_SUB256
SC_CHECK(val)
struct cdog_config_t
    #include <fsl_cdog.h>
```

2.5 Clock Driver

```
enum _clock_name
    Clock name used to get clock frequency.
    Values:
    enumerator kCLOCK_CoreSysClk
        Core clock (aka HCLK)
    enumerator kCLOCK_BusClk
        Bus clock (AHB/APB clock, aka HCLK)
    enumerator kCLOCK_MclkClk
        MCLK, to MCLK pin
enum _clock_ip_name
    Peripheral clock name difinition used for clock gate.
    Values:
    enumerator kCLOCK_IpInvalid
    enumerator kCLOCK_TcpuMciClk
    enumerator kCLOCK_TcpuMciFlexspiClk
    enumerator kCLOCK_TddrMciEnetClk
    enumerator kCLOCK_TddrMciFlexspiClk
```

enumerator kCLOCK__T3PllMciIrcClk
enumerator kCLOCK__T3PllMci256mClk
enumerator kCLOCK__T3PllMci213mClk
enumerator kCLOCK__T3PllMciFlexspiClk
enumerator kCLOCK__RefClkSys
enumerator kCLOCK__RefClkTepu
enumerator kCLOCK__RefClkTddr
enumerator kCLOCK__RefClkAud
enumerator kCLOCK__RefClkUsb
enumerator kCLOCK__RefClkCauSlp
enumerator kCLOCK__Cpu
enumerator kCLOCK__Matrix
enumerator kCLOCK__Romcp
enumerator kCLOCK__PowerQuad
enumerator kCLOCK__Pkc
enumerator kCLOCK__Els
enumerator kCLOCK__Puf
enumerator kCLOCK__Flexspi
enumerator kCLOCK__Hpu
enumerator kCLOCK__Usb
enumerator kCLOCK__Sct
enumerator kCLOCK__AonMem
enumerator kCLOCK__Gdma
enumerator kCLOCK__Dma0
enumerator kCLOCK__Dma1
enumerator kCLOCK__Sdio
enumerator kCLOCK__ElsApb
enumerator kCLOCK__SdioSlv
enumerator kCLOCK__Gau
enumerator kCLOCK__Otp
enumerator kCLOCK__SecureGpio
enumerator kCLOCK__EnetIpg
enumerator kCLOCK__EnetIpgS

enumerator kCLOCK_Trng
enumerator kCLOCK_Utick
enumerator kCLOCK_Wwdt0
enumerator kCLOCK_Usim
enumerator kCLOCK_Itrc
enumerator kCLOCK_FreeMrt
enumerator kCLOCK_Lcdic
enumerator kCLOCK_Flexcomm0
enumerator kCLOCK_Flexcomm1
enumerator kCLOCK_Flexcomm2
enumerator kCLOCK_Flexcomm3
enumerator kCLOCK_Flexcomm14
enumerator kCLOCK_Dmic0
enumerator kCLOCK_OsEventTimer
enumerator kCLOCK_HsGpio0
enumerator kCLOCK_HsGpio1
enumerator kCLOCK_Crc
enumerator kCLOCK_Freqme
enumerator kCLOCK_Ct32b0
enumerator kCLOCK_Ct32b1
enumerator kCLOCK_Ct32b2
enumerator kCLOCK_Ct32b3
enumerator kCLOCK_Ct32b4
enumerator kCLOCK_Pmu
enumerator kCLOCK_Rtc
enumerator kCLOCK_Mrt
enumerator kCLOCK_Pint
enumerator kCLOCK_InputMux

enum _clock_attach_id

Peripheral clock source selection definition.

Values:

enumerator kXTAL_to_SYSOSC_CLK
enumerator kCLKIN_to_SYSOSC_CLK
enumerator kNONE_to_SYSOSC_CLK

enumerator kSYSOSC_to_MAIN_CLK
enumerator kFFRO_DIV4_to_MAIN_CLK
enumerator kLPOSC_to_MAIN_CLK
enumerator kFFRO_to_MAIN_CLK
enumerator kSFRO_to_MAIN_CLK
enumerator kMAIN_PLL_to_MAIN_CLK
enumerator kCLK32K_to_MAIN_CLK
enumerator kMAIN_CLK_to_FLEXSPI_CLK
enumerator kT3PLL_MCI_FLEXSPI_to_FLEXSPI_CLK
enumerator kAUX0_PLL_to_FLEXSPI_CLK
enumerator kTCPU_MCI_FLEXSPI_to_FLEXSPI_CLK
enumerator kAUX1_PLL_to_FLEXSPI_CLK
enumerator kTDDR_MCI_FLEXSPI_to_FLEXSPI_CLK
enumerator kT3PLL_MCI_256M_to_FLEXSPI_CLK
enumerator kNONE_to_FLEXSPI_CLK
enumerator kMAIN_CLK_to_SCT_CLK
enumerator kMAIN_PLL_to_SCT_CLK
enumerator kAUX0_PLL_to_SCT_CLK
enumerator kFFRO_to_SCT_CLK
enumerator kAUX1_PLL_to_SCT_CLK
enumerator kAUDIO_PLL_to_SCT_CLK
enumerator kNONE_to_SCT_CLK
enumerator kLPOSC_to_UTICK_CLK
enumerator kMAIN_CLK_to_UTICK_CLK
enumerator kNONE_to_UTICK_CLK
enumerator kLPOSC_to_WDT0_CLK
enumerator kMAIN_CLK_to_WDT0_CLK
enumerator kNONE_to_WDT0_CLK
enumerator kSYSTICK_DIV_to_SYSTICK_CLK
enumerator kLPOSC_to_SYSTICK_CLK
enumerator kCLK32K_to_SYSTICK_CLK
enumerator kSFRO_to_SYSTICK_CLK
enumerator kNONE_to_SYSTICK_CLK

enumerator kMAIN_CLK_to_USIM_CLK
enumerator kAUDIO_PLL_to_USIM_CLK
enumerator kFFRO_to_USIM_CLK
enumerator kNONE_to_USIM_CLK
enumerator kMAIN_CLK_to_LCD_CLK
enumerator kT3PLL_MCI_FLEXSPI_to_LCD_CLK
enumerator kTCPU_MCI_FLEXSPI_to_LCD_CLK
enumerator kTDDR_MCI_FLEXSPI_to_LCD_CLK
enumerator kNONE_to_LCD_CLK
enumerator kMAIN_CLK_to_GAU_CLK
enumerator kT3PLL_MCI_256M_to_GAU_CLK
enumerator kAVPLL_CH2_to_GAU_CLK
enumerator kNONE_to_GAU_CLK
enumerator kT3PLL_MCI_256M_to_ELS_GDET
enumerator kELS_128M_to_ELS_GDET
enumerator kELS_64M_to_ELS_GDET
enumerator kOTP_FUSE_32M_to_ELS_GDET
enumerator kNONE_to_ELS_GDET
enumerator kLPOSC_to_OSTIMER_CLK
enumerator kCLK32K_to_OSTIMER_CLK
enumerator kHCLK_to_OSTIMER_CLK
enumerator kMAIN_CLK_to_OSTIMER_CLK
enumerator kNONE_to_OSTIMER_CLK
enumerator kSFRO_to_FLEXCOMM0
enumerator kFFRO_to_FLEXCOMM0
enumerator kAUDIO_PLL_to_FLEXCOMM0
enumerator kMCLK_IN_to_FLEXCOMM0
enumerator kFRG_to_FLEXCOMM0
enumerator kNONE_to_FLEXCOMM0
enumerator kSFRO_to_FLEXCOMM1
enumerator kFFRO_to_FLEXCOMM1
enumerator kAUDIO_PLL_to_FLEXCOMM1
enumerator kMCLK_IN_to_FLEXCOMM1

enumerator kFRG_to_FLEXCOMM1
enumerator kNONE_to_FLEXCOMM1
enumerator kSFRO_to_FLEXCOMM2
enumerator kFFRO_to_FLEXCOMM2
enumerator kAUDIO_PLL_to_FLEXCOMM2
enumerator kMCLK_IN_to_FLEXCOMM2
enumerator kFRG_to_FLEXCOMM2
enumerator kNONE_to_FLEXCOMM2
enumerator kSFRO_to_FLEXCOMM3
enumerator kFFRO_to_FLEXCOMM3
enumerator kAUDIO_PLL_to_FLEXCOMM3
enumerator kMCLK_IN_to_FLEXCOMM3
enumerator kFRG_to_FLEXCOMM3
enumerator kNONE_to_FLEXCOMM3
enumerator kSFRO_to_FLEXCOMM14
enumerator kFFRO_to_FLEXCOMM14
enumerator kAUDIO_PLL_to_FLEXCOMM14
enumerator kMCLK_IN_to_FLEXCOMM14
enumerator kFRG_to_FLEXCOMM14
enumerator kNONE_to_FLEXCOMM14
enumerator kSFRO_to_DMIC_CLK
enumerator kFFRO_to_DMIC_CLK
enumerator kAUDIO_PLL_to_DMIC_CLK
enumerator kMCLK_IN_to_DMIC_CLK
enumerator kLPOSC_to_DMIC_CLK
enumerator kCLK32K_to_DMIC_CLK
enumerator kMAIN_CLK_to_DMIC_CLK
enumerator kNONE_to_DMIC_CLK
enumerator kMAIN_CLK_to_CTIMER0
enumerator kSFRO_to_CTIMER0
enumerator kFFRO_to_CTIMER0
enumerator kAUDIO_PLL_to_CTIMER0
enumerator kMCLK_IN_to_CTIMER0

enumerator kLPOSC_to_CTIMER0
enumerator kNONE_to_CTIMER0
enumerator kMAIN_CLK_to_CTIMER1
enumerator kSFRO_to_CTIMER1
enumerator kFFRO_to_CTIMER1
enumerator kAUDIO_PLL_to_CTIMER1
enumerator kMCLK_IN_to_CTIMER1
enumerator kLPOSC_to_CTIMER1
enumerator kNONE_to_CTIMER1
enumerator kMAIN_CLK_to_CTIMER2
enumerator kSFRO_to_CTIMER2
enumerator kFFRO_to_CTIMER2
enumerator kAUDIO_PLL_to_CTIMER2
enumerator kMCLK_IN_to_CTIMER2
enumerator kLPOSC_to_CTIMER2
enumerator kNONE_to_CTIMER2
enumerator kMAIN_CLK_to_CTIMER3
enumerator kSFRO_to_CTIMER3
enumerator kFFRO_to_CTIMER3
enumerator kAUDIO_PLL_to_CTIMER3
enumerator kMCLK_IN_to_CTIMER3
enumerator kLPOSC_to_CTIMER3
enumerator kNONE_to_CTIMER3
enumerator kFFRO_to_MCLK_CLK
enumerator kAUDIO_PLL_to_MCLK_CLK
enumerator kMAIN_CLK_to_MCLK_CLK
enumerator kNONE_to_MCLK_CLK
enumerator kSFRO_to_CLKOUT
enumerator kSYSOSC_to_CLKOUT
enumerator kLPOSC_to_CLKOUT
enumerator kFFRO_to_CLKOUT
enumerator kMAIN_CLK_to_CLKOUT
enumerator kREFCLK_SYS_to_CLKOUT

enumerator kAVPLL_CH2_to_CLKOUT
enumerator kMAIN_PLL_to_CLKOUT
enumerator kAUX0_PLL_to_CLKOUT
enumerator kAUX1_PLL_to_CLKOUT
enumerator kAUDIO_PLL_to_CLKOUT
enumerator kCLK32K_to_CLKOUT
enumerator kTCPU_MCI_FLEXSPI_to_CLKOUT
enumerator kTDDR_MCI_FLEXSPI_to_CLKOUT
enumerator kT3PLL_MCI_FLEXSPI_to_CLKOUT
enumerator kT3PLL_MCI_256M_to_CLKOUT
enumerator kCAU_SLP_REF_CLK_to_CLKOUT
enumerator kTDDR_MCI_ENET_to_CLKOUT
enumerator kNONE_to_CLKOUT
enumerator kRC32K_to_CLK32K
enumerator kXTAL32K_to_CLK32K
enumerator kNCO32K_to_CLK32K

enum _clock_div_name

Clock divider definition.

Values:

enumerator kCLOCK_DivMainPllClk
enumerator kCLOCK_DivAux0PllClk
enumerator kCLOCK_DivAux1PllClk
enumerator kCLOCK_DivSysCpuAhbClk
enumerator kCLOCK_DivPfc1Clk
enumerator kCLOCK_DivFlexspiClk
enumerator kCLOCK_DivSctClk
enumerator kCLOCK_DivUsbHsFclk
enumerator kCLOCK_DivSystickClk
enumerator kCLOCK_DivLcdClk
enumerator kCLOCK_DivGauClk
enumerator kCLOCK_DivUsimClk
enumerator kCLOCK_DivPmuFclk
enumerator kCLOCK_DivAudioPllClk
enumerator kCLOCK_DivPllFrgClk

enumerator kCLOCK_DivDmicClk

enumerator kCLOCK_DivMclClk

enumerator kCLOCK_DivClockOut

enum clock_tcpu_flexspi_div_t

TCPU PLL divider for tcpu_mci_flexspi_clk.

Values:

enumerator kCLOCK_TcpuFlexspiDiv12

Divided by 12

enumerator kCLOCK_TcpuFlexspiDiv11

Divided by 11

enumerator kCLOCK_TcpuFlexspiDiv10

Divided by 10

enumerator kCLOCK_TcpuFlexspiDiv9

Divided by 9

enum clock_tddr_flexspi_div_t

TDDR PLL divider for tddr_mci_flexspi_clk.

Values:

enumerator kCLOCK_TddrFlexspiDiv11

Divided by 11

enumerator kCLOCK_TddrFlexspiDiv10

Divided by 10

enumerator kCLOCK_TddrFlexspiDiv9

Divided by 9

enumerator kCLOCK_TddrFlexspiDiv8

Divided by 8

enum clock_t3_mci_irc_config_t

T3 PLL IRC configuration.

Values:

enumerator kCLOCK_T3MciIrc60m

T3 MCI IRC 59.53MHz

enumerator kCLOCK_T3MciIrc48m

T3 MCI IRC 48.30MHz

enum clock_avpll_ch_freq_t

AVPLL channel1 frequency configuration.

Values:

enumerator kCLOCK_AvPllChUnchanged

AVPLL channel frequency unchanged.

enumerator kCLOCK_AvPllChFreq2p048m

AVPLL channel frequency 2.048MHz

enumerator kCLOCK_AvPllChFreq4p096m

AVPLL channel frequency 4.096MHz

```

enumerator kCLOCK_AvPllChFreq6p144m
    AVPLL channel frequency 6.144MHz
enumerator kCLOCK_AvPllChFreq8p192m
    AVPLL channel frequency 8.192MHz
enumerator kCLOCK_AvPllChFreq11p2896m
    AVPLL channel frequency 11.2896MHz
enumerator kCLOCK_AvPllChFreq12m
    AVPLL channel frequency 12MHz
enumerator kCLOCK_AvPllChFreq12p288m
    AVPLL channel frequency 12.288MHz
enumerator kCLOCK_AvPllChFreq24p576m
    AVPLL channel frequency 24.576MHz
enumerator kCLOCK_AvPllChFreq64m
    AVPLL channel frequency 64MHz
enumerator kCLOCK_AvPllChFreq98p304m
    AVPLL channel frequency 98.304MHz

```

```

typedef enum _clock_name clock_name_t
    Clock name used to get clock frequency.

```

```

typedef enum _clock_ip_name clock_ip_name_t
    Peripheral clock name difinition used for clock gate.

```

```

typedef enum _clock_attach_id clock_attach_id_t
    Peripheral clock source selection definition.

```

```

typedef enum _clock_div_name clock_div_name_t
    Clock divider definition.

```

```

typedef struct _clock_frg_clk_config clock_frg_clk_config_t
    PLL configuration for FRG.

```

```

volatile uint32_t g_clkinFreq
    External CLK_IN pin clock frequency (clkin) clock frequency.

```

The CLK_IN pin (clkin) clock frequency in Hz, when the clock is setup, use the function `CLOCK_SetClkinFreq` to set the value in to clock driver. For example, if CLK_IN is 16MHz,

```
CLOCK_SetClkinFreq(16000000);
```

```

volatile uint32_t g_mclkinFreq
    External MCLK IN clock frequency.

```

The MCLK in (mclk_in) PIN clock frequency in Hz, when the clock is setup, use the function `CLOCK_SetMclkInFreq` to set the value in to clock driver. For example, if mclk_In is 16MHz,

```
CLOCK_SetMclkInFreq(16000000);
```

```

uint32_t CLOCK_GetT3PllMciIrcClkFreq(void)
    Return Frequency of t3pll_mci_48_60m_irc.

```

Returns

Frequency of t3pll_mci_48_60m_irc

uint32_t CLOCK_GetT3PllMci213mClkFreq(void)

Return Frequency of t3pll_mci_213p3m.

Returns

Frequency of t3pll_mci_213p3m

uint32_t CLOCK_GetT3PllMci256mClkFreq(void)

Return Frequency of t3pll_mci_256m.

Returns

Frequency of t3pll_mci_256m

uint32_t CLOCK_GetT3PllMciFlexspiClkFreq(void)

Return Frequency of t3pll_mci_flexspi_clk.

Returns

Frequency of t3pll_mci_flexspi_clk

uint32_t CLOCK_GetTcpuMciClkFreq(void)

Return Frequency of tcpu_mci_clk.

Returns

Frequency of tcpu_mci_clk

uint32_t CLOCK_GetTcpuMciFlexspiClkFreq(void)

Return Frequency of tcpu_mci_flexspi_clk.

Returns

Frequency of tcpu_mci_flexspi_clk

uint32_t CLOCK_GetTddrMciFlexspiClkFreq(void)

Return Frequency of tddr_mci_flexspi_clk.

Returns

Frequency of tddr_mci_flexspi_clk

uint32_t CLOCK_GetTddrMciEnetClkFreq(void)

Return Frequency of tddr_mci_enet_clk.

Returns

Frequency of tddr_mci_enet_clk

void CLOCK_EnableClock(*clock_ip_name_t* clk)

Enable the clock for specific IP.

Parameters

- clk – Which clock to enable, see clock_ip_name_t.

void CLOCK_DisableClock(*clock_ip_name_t* clk)

Disable the clock for specific IP.

Parameters

- clk – Which clock to disable, see clock_ip_name_t.

void CLOCK_AttachClk(*clock_attach_id_t* connection)

Configure the clock selection muxes.

Parameters

- connection – : Clock to be configured.

void CLOCK_SetClkDiv(*clock_div_name_t* name, uint32_t divider)

Setup clock dividers.

Parameters

- name – : Clock divider name
- divider – : Value to be divided.

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Return Frequency of selected clock.

Returns

Frequency of selected clock

uint32_t CLOCK_GetFRGClock(uint32_t id)

Return Input frequency for the Fractional baud rate generator.

Returns

Input Frequency for FRG

void CLOCK_SetFRGClock(const *clock_frg_clk_config_t* *config)

Set output of the Fractional baud rate generator.

Parameters

- config – : Configuration to set to FRGn clock.

uint32_t CLOCK_GetFFroFreq(void)

Return Frequency of FFRO.

Returns

Frequency of FFRO

uint32_t CLOCK_GetSFroFreq(void)

Return Frequency of SFRO.

Returns

Frequency of SFRO

uint32_t CLOCK_GetAvPllCh1Freq(void)

Return Frequency of AUDIO PLL (AVPLL CH1)

Returns

Frequency of AUDIO PLL

uint32_t CLOCK_GetAvPllCh2Freq(void)

Return Frequency of AVPLL CH2.

Returns

Frequency of AVPLL CH2

uint32_t CLOCK_GetMainClkFreq(void)

Return Frequency of main clk.

Returns

Frequency of main clk

uint32_t CLOCK_GetCoreSysClkFreq(void)

Return Frequency of core/bus clk.

Returns

Frequency of core/bus clk

uint32_t CLOCK_GetSystickClkFreq(void)

Return Frequency of systick clk.

Returns

Frequency of systick clk

static inline uint32_t CLOCK_GetSysOscFreq(void)

Return Frequency of sys osc Clock.

Returns

Frequency of sys osc Clock. Or CLK_IN pin frequency.

static inline uint32_t CLOCK_GetMclkInClkFreq(void)

Return Frequency of MCLK Input Clock.

Returns

Frequency of MCLK input Clock.

static inline uint32_t CLOCK_GetLpOscFreq(void)

Return Frequency of LPOSC.

Returns

Frequency of LPOSC

static inline uint32_t CLOCK_GetClk32KFreq(void)

Return Frequency of CLK_32K.

Returns

Frequency of 32KHz osc

void CLOCK_EnableXtal32K(bool enable)

Enables and disables 32KHz XTAL.

Parameters

- enable – : true to enable 32k XTAL clock, false to disable clock

void CLOCK_EnableRtc32K(bool enable)

Enables and disables RTC 32KHz.

Parameters

- enable – : true to enable 32k RTC clock, false to disable clock

static inline void CLOCK_SetClkinFreq(uint32_t freq)

Set the CLKIN (CLKIN pin) frequency based on GPIO4 input.

Parameters

- freq – : The CLK_IN pin input clock frequency in Hz.

static inline void CLOCK_SetMclkInFreq(uint32_t freq)

Set the MCLK in (mclk_in) clock frequency based on board setting.

Parameters

- freq – : The MCLK input clock frequency in Hz.

uint32_t CLOCK_GetDmicClkFreq(void)

Return Frequency of DMIC clk.

Returns

Frequency of DMIC clk

uint32_t CLOCK_GetLcdClkFreq(void)

Return Frequency of LCD clk.

Returns

Frequency of LCD clk

uint32_t CLOCK_GetWdtClkFreq(void)

Return Frequency of WDT clk.

Returns

Frequency of WDT clk

uint32_t CLOCK_GetMclkClkFreq(void)

Return Frequency of mclk.

Returns

Frequency of mclk clk

uint32_t CLOCK_GetSctClkFreq(void)

Return Frequency of sct.

Returns

Frequency of sct clk

uint32_t CLOCK_GetFlexCommClkFreq(uint32_t id)

Return Frequency of Flexcomm functional Clock.

Parameters

- id – : flexcomm index to get frequency.

Returns

Frequency of Flexcomm functional Clock

uint32_t CLOCK_GetCTimerClkFreq(uint32_t id)

Return Frequency of CTimer Clock.

Parameters

- id – : ctimer index to get frequency.

Returns

Frequency of CTimer Clock

uint32_t CLOCK_GetUtickClkFreq(void)

Return Frequency of Utick Clock.

Returns

Frequency of Utick Clock

uint32_t CLOCK_GetFlexspiClkFreq(void)

Return Frequency of Flexspi Clock.

Returns

Frequency of Flexspi.

uint32_t CLOCK_GetUsimClkFreq(void)

Return Frequency of USIM Clock.

Returns

Frequency of USIM.

uint32_t CLOCK_GetGauClkFreq(void)

Return Frequency of GAU Clock.

Returns

Frequency of GAU.

uint32_t CLOCK_GetOSTimerClkFreq(void)

Return Frequency of OSTimer Clock.

Returns

Frequency of OSTimer.

uint32_t CLOCK_InitTcpuRefClk(uint32_t targetHz, *clock_tcpu_flexspi_div_t* div)

Initialize TCPU FVCO to target frequency. For 40MHz XTAL, FVCO ranges from 3000MHz to 3840MHz. For 38.4MHz XTAL, FVCO ranges from 2995.2MHz to 3840MHz.

Parameters

- targetHz – : Target FVCO frequency in Hz.
- div – : Divider for tcpu_mci_flexspi_clk.

Returns

Actual FVCO frequency in Hz.

void CLOCK_DeinitTcpuRefClk(void)

Deinit the TCPU reference clock.

void CLOCK_InitTddrRefClk(*clock_tddr_flexspi_div_t* div)

Initialize the TDDR reference clock.

Parameters

- div – : Divider for tddr_mci_flexspi_clk.

void CLOCK_DeinitTddrRefClk(void)

Deinit the TDDR reference clock.

void CLOCK_InitT3RefClk(*clock_t3_mci_irc_config_t* cnfg)

Initialize the T3 reference clock.

Parameters

- cnfg – : t3pll_mci_48_60m_irc clock configuration

void CLOCK_DeinitT3RefClk(void)

Deinit the T3 reference clock.

void CLOCK_InitAvPll(const *clock_avpll_config_t* *cnfg)

Initialize the AVPLL. Both channel 1 and 2 are enabled.

Parameters

- cnfg – : AVPLL clock configuration

void CLOCK_DeinitAvPll(void)

Deinit the AVPLL. All channels are disabled.

void CLOCK_ConfigAvPllCh(*clock_avpll_ch_freq_t* ch1Freq, *clock_avpll_ch_freq_t* ch2Freq, bool enableCali)

Update the AVPLL channel configuration. Enable/Disable state keeps unchanged.

Parameters

- ch1Freq – : Channel 1 frequency to set.
- ch2Freq – : Channel 2 frequency to set.
- enableCali – : Enable AVPLL calibration.

void CLOCK_EnableAvPllCh(bool enableCh1, bool enableCh2, bool enableCali)

Enable the AVPLL channel.

Parameters

- enableCh1 – : Enable AVPLL channel1, channel unchanged on false.
- enableCh2 – : Enable AVPLL channel2, channel unchanged on false.
- enableCali – : Enable AVPLL calibration.

`void CLOCK_DisableAvPllCh(bool disableCh1, bool disableCh2)`

Disable the AVPLL.

Parameters

- `disableCh1` – : Disable AVPLL channel1, channel unchanged on false.
- `disableCh2` – : Disable AVPLL channel2, channel unchanged on false.

`void CLOCK_EnableUsbhsPhyClock(void)`

Enable USB HS PHY PLL clock.

This function enables USB HS PHY PLL clock.

`void CLOCK_DisableUsbhsPhyClock(void)`

Disable USB HS PHY PLL clock.

This function disables USB HS PHY PLL clock.

`FSL_CLOCK_DRIVER_VERSION`

CLOCK driver version 2.3.2.

`SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY`

`GPIO_CLOCKS`

Clock ip name array for GPIO.

`CACHE64_CLOCKS`

Clock ip name array for CACHE64.

`FLEXSPI_CLOCKS`

Clock ip name array for FLEXSPI.

`FLEXCOMM_CLOCKS`

Clock ip name array for FLEXCOMM.

`USART_CLOCKS`

Clock ip name array for LPUART.

`I2C_CLOCKS`

Clock ip name array for I2C.

`SPI_CLOCKS`

Clock ip name array for SPI.

`ACOMP_CLOCKS`

Clock ip name array for ACOMP.

`ADC_CLOCKS`

Clock ip name array for ADC.

`DAC_CLOCKS`

Clock ip name array for DAC.

`LCDIC_CLOCKS`

Clock ip name array for LCDIC.

`DMA_CLOCKS`

Clock ip name array for DMA.

`DMIC_CLOCKS`

Clock ip name array for DMIC.

`ENET_CLOCKS`

Clock ip name array for ENET.

ENET_EXTRA_CLOCKS

Extra clock ip name array for ENET.

POWERQUAD_CLOCKS

Clock ip name array for Powerquad.

OSTIMER_CLOCKS

Clock ip name array for OSTimer.

CTIMER_CLOCKS

Clock ip name array for CT32B.

UTICK_CLOCKS

Clock ip name array for UTICK.

MRT_CLOCKS

Clock ip name array for MRT.

SCT_CLOCKS

Clock ip name array for SCT.

RTC_CLOCKS

Clock ip name array for RTC.

WWDT_CLOCKS

Clock ip name array for WWDT.

TRNG_CLOCKS

Clock ip name array for TRNG.

USIM_CLOCKS

Clock ip name array for USIM.

CLK_GATE_REG_OFFSET_SHIFT

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

CLK_GATE_REG_OFFSET_MASK

CLK_GATE_BIT_SHIFT_SHIFT

CLK_GATE_BIT_SHIFT_MASK

CLK_GATE_DEFINE(reg_offset, bit_shift)

CLK_GATE_ABSTRACT_REG_OFFSET(x)

CLK_GATE_ABSTRACT_BITS_SHIFT(x)

CLK_CTL0_PSCCTL0

CLK_CTL0_PSCCTL1

CLK_CTL0_PSCCTL2

CLK_CTL1_PSCCTL0

CLK_CTL1_PSCCTL1

CLK_CTL1_PSCCTL2

SYS_CLK_GATE_FLAG_MASK

SYS_CLK_GATE_DEFINE(bit_shift)

SYS_CLK_GATE_BIT_MASK(x)
CLKCTL0_TUPLE_MUXA(reg, choice)
CLKCTL0_TUPLE_MUXB(reg, choice)
CLKCTL1_TUPLE_FLAG_MASK
CLKCTL1_TUPLE_MUXA(reg, choice)
CLKCTL1_TUPLE_MUXB(reg, choice)
CLKCTL_TUPLE_REG(base, tuple)
CLKCTL_TUPLE_SEL(tuple)
CLKOUT_TUPLE_MUX_AVAIL
CLKOUT_TUPLE_MUX(ch0, ch1, ch2)
PMU_TUPLE_MUX_AVAIL
PMU_TUPLE_MUX(reg, choice)
PMU_TUPLE_REG(base, tuple)
PMU_TUPLE_SEL(tuple)

Values:

enumerator kCLOCK_FrgMainClk
Main System clock
enumerator kCLOCK_FrgPllDiv
Main pll clock divider
enumerator kCLOCK_FrgSFro
16MHz FRO
enumerator kCLOCK_FrgFFro
FRO48/60
uint8_t num
FRG clock
enum *clock_frg_clk_config* sfg_clock_src
uint8_t divider
Denominator of the fractional divider.
uint8_t mult
Numerator of the fractional divider.
clock_avpll_ch_freq_t ch1Freq
AVPLL channel 1 frequency configuration
clock_avpll_ch_freq_t ch2Freq
AVPLL channel 2 frequency configuration
bool enableCali
Enable calibration

FSL_SDK_DISABLE_DRIVER_CLOCK_CONTROL

Configure whether driver controls clock.

When set to 0, peripheral drivers will enable clock in initialize function and disable clock in de-initialize function. When set to 1, peripheral driver will not control the clock, application could control the clock out of the driver.

Note: All drivers share this feature switcher. If it is set to 1, application should handle clock enable and disable for all drivers.

struct _clock_frg_clk_config

#include <fsl_clock.h> PLL configuration for FRG.

struct clock_avpll_config_t

#include <fsl_clock.h> AVPLL configuration.

2.6 CRC: Cyclic Redundancy Check Driver

FSL_CRC_DRIVER_VERSION

CRC driver version. Version 2.1.1.

Current version: 2.1.1

Change log:

- Version 2.0.0
 - initial version
- Version 2.0.1
 - add explicit type cast when writing to WR_DATA
- Version 2.0.2
 - Fix MISRA issue
- Version 2.1.0
 - Add CRC_WriteSeed function
- Version 2.1.1
 - Fix MISRA issue

enum _crc_polynomial

CRC polynomials to use.

Values:

enumerator kCRC_Polynomial_CRC_CCITT

$x^{16}+x^{12}+x^5+1$

enumerator kCRC_Polynomial_CRC_16

$x^{16}+x^{15}+x^2+1$

enumerator kCRC_Polynomial_CRC_32

$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$

typedef enum _crc_polynomial crc_polynomial_t

CRC polynomials to use.

```
typedef struct _crc_config crc_config_t
```

CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

```
void CRC_Init(CRC_Type *base, const crc_config_t *config)
```

Enables and configures the CRC peripheral module.

This functions enables the CRC peripheral clock in the LPC SYSCON block. It also configures the CRC engine and starts checksum computation by writing the seed.

Parameters

- *base* – CRC peripheral address.
- *config* – CRC module configuration structure.

```
static inline void CRC_Deinit(CRC_Type *base)
```

Disables the CRC peripheral module.

This functions disables the CRC peripheral clock in the LPC SYSCON block.

Parameters

- *base* – CRC peripheral address.

```
void CRC_Reset(CRC_Type *base)
```

resets CRC peripheral module.

Parameters

- *base* – CRC peripheral address.

```
void CRC_WriteSeed(CRC_Type *base, uint32_t seed)
```

Write seed to CRC peripheral module.

Parameters

- *base* – CRC peripheral address.
- *seed* – CRC Seed value.

```
void CRC_GetDefaultConfig(crc_config_t *config)
```

Loads default values to CRC protocol configuration structure.

Loads default values to CRC protocol configuration structure. The default values are:

```
config->polynomial = kCRC_Polynomial_CRC_CCITT;
config->reverseIn = false;
config->complementIn = false;
config->reverseOut = false;
config->complementOut = false;
config->seed = 0xFFFFU;
```

Parameters

- *config* – CRC protocol configuration structure

```
void CRC_GetConfig(CRC_Type *base, crc_config_t *config)
```

Loads actual values configured in CRC peripheral to CRC protocol configuration structure.

The values, including seed, can be used to resume CRC calculation later.

Parameters

- *base* – CRC peripheral address.
- *config* – CRC protocol configuration structure

```
void CRC_WriteData(CRC_Type *base, const uint8_t *data, size_t dataSize)
```

Writes data to the CRC module.

Writes input data buffer bytes to CRC data register.

Parameters

- `base` – CRC peripheral address.
- `data` – Input data stream, MSByte in `data[0]`.
- `dataSize` – Size of the input data buffer in bytes.

```
static inline uint32_t CRC_Get32bitResult(CRC_Type *base)
```

Reads 32-bit checksum from the CRC module.

Reads CRC data register.

Parameters

- `base` – CRC peripheral address.

Returns

final 32-bit checksum, after configured bit reverse and complement operations.

```
static inline uint16_t CRC_Get16bitResult(CRC_Type *base)
```

Reads 16-bit checksum from the CRC module.

Reads CRC data register.

Parameters

- `base` – CRC peripheral address.

Returns

final 16-bit checksum, after configured bit reverse and complement operations.

```
CRC_DRIVER_USE_CRC16_CCITT_FALSE_AS_DEFAULT
```

Default configuration structure filled by `CRC_GetDefaultConfig()`. Uses CRC-16/CCITT-FALSE as default.

```
struct __crc_config
```

#include <fsl_crc.h> CRC protocol configuration.

This structure holds the configuration for the CRC protocol.

Public Members

`crc_polynomial_t` polynomial

CRC polynomial.

`bool reverseIn`

Reverse bits on input.

`bool complementIn`

Perform 1's complement on input.

`bool reverseOut`

Reverse bits on output.

`bool complementOut`

Perform 1's complement on output.

`uint32_t seed`

Starting checksum value.

2.7 CTIMER: Standard counter/timers

`void CTIMER_Init(CTIMER_Type *base, const ctimer_config_t *config)`

Ungates the clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application before using the driver.

Parameters

- `base` – Ctimer peripheral base address
- `config` – Pointer to the user configuration structure.

`void CTIMER_Deinit(CTIMER_Type *base)`

Gates the timer clock.

Parameters

- `base` – Ctimer peripheral base address

`void CTIMER_GetDefaultConfig(ctimer_config_t *config)`

Fills in the timers configuration structure with the default settings.

The default values are:

```
config->mode = kCTIMER_TimerMode;
config->input = kCTIMER_Capture_0;
config->prescale = 0;
```

Parameters

- `config` – Pointer to the user configuration structure.

`status_t CTIMER_SetupPwmPeriod(CTIMER_Type *base, const ctimer_match_t pwmPeriodChannel, ctimer_match_t matchChannel, uint32_t pwmPeriod, uint32_t pulsePeriod, bool enableInt)`

Configures the PWM signal parameters.

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function can manually assign the specified channel to set the PWM cycle.

Note: When setting PWM output from multiple output pins, all should use the same PWM period

Parameters

- `base` – Ctimer peripheral base address
- `pwmPeriodChannel` – Specify the channel to control the PWM period
- `matchChannel` – Match pin to be used to output the PWM signal
- `pwmPeriod` – PWM period match value
- `pulsePeriod` – Pulse width match value
- `enableInt` – Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt will be generated.

Returns

kStatus_Success on success kStatus_Fail If matchChannel is equal to pwmPeriodChannel; this channel is reserved to set the PWM cycle If PWM pulse width register value is larger than 0xFFFFFFFF.

```
status_t CTIMER_SetupPwm(CTIMER_Type *base, const ctimer_match_t pwmPeriodChannel,
                        ctimer_match_t matchChannel, uint8_t dutyCyclePercent, uint32_t
                        pwmFreq_Hz, uint32_t srcClock_Hz, bool enableInt)
```

Configures the PWM signal parameters.

Enables PWM mode on the match channel passed in and will then setup the match value and other match parameters to generate a PWM signal. This function can manually assign the specified channel to set the PWM cycle.

Note: When setting PWM output from multiple output pins, all should use the same PWM frequency. Please use CTIMER_SetupPwmPeriod to set up the PWM with high resolution.

Parameters

- base – Ctimer peripheral base address
- pwmPeriodChannel – Specify the channel to control the PWM period
- matchChannel – Match pin to be used to output the PWM signal
- dutyCyclePercent – PWM pulse width; the value should be between 0 to 100
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – Timer counter clock in Hz
- enableInt – Enable interrupt when the timer value reaches the match value of the PWM pulse, if it is 0 then no interrupt will be generated.

```
static inline void CTIMER_UpdatePwmPulsePeriod(CTIMER_Type *base, ctimer_match_t
                                              matchChannel, uint32_t pulsePeriod)
```

Updates the pulse period of an active PWM signal.

Parameters

- base – Ctimer peripheral base address
- matchChannel – Match pin to be used to output the PWM signal
- pulsePeriod – New PWM pulse width match value

```
status_t CTIMER_UpdatePwmDutycycle(CTIMER_Type *base, const ctimer_match_t
                                   pwmPeriodChannel, ctimer_match_t matchChannel,
                                   uint8_t dutyCyclePercent)
```

Updates the duty cycle of an active PWM signal.

Note: Please use CTIMER_SetupPwmPeriod to update the PWM with high resolution. This function can manually assign the specified channel to set the PWM cycle.

Parameters

- base – Ctimer peripheral base address
- pwmPeriodChannel – Specify the channel to control the PWM period
- matchChannel – Match pin to be used to output the PWM signal
- dutyCyclePercent – New PWM pulse width; the value should be between 0 to 100

Returns

kStatus_Success on success kStatus_Fail If PWM pulse width register value is larger than 0xFFFFFFFF.

```
static inline void CTIMER_EnableInterrupts(CTIMER_Type *base, uint32_t mask)
```

Enables the selected Timer interrupts.

Parameters

- base – Ctimer peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration ctimer_interrupt_enable_t

```
static inline void CTIMER_DisableInterrupts(CTIMER_Type *base, uint32_t mask)
```

Disables the selected Timer interrupts.

Parameters

- base – Ctimer peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration ctimer_interrupt_enable_t

```
static inline uint32_t CTIMER_GetEnabledInterrupts(CTIMER_Type *base)
```

Gets the enabled Timer interrupts.

Parameters

- base – Ctimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration ctimer_interrupt_enable_t

```
static inline uint32_t CTIMER_GetStatusFlags(CTIMER_Type *base)
```

Gets the Timer status flags.

Parameters

- base – Ctimer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration ctimer_status_flags_t

```
static inline void CTIMER_ClearStatusFlags(CTIMER_Type *base, uint32_t mask)
```

Clears the Timer status flags.

Parameters

- base – Ctimer peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration ctimer_status_flags_t

```
static inline void CTIMER_StartTimer(CTIMER_Type *base)
```

Starts the Timer counter.

Parameters

- base – Ctimer peripheral base address

```
static inline void CTIMER_StopTimer(CTIMER_Type *base)
```

Stops the Timer counter.

Parameters

- base – Ctimer peripheral base address

FSL_CTIMER_DRIVER_VERSION

Version 2.3.3

enum _ctimer_capture_channel

List of Timer capture channels.

Values:

enumerator kCTIMER_Capture_0

Timer capture channel 0

enumerator kCTIMER_Capture_1

Timer capture channel 1

enumerator kCTIMER_Capture_3

Timer capture channel 3

enum _ctimer_capture_edge

List of capture edge options.

Values:

enumerator kCTIMER_Capture_RiseEdge

Capture on rising edge

enumerator kCTIMER_Capture_FallEdge

Capture on falling edge

enumerator kCTIMER_Capture_BothEdge

Capture on rising and falling edge

enum _ctimer_match

List of Timer match registers.

Values:

enumerator kCTIMER_Match_0

Timer match register 0

enumerator kCTIMER_Match_1

Timer match register 1

enumerator kCTIMER_Match_2

Timer match register 2

enumerator kCTIMER_Match_3

Timer match register 3

enum _ctimer_external_match

List of external match.

Values:

enumerator kCTIMER_External_Match_0

External match 0

enumerator kCTIMER_External_Match_1

External match 1

enumerator kCTIMER_External_Match_2

External match 2

enumerator kCTIMER_External_Match_3

External match 3

enum _ctimer_match_output_control

List of output control options.

Values:

enumerator kCTIMER_Output_NoAction

No action is taken

enumerator kCTIMER_Output_Clear

Clear the EM bit/output to 0

enumerator kCTIMER_Output_Set

Set the EM bit/output to 1

enumerator kCTIMER_Output_Toggle

Toggle the EM bit/output

enum _ctimer_timer_mode

List of Timer modes.

Values:

enumerator kCTIMER_TimerMode

enumerator kCTIMER_IncreaseOnRiseEdge

enumerator kCTIMER_IncreaseOnFallEdge

enumerator kCTIMER_IncreaseOnBothEdge

enum _ctimer_interrupt_enable

List of Timer interrupts.

Values:

enumerator kCTIMER_Match0InterruptEnable

Match 0 interrupt

enumerator kCTIMER_Match1InterruptEnable

Match 1 interrupt

enumerator kCTIMER_Match2InterruptEnable

Match 2 interrupt

enumerator kCTIMER_Match3InterruptEnable

Match 3 interrupt

enum _ctimer_status_flags

List of Timer flags.

Values:

enumerator kCTIMER_Match0Flag

Match 0 interrupt flag

enumerator kCTIMER_Match1Flag

Match 1 interrupt flag

enumerator kCTIMER_Match2Flag

Match 2 interrupt flag

enumerator kCTIMER_Match3Flag

Match 3 interrupt flag

enum *ctimer_callback_type_t*

Callback type when registering for a callback. When registering a callback an array of function pointers is passed the size could be 1 or 8, the callback type will tell that.

Values:

enumerator *kCTIMER_SingleCallback*

Single Callback type where there is only one callback for the timer. based on the status flags different channels needs to be handled differently

enumerator *kCTIMER_MultipleCallback*

Multiple Callback type where there can be 8 valid callbacks, one per channel. for both match/capture

typedef enum *_ctimer_capture_channel* *ctimer_capture_channel_t*

List of Timer capture channels.

typedef enum *_ctimer_capture_edge* *ctimer_capture_edge_t*

List of capture edge options.

typedef enum *_ctimer_match* *ctimer_match_t*

List of Timer match registers.

typedef enum *_ctimer_external_match* *ctimer_external_match_t*

List of external match.

typedef enum *_ctimer_match_output_control* *ctimer_match_output_control_t*

List of output control options.

typedef enum *_ctimer_timer_mode* *ctimer_timer_mode_t*

List of Timer modes.

typedef enum *_ctimer_interrupt_enable* *ctimer_interrupt_enable_t*

List of Timer interrupts.

typedef enum *_ctimer_status_flags* *ctimer_status_flags_t*

List of Timer flags.

typedef void (**ctimer_callback_t*)(uint32_t flags)

typedef struct *_ctimer_match_config* *ctimer_match_config_t*

Match configuration.

This structure holds the configuration settings for each match register.

typedef struct *_ctimer_config* *ctimer_config_t*

Timer configuration structure.

This structure holds the configuration settings for the Timer peripheral. To initialize this structure to reasonable defaults, call the `CTIMER_GetDefaultConfig()` function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

void `CTIMER_SetupMatch(CTIMER_Type *base, ctimer_match_t matchChannel, const ctimer_match_config_t *config)`

Setup the match register.

User configuration is used to setup the match value and action to be taken when a match occurs.

Parameters

- `base` – Ctimer peripheral base address
- `matchChannel` – Match register to configure

- `config` – Pointer to the match configuration structure

```
uint32_t CTIMER_GetOutputMatchStatus(CTIMER_Type *base, uint32_t matchChannel)
```

Get the status of output match.

This function gets the status of output MAT, whether or not this output is connected to a pin. This status is driven to the MAT pins if the match function is selected via IOCON. 0 = LOW. 1 = HIGH.

Parameters

- `base` – Ctimer peripheral base address
- `matchChannel` – External match channel, user can obtain the status of multiple match channels at the same time by using the logic of “|” enumeration `ctimer_external_match_t`

Returns

The mask of external match channel status flags. Users need to use the `_ctimer_external_match` type to decode the return variables.

```
void CTIMER_SetupCapture(CTIMER_Type *base, ctimer_capture_channel_t capture,
                        ctimer_capture_edge_t edge, bool enableInt)
```

Setup the capture.

Parameters

- `base` – Ctimer peripheral base address
- `capture` – Capture channel to configure
- `edge` – Edge on the channel that will trigger a capture
- `enableInt` – Flag to enable channel interrupts, if enabled then the registered call back is called upon capture

```
static inline uint32_t CTIMER_GetTimerCountValue(CTIMER_Type *base)
```

Get the timer count value from TC register.

Parameters

- `base` – Ctimer peripheral base address.

Returns

return the timer count value.

```
void CTIMER_RegisterCallBack(CTIMER_Type *base, ctimer_callback_t *cb_func,
                           ctimer_callback_type_t cb_type)
```

Register callback.

This function configures CTimer Callback in following modes:

- **Single Callback:** `cb_func` should be pointer to callback function pointer
For example: `ctimer_callback_t ctimer_callback = pwm_match_callback;`
`CTIMER_RegisterCallBack(CTIMER, &ctimer_callback, kCTIMER_SingleCallback);`
- **Multiple Callback:** `cb_func` should be pointer to array of callback function pointers Each element corresponds to Interrupt Flag in IR register.
For example: `ctimer_callback_t ctimer_callback_table[] = { ctimer_match0_callback, NULL, NULL, ctimer_match3_callback, NULL, NULL, NULL, NULL};`
`CTIMER_RegisterCallBack(CTIMER, &ctimer_callback_table[0], kCTIMER_MultipleCallback);`

Parameters

- `base` – Ctimer peripheral base address
- `cb_func` – Pointer to callback function pointer

- `cb_type` – callback function type, singular or multiple

static inline void CTIMER_Reset(CTIMER_Type *base)

Reset the counter.

The timer counter and prescale counter are reset on the next positive edge of the APB clock.

Parameters

- `base` – Ctimer peripheral base address

static inline void CTIMER_SetPrescale(CTIMER_Type *base, uint32_t prescale)

Setup the timer prescale value.

Specifies the maximum value for the Prescale Counter.

Parameters

- `base` – Ctimer peripheral base address
- `prescale` – Prescale value

static inline uint32_t CTIMER_GetCaptureValue(CTIMER_Type *base, *ctimer_capture_channel_t* capture)

Get capture channel value.

Get the counter/timer value on the corresponding capture channel.

Parameters

- `base` – Ctimer peripheral base address
- `capture` – Select capture channel

Returns

The timer count capture value.

static inline void CTIMER_EnableResetMatchChannel(CTIMER_Type *base, *ctimer_match_t* match, bool enable)

Enable reset match channel.

Set the specified match channel reset operation.

Parameters

- `base` – Ctimer peripheral base address
- `match` – match channel used
- `enable` – Enable match channel reset operation.

static inline void CTIMER_EnableStopMatchChannel(CTIMER_Type *base, *ctimer_match_t* match, bool enable)

Enable stop match channel.

Set the specified match channel stop operation.

Parameters

- `base` – Ctimer peripheral base address.
- `match` – match channel used.
- `enable` – Enable match channel stop operation.

static inline void CTIMER_EnableMatchChannelReload(CTIMER_Type *base, *ctimer_match_t* match, bool enable)

Enable reload channel falling edge.

Enable the specified match channel reload match shadow value.

Parameters

- base – Ctimer peripheral base address.
- match – match channel used.
- enable – Enable .

```
static inline void CTIMER_EnableRisingEdgeCapture(CTIMER_Type *base,
                                                ctimer_capture_channel_t capture, bool
                                                enable)
```

Enable capture channel rising edge.

Sets the specified capture channel for rising edge capture.

Parameters

- base – Ctimer peripheral base address.
- capture – capture channel used.
- enable – Enable rising edge capture.

```
static inline void CTIMER_EnableFallingEdgeCapture(CTIMER_Type *base,
                                                  ctimer_capture_channel_t capture, bool
                                                  enable)
```

Enable capture channel falling edge.

Sets the specified capture channel for falling edge capture.

Parameters

- base – Ctimer peripheral base address.
- capture – capture channel used.
- enable – Enable falling edge capture.

```
static inline void CTIMER_SetShadowValue(CTIMER_Type *base, ctimer_match_t match,
                                         uint32_t matchvalue)
```

Set the specified match shadow channel.

Parameters

- base – Ctimer peripheral base address.
- match – match channel used.
- matchvalue – Reload the value of the corresponding match register.

```
struct __ctimer_match_config
#include <fsl_ctimer.h> Match configuration.
```

This structure holds the configuration settings for each match register.

Public Members

uint32_t matchValue

This is stored in the match register

bool enableCounterReset

true: Match will reset the counter false: Match will not reset the counter

bool enableCounterStop

true: Match will stop the counter false: Match will not stop the counter

ctimer_match_output_control_t outControl

Action to be taken on a match on the EM bit/output

bool outPinInitState

Initial value of the EM bit/output

bool enableInterrupt

true: Generate interrupt upon match false: Do not generate interrupt on match

struct *_ctimer_config*

#include <fsl_ctimer.h> Timer configuration structure.

This structure holds the configuration settings for the Timer peripheral. To initialize this structure to reasonable defaults, call the CTIMER_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

Public Members

ctimer_timer_mode_t mode

Timer mode

ctimer_capture_channel_t input

Input channel to increment the timer, used only in timer modes that rely on this input signal to increment TC

uint32_t prescale

Prescale value

2.8 DAC: Digital Analog Converter

void DAC_Init(DAC_Type *base, const *dac_config_t* *config)

Initializes DAC module, including set reference voltage source, set conversion range, and set output voltage range.

Parameters

- base – DAC peripheral base address.
- config – Pointer to the structure which in type of *dac_config_t*.

void DAC_GetDefaultConfig(*dac_config_t* *config)

Gets the default configurations of DAC module.

```
config->conversionRate = kDAC_ConversionRate62P5KHZ;  
config->refSource = kDAC_ReferenceInternalVoltageSource;  
config->rangeSelect = kDAC_RangeLarge;
```

Parameters

- config – Pointer to the structure which in the type of *dac_config_t*.

void DAC_Deinit(DAC_Type *base)

De-initializes the DAC module, including reset clock divider, reset each channel, and so on.

Parameters

- base – DAC peripheral base address.

```
void DAC_SetChannelConfig(DAC_Type *base, uint32_t channelMask, const
                        dac_channel_config_t *channelConfig)
```

Configures the DAC channels, including enable channel conversion, set wave type, set timing mode, and so on.

Parameters

- `base` – DAC peripheral base address.
- `channelMask` – The mask of channel, can be the OR'ed value of `dac_channel_id_t`.
- `channelConfig` – The pointer of structure which in the type of `dac_channel_config_t`.

```
static inline void DAC_ResetChannel(DAC_Type *base, uint32_t channelMask)
```

Does software reset for the selected DAC channels.

Parameters

- `base` – DAC peripheral base address.
- `channelMask` – The mask of channel to be reset, should be the OR'ed value of `dac_channel_id_t`.

```
static inline void DAC_EnableChannelConversion(DAC_Type *base, uint32_t channelMask, bool
                                              enable)
```

Enables/Disables selected channel conversion.

Note: To enable/disable the conversions of both channels, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB** .

Parameters

- `base` – DAC peripheral base address.
- `channelMask` – The mask of channel to be reset, can be the OR'ed value of `dac_channel_id_t`.
- `enable` – Enable/Disable channel conversion.
 - **true** Enable selected channels' conversion.
 - **false** Disable selected channels' conversion.

```
static inline void DAC_SetChannelOutMode(DAC_Type *base, uint32_t channelMask,
                                         dac_channel_output_t outMode)
```

Sets channels out mode, including `kDAC_ChannelOutputInternal` and `kDAC_ChannelOutputPad`.

Parameters

- `base` – DAC peripheral base address.
- `channelMask` – The mask of channel, can be the OR'ed value of `dac_channel_id_t`.
- `outMode` – The out mode of selected channels, please refer to `dac_channel_output_t` for details.

```
static inline void DAC_EnableChannelTriggerMode(DAC_Type *base, uint32_t channelMask, bool
                                              enable)
```

Enables/Disables channels trigger mode.

Note: To enable/disable the trigger mode of both two channels, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB** .

Parameters

- **base** – DAC peripheral base address.
- **channelMask** – The mask of channel, can be the OR'ed value of **dac_channel_id_t**.
- **enable** – Enable/Disable channel trigger mode.
 - **true** Channels' conversion triggered by external event enabled.
 - **false** Channels' conversion triggered by external event disabled.

```
static inline void DAC_SetChannelTrigSource(DAC_Type *base, uint32_t channelMask,  
                                           dac_channel_trigger_source_t trigSource)
```

Sets channels trigger source.

Note: To set the same trigger source to both two channels, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB** .

Parameters

- **base** – DAC peripheral base address.
- **channelMask** – The mask of channel, can be the OR'ed value of **dac_channel_id_t**.
- **trigSource** – The selected trigger source, please refer to **dac_channel_trigger_source_t** for details.

```
static inline void DAC_SetChannelTrigType(DAC_Type *base, uint32_t channelMask,  
                                         dac_channel_trigger_type_t trigType)
```

Sets channels trigger type, such as rising edge trigger, falling edge trigger, or both edge trigger.

Note: To set the same trigger type to both two channels, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB** .

Parameters

- **base** – DAC peripheral base address.
- **channelMask** – The mask of channel, can be the OR'ed value of **dac_channel_id_t**;
- **trigType** – The selected trigger type, please refer to **dac_channel_trigger_type_t**;

```
static inline void DAC_SetChannelTimingMode(DAC_Type *base, uint32_t channelMask,  
                                           dac_channel_timing_mode_t timingMode)
```

Sets channels timing mode, including not-timing related or timing related.

Note: To the same timing mode to both two channels, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB** .

Parameters

- base – DAC peripheral base address.
- channelMask – The mask of channel, can be the OR'ed value of `dac_channel_id_t`.
- timingMode – The selected timing mode, please refer to `dac_channel_timing_mode_t` for details.

```
static inline void DAC_EnableChannelDMA(DAC_Type *base, uint32_t channelMask, bool enable)
```

Enables/Disables channels DMA.

Parameters

- base – DAC peripheral base address.
- channelMask – The mask of channel, can be the OR'ed value of `dac_channel_id_t`.
- enable – Enable/Disable channel DMA data transfer.
 - **true** DMA data transfer enabled.
 - **false** DMA data transfer disabled.

```
static inline void DAC_SetChannelWaveType(DAC_Type *base, uint32_t channelMask, dac_channel_wave_type_t waveType)
```

Sets channels wave type, such as sine, noise, or triangle.

Note: To set the same wave type to both channel, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB**.

Parameters

- base – DAC peripheral base address.
- channelMask – The mask of channel, should be the OR'ed value of `dac_channel_id_t`.
- waveType – The wave type to set, please refer to `dac_channel_wave_type_t`.

```
static inline void DAC_SetChannelData(DAC_Type *base, uint32_t channelMask, uint16_t data)
```

Sets DAC channels data.

Note: To set the same data to both channel, invoking this API with the parameter **channelMask** set as **kDAC_ChannelA | kDAC_ChannelB**.

Parameters

- base – DAC peripheral base address.
- channelMask – The mask of channel, can be the OR'ed value of `dac_channel_id_t`.
- data –

```
void DAC_SetTriangleConfig(DAC_Type *base, const dac_triangle_config_t *triangleConfig)
```

Configures the options of triangle waveform.

Note: This API should be invoked to set the options of triangle waveform when channel A's output wave type is selected as kDAC_WaveTriangle.

Parameters

- base – DAC peripheral base address.
- triangleConfig – The pointer of structure which in the type of `dac_triangle_config_t`.

static inline void DAC__EnableInterrupts(DAC_Type *base, uint32_t interruptMask)

Enables interrupts, such as channel A data ready interrupt, channel A timeout interrupt, and so on.

Parameters

- base – DAC peripheral base address.
- interruptMask – The or'ed value of the interrupts to be enabled, please refer to `_dac_interrupt_enable`.

static inline void DAC__DisableInterrupts(DAC_Type *base, uint32_t interruptMask)

Disables interrupts, such as channel B data ready interrupt, channel B timeout interrupt, and so on.

Parameters

- base – DAC peripheral base address.
- interruptMask – The or'ed value of the interrupts to be disabled, please refer to `_dac_interrupt_enable`.

static inline uint32_t DAC__GetStatusFlags(DAC_Type *base)

Gets the status flags, including interrupt status flags, raw status flags, and conversion status flags.

Parameters

- base – DAC peripheral base address.

Returns

The mask of status flags, please refer to `_dac_status_flags`.

static inline void DAC__ClearStatusFlags(DAC_Type *base, uint32_t statusFlagsMask)

Clears the interrupts status flags, such as channel A data ready interrupt flag, channel B data ready interrupt flag, and so on.

Parameters

- base – DAC peripheral base address.
- statusFlagsMask – The mask of the status flags to be cleared, please refer to `_dac_status_flags`.

enum _dac_interrupt_enable

The enumeration of interrupts that DAC support.

Values:

enumerator kDAC_ChannelAReadyInterruptEnable

Enable channel A data ready interrupt.

enumerator kDAC_ChannelBReadyInterruptEnable

Enable channel B data ready interrupt.

enumerator kDAC_ChannelATimeoutInterruptEnable
Enable channel A time out interrupt.

enumerator kDAC_ChannelBTimeoutInterruptEnable
Enable channel B time out interrupt.

enumerator kDAC_TriangleOverflowInterruptEnable
Enable triangle overflow interrupt.

enum _dac_status_flags

The enumeration of DAC status flags, including interrupt status flags, raw status flags, and conversion status flags.

Note: The interrupt status flags can only be asserted upon both enabling and happening of related interrupts. Comparatively, the raw status flags will be asserted as long as related events happen regardless of whether related interrupts are enabled or not.

Note: Only interrupt status flags can be cleared manually.

Values:

enumerator kDAC_ChannelADataReadyInterruptFlag
Channel A data ready.

enumerator kDAC_ChannelBDataReadyInterruptFlag
Channel B data ready.

enumerator kDAC_ChannelATimeoutInterruptFlag
Channel A time out.

enumerator kDAC_ChannelBTimeoutInterruptFlag
Channel B time out.

enumerator kDAC_TriangleOverflowInterruptFlag
Triangle overflow.

enumerator kDAC_RawChannelADataReadyFlag
Channel A data ready raw.

enumerator kDAC_RawChannelBDataReadyFlag
Channel B data ready raw.

enumerator kDAC_RawChannelATimeoutFlag
Channel A timeout raw.

enumerator kDAC_RawChannelBTimeoutFlag
Channel B timeout raw.

enumerator kDAC_RawTriangleOverflowFlag
Triangle overflow raw.

enumerator kDAC_ChannelAConversionCompleteFlag
Channel A conversion complete.

enumerator kDAC_ChannelBConversionCompleteFlag
Channel B conversion complete.

enum _dac_channel_id

The enumeration of dac channels, including channel A and channel B.

Values:

enumerator kDAC_ChannelA

enumerator kDAC_ChannelB

enum _dac_conversion_rate

The enumeration of dac conversion rate, including 62.5 KHz, 125 KHz, 250 KHz, and 500 KHz.

Values:

enumerator kDAC_ConversionRate62P5KHZ

DAC Conversion Rate selects as 62.5 KHz.

enumerator kDAC_ConversionRate125KHZ

DAC Conversion Rate selects as 125 KHz.

enumerator kDAC_ConversionRate250KHZ

DAC Conversion Rate selects as 250 KHz.

enumerator kDAC_ConversionRate500KHZ

DAC Conversion Rate selects as 500 KHz.

enum _dac_reference_voltage_source

The enumeration of dac reference voltage source.

Values:

enumerator kDAC_ReferenceInternalVoltageSource

Select internal voltage reference.

enumerator kDAC_ReferenceExternalVoltageSource

Select external voltage reference.

enum _dac_output_voltage_range

The enumeration of dac output voltage range.

Values:

enumerator kDAC_RangeSmall

DAC output small range.

enumerator kDAC_RangeMiddle

DAC output middle range.

enumerator kDAC_RangeLarge

DAC output large range.

enum _dac_channel_output

The enumeration of dac channel's output mode.

Values:

enumerator kDAC_ChannelOutputInternal

Enable internal output but disable output to pad

enumerator kDAC_ChannelOutputPAD

Enable output to pad but disable internal output

enum _dac_channel_trigger_type

The enumeration of dac channel's trigger type, including rising edge trigger, falling edge trigger, and both edge triggers.

Values:

enumerator kDAC_RisingEdgeTrigger

Rising edge trigger.

enumerator kDAC_FallingEdgeTrigger

Falling edge trigger.

enumerator kDAC_BothEdgeTrigger

Rising and Falling edge trigger.

enum _dac_channel_timing_mode

The enumeration of dac channel timing mode.

Values:

enumerator kDAC_NonTimingCorrelated

DAC non-timing-correlated mode.

enumerator kDAC_TimingCorrelated

DAC timing-correlated mode.

enum _dac_channel_wave_type

The enumerator of channel output wave type, please note that not all wave types are effective to A and B channel.

Values:

enumerator kDAC_WaveNormal

No predefined waveform, effective to A or B channel

enumerator kDAC_WaveTriangle

Triangle wave, effective only to A channel

enumerator kDAC_WaveSine

Sine wave, effective only to A channel

enumerator kDAC_WaveNoiseDifferential

Noise wave, effective only to A channel; Differential mode, one's complemental code from A data, effective only to B channel

enum _dac_triangle_mamp

DAC triangle maximum amplitude type.

Values:

enumerator kDAC_TriangleAmplitude63

DAC triangle amplitude 63 lsb

enumerator kDAC_TriangleAmplitude127

DAC triangle amplitude 127 lsb

enumerator kDAC_TriangleAmplitude191

DAC triangle amplitude 191 lsb

enumerator kDAC_TriangleAmplitude255

DAC triangle amplitude 255 lsb

enumerator kDAC_TriangleAmplitude319

DAC triangle amplitude 319 lsb

enumerator kDAC_TriangleAmplitude383

DAC triangle amplitude 383 lsb

enumerator kDAC_TriangleAmplitude447

DAC triangle amplitude 447 lsb

enumerator kDAC_TriangleAmplitude511

DAC triangle amplitude 511 lsb

enumerator kDAC_TriangleAmplitude575

DAC triangle amplitude 575 lsb

enumerator kDAC_TriangleAmplitude639

DAC triangle amplitude 639 lsb

enumerator kDAC_TriangleAmplitude703

DAC triangle amplitude 703 lsb

enumerator kDAC_TriangleAmplitude767

DAC triangle amplitude 767 lsb

enumerator kDAC_TriangleAmplitude831

DAC triangle amplitude 831 lsb

enumerator kDAC_TriangleAmplitude895

DAC triangle amplitude 895 lsb

enumerator kDAC_TriangleAmplitude959

DAC triangle amplitude 959 lsb

enumerator kDAC_TriangleAmplitude1023

DAC triangle amplitude 1023 lsb

enum _dac_triangle_step_size

DAC triangle step size type.

Values:

enumerator kDAC_TriangleStepSize1

DAC triangle step size 1 lsb

enumerator kDAC_TriangleStepSize3

DAC triangle step size 3 lsb

enumerator kDAC_TriangleStepSize15

DAC triangle step size 15 lsb

enumerator kDAC_TriangleStepSize511

DAC triangle step size 511 lsb

enum _dac_triangle_waveform_type

DAC triangle waveform type.

Values:

enumerator kDAC_TriangleFull

DAC full triangle waveform

enumerator kDAC_TriangleHalf

DAC half triangle waveform

typedef enum _dac_channel_id dac_channel_id_t

The enumeration of dac channels, including channel A and channel B.

`typedef enum _dac_conversion_rate` `dac_conversion_rate_t`

The enumeration of dac conversion rate, including 62.5 KHz, 125 KHz, 250 KHz, and 500 KHz.

`typedef enum _dac_reference_voltage_source` `dac_reference_voltage_source_t`

The enumeration of dac reference voltage source.

`typedef enum _dac_output_voltage_range` `dac_output_voltage_range_t`

The enumeration of dac output voltage range.

`typedef enum _dac_channel_output` `dac_channel_output_t`

The enumeration of dac channel's output mode.

`typedef enum _dac_channel_trigger_type` `dac_channel_trigger_type_t`

The enumeration of dac channel's trigger type, including rising edge trigger, falling edge trigger, and both edge triggers.

`typedef enum _dac_channel_timing_mode` `dac_channel_timing_mode_t`

The enumeration of dac channel timing mode.

`typedef enum _dac_channel_wave_type` `dac_channel_wave_type_t`

The enumerator of channel output wave type, please note that not all wave types are effective to A and B channel.

`typedef enum _dac_triangle_mamp` `dac_triangle_mamp_t`

DAC triangle maximum amplitude type.

`typedef enum _dac_triangle_step_size` `dac_triangle_step_size_t`

DAC triangle step size type.

`typedef enum _dac_triangle_waveform_type` `dac_triangle_waveform_type_t`

DAC triangle waveform type.

`typedef struct _dac_config` `dac_config_t`

The structure of dac module basic configuration, including conversion rate, output range, and reference voltage source.

`typedef struct _dac_channel_config` `dac_channel_config_t`

The structure of dac channel configuration, such as trigger type, wave type, timing mode, and so on.

`typedef struct _dac_triangle_config` `dac_triangle_config_t`

The structure of triangle waveform, including maximum value, minimum value, step size, and so on.

`FSL_DAC_DRIVER_VERSION`

DAC driver version.

Version 2.1.1.

`IS_DAC_CHANNEL_A_WAVE(CH_WAVE)`

DAC channel A wave mode check.

`IS_DAC_CHANNEL_B_WAVE(CH_WAVE)`

DAC channel B wave mode check.

`struct _dac_config`

`#include <fsl_dac.h>` The structure of dac module basic configuration, including conversion rate, output range, and reference voltage source.

Public Members

dac_conversion_rate_t conversionRate

Configure DAC conversion rate, please refer to *dac_conversion_rate_t*.

dac_reference_voltage_source_t refSource

Configure DAC vref source, please refer to *dac_reference_voltage_source_t*.

dac_output_voltage_range_t rangeSelect

Configure DAC channel output range, please refer to *dac_output_voltage_range_t*.

struct *_dac_channel_config*

#include <fsl_dac.h> The structure of dac channel configuration, such as trigger type, wave type, timing mode, and so on.

Public Members

bool enableConversion

Enable/Disable selected channel's conversion.

- **true** Enable selected channel's conversion.
- **false** Disable selected channel's conversion.

dac_channel_output_t outMode

Configure channel output mode, please refer to *dac_channel_output_t*

bool enableDMA

Enable/Disable channel DAM data transfer.

- **true** DMA data transfer enabled.
- **false** DMA data transfer disabled.

bool enableTrigger

Enable/Disable external event trigger.

dac_channel_trigger_type_t triggerType

Configure the channel trigger type, please refer to *dac_channel_trigger_type_t*.

dac_channel_trigger_source_t triggerSource

Configure DAC channel trigger source, please refer to *dac_channel_trigger_source_t*.

dac_channel_timing_mode_t timingMode

Configure channel timing mode, please refer to *dac_channel_timing_mode_t*.

dac_channel_wave_type_t waveType

Configure wave type for the selected channel, please refer to *dac_channel_wave_type_t*.

struct *_dac_triangle_config*

#include <fsl_dac.h> The structure of triangle waveform, including maximum value, minimum value, step size, and so on.

Public Members

dac_triangle_mamp_t triangleMamp

Configure triangle maximum value.

dac_triangle_step_size_t triangleStepSize

Configure triangle step size.

dac_triangle_waveform_type_t triangleWaveform

Configure triangle waveform type.

uint32_t triangleBase

Configure triangle minimum value.

2.9 DMA: Direct Memory Access Controller Driver

`void DMA_Init(DMA_Type *base)`

Initializes DMA peripheral.

This function enable the DMA clock, set descriptor table and enable DMA peripheral.

Parameters

- *base* – DMA peripheral base address.

`void DMA_Deinit(DMA_Type *base)`

Deinitializes DMA peripheral.

This function gates the DMA clock.

Parameters

- *base* – DMA peripheral base address.

`void DMA_InstallDescriptorMemory(DMA_Type *base, void *addr)`

Install DMA descriptor memory.

This function used to register DMA descriptor memory for linked transfer, a typical case is ping pong transfer which will request more than one DMA descriptor memory space, although current DMA driver has a default DMA descriptor buffer, but it support one DMA descriptor for one channel only.

Parameters

- *base* – DMA base address.
- *addr* – DMA descriptor address

`static inline bool DMA_ChannelIsActive(DMA_Type *base, uint32_t channel)`

Return whether DMA channel is processing transfer.

Parameters

- *base* – DMA peripheral base address.
- *channel* – DMA channel number.

Returns

True for active state, false otherwise.

`static inline bool DMA_ChannelIsBusy(DMA_Type *base, uint32_t channel)`

Return whether DMA channel is busy.

Parameters

- *base* – DMA peripheral base address.
- *channel* – DMA channel number.

Returns

True for busy state, false otherwise.

static inline void DMA_EnableChannelInterrupts(DMA_Type *base, uint32_t channel)
Enables the interrupt source for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableChannelInterrupts(DMA_Type *base, uint32_t channel)
Disables the interrupt source for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_EnableChannel(DMA_Type *base, uint32_t channel)
Enable DMA channel.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableChannel(DMA_Type *base, uint32_t channel)
Disable DMA channel.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_EnableChannelPeriphRq(DMA_Type *base, uint32_t channel)
Set PERIPHREQEN of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableChannelPeriphRq(DMA_Type *base, uint32_t channel)
Get PERIPHREQEN value of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

True for enabled PeriphRq, false for disabled.

void DMA_ConfigureChannelTrigger(DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger)

Set trigger settings of DMA channel.

Deprecated:

Do not use this function. It has been superceded by DMA_SetChannelConfig.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

- trigger – trigger configuration.

```
void DMA_SetChannelConfig(DMA_Type *base, uint32_t channel, dma_channel_trigger_t  
                        *trigger, bool isPeriph)
```

set channel config.

This function provide a interface to configure channel configuration registers.

Parameters

- base – DMA base address.
- channel – DMA channel number.
- trigger – channel configurations structure.
- isPeriph – true is periph request, false is not.

```
static inline uint32_t DMA_SetChannelXferConfig(bool reload, bool clrTrig, bool intA, bool intB,  
                                              uint8_t width, uint8_t srcInc, uint8_t dstInc,  
                                              uint32_t bytes)
```

DMA channel xfer transfer configurations.

Parameters

- reload – true is reload link descriptor after current exhaust, false is not
- clrTrig – true is clear trigger status, wait software trigger, false is not
- intA – enable interruptA
- intB – enable interruptB
- width – transfer width
- srcInc – source address interleave size
- dstInc – destination address interleave size
- bytes – transfer bytes

Returns

The value of xfer config

```
uint32_t DMA_GetRemainingBytes(DMA_Type *base, uint32_t channel)
```

Gets the remaining bytes of the current DMA descriptor transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

The number of bytes which have not been transferred yet.

```
static inline void DMA_SetChannelPriority(DMA_Type *base, uint32_t channel, dma_priority_t  
                                       priority)
```

Set priority of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- priority – Channel priority value.

```
static inline dma_priority_t DMA_GetChannelPriority(DMA_Type *base, uint32_t channel)
```

Get priority of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

Channel priority value.

```
static inline void DMA_SetChannelConfigValid(DMA_Type *base, uint32_t channel)
```

Set channel configuration valid.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

```
static inline void DMA_DoChannelSoftwareTrigger(DMA_Type *base, uint32_t channel)
```

Do software trigger for the channel.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

```
static inline void DMA_LoadChannelTransferConfig(DMA_Type *base, uint32_t channel, uint32_t  
xfer)
```

Load channel transfer configurations.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- xfer – transfer configurations.

```
void DMA_CreateDescriptor(dma_descriptor_t *desc, dma_xfercfg_t *xfercfg, void *srcAddr, void  
*dstAddr, void *nextDesc)
```

Create application specific DMA descriptor to be used in a chain in transfer.

Deprecated:

Do not use this function. It has been superceded by DMA_SetupDescriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcAddr – Address of last item to transmit
- dstAddr – Address of last item to receive.
- nextDesc – Address of next descriptor in chain.

```
void DMA_SetupDescriptor(dma_descriptor_t *desc, uint32_t xfercfg, void *srcStartAddr, void  
*dstStartAddr, void *nextDesc)
```

setup dma descriptor

Note: This function do not support configure wrap descriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcStartAddr – Start address of source address.
- dstStartAddr – Start address of destination address.
- nextDesc – Address of next descriptor in chain.

```
void DMA_SetupChannelDescriptor(dma_descriptor_t *desc, uint32_t xfercfg, void *srcStartAddr,
                               void *dstStartAddr, void *nextDesc, dma_burst_wrap_t
                               wrapType, uint32_t burstSize)
```

setup dma channel descriptor

Note: This function support configure wrap descriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcStartAddr – Start address of source address.
- dstStartAddr – Start address of destination address.
- nextDesc – Address of next descriptor in chain.
- wrapType – burst wrap type.
- burstSize – burst size, reference `_dma_burst_size`.

```
void DMA_LoadChannelDescriptor(DMA_Type *base, uint32_t channel, dma_descriptor_t
                               *descriptor)
```

load channel transfer decriptor.

This function can be used to load decriptor to driver internal channel descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, it is useful for the case:

- for the polling transfer, application can allocate a local descriptor memory table to prepare a descriptor firstly and then call this api to load the configured descriptor to driver descriptor table.

```
DMA_Init(DMA0);
DMA_EnableChannel(DMA0, DEMO_DMA_CHANNEL);
DMA_SetupDescriptor(desc, xferCfg, s_srcBuffer, &s_destBuffer[0], NULL);
DMA_LoadChannelDescriptor(DMA0, DEMO_DMA_CHANNEL, (dma_descriptor_t *)desc);
DMA_DoChannelSoftwareTrigger(DMA0, DEMO_DMA_CHANNEL);
while(DMA_ChannelIsBusy(DMA0, DEMO_DMA_CHANNEL))
{ }
```

Parameters

- base – DMA base address.
- channel – DMA channel.
- descriptor – configured DMA descriptor.

```
void DMA_AbortTransfer(dma_handle_t *handle)
```

Abort running transfer by handle.

This function aborts DMA transfer specified by handle.

Parameters

- handle – DMA handle pointer.

void DMA_CreateHandle(*dma_handle_t* *handle, DMA_Type *base, uint32_t channel)

Creates the DMA handle.

This function is called if using transaction API for DMA. This function initializes the internal state of DMA handle.

Parameters

- handle – DMA handle pointer. The DMA handle stores callback function and parameters.
- base – DMA peripheral base address.
- channel – DMA channel number.

void DMA_SetCallback(*dma_handle_t* *handle, *dma_callback* callback, void *userData)

Installs a callback function for the DMA transfer.

This callback is called in DMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

- handle – DMA handle pointer.
- callback – DMA callback function pointer.
- userData – Parameter for callback function.

void DMA_PrepareTransfer(*dma_transfer_config_t* *config, void *srcAddr, void *dstAddr, uint32_t byteWidth, uint32_t transferBytes, *dma_transfer_type_t* type, void *nextDesc)

Prepares the DMA transfer structure.

Deprecated:

Do not use this function. It has been superceded by DMA_PrepareChannelTransfer. This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, so the source address must be 4 bytes aligned, or it shall result in source address error(SAE).

Parameters

- config – The user configuration structure of type *dma_transfer_t*.
- srcAddr – DMA transfer source address.
- dstAddr – DMA transfer destination address.
- byteWidth – DMA transfer destination address width(bytes).
- transferBytes – DMA transfer bytes to be transferred.
- type – DMA transfer type.
- nextDesc – Chain custom descriptor to transfer.

void DMA_PrepareChannelTransfer(*dma_channel_config_t* *config, void *srcStartAddr, void *dstStartAddr, uint32_t xferCfg, *dma_transfer_type_t* type, *dma_channel_trigger_t* *trigger, void *nextDesc)

Prepare channel transfer configurations.

This function used to prepare channel transfer configurations.

Parameters

- config – Pointer to DMA channel transfer configuration structure.
- srcStartAddr – source start address.
- dstStartAddr – destination start address.
- xferCfg – xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
- type – transfer type.
- trigger – DMA channel trigger configurations.
- nextDesc – address of next descriptor.

status_t DMA_SubmitTransfer(*dma_handle_t* *handle, *dma_transfer_config_t* *config)

Submits the DMA transfer request.

Deprecated:

Do not use this function. It has been superceded by DMA_SubmitChannelTransfer.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.

Return values

- kStatus_DMA_Success – It means submit transfer request succeed.
- kStatus_DMA_QueueFull – It means TCD queue is full. Submit transfer request is not allowed.
- kStatus_DMA_Busy – It means the given channel is busy, need to submit request later.

void DMA_SubmitChannelTransferParameter(*dma_handle_t* *handle, *uint32_t* xferCfg, *void* *srcStartAddr, *void* *dstStartAddr, *void* *nextDesc)

Submit channel transfer paramter directly.

This function used to configure channel head descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, it is useful for the case:

- for the single transfer, application doesn't need to allocate descriptor table, the head descriptor can be used for it.

```
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig,
↪intA, intB, width, srcInc, dstInc,
bytes), srcStartAddr, dstStartAddr, NULL);
DMA_StartTransfer(handle)
```

- for the linked transfer, application should responsible for link descriptor, for example, if 4 transfer is required, then application should prepare three descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[3]);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, NULL);
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig, ↪
↪ intA, intB, width, srcInc, dstInc,
bytes), srcStartAddr, dstStartAddr, nextDesc0);
DMA_StartTransfer(handle);
```

Parameters

- handle – Pointer to DMA handle.
- xferCfg – xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
- srcStartAddr – source start address.
- dstStartAddr – destination start address.
- nextDesc – address of next descriptor.

void DMA_SubmitChannelDescriptor(*dma_handle_t* *handle, *dma_descriptor_t* *descriptor)

Submit channel descriptor.

This function used to configure channel head descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, this function is typical for the ping pong case:

- a. for the ping pong case, application should responsible for the descriptor, for example, application should prepare two descriptor table with macro.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[2]);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc0);
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelDescriptor(handle, nextDesc0);
DMA_StartTransfer(handle);
```

Parameters

- handle – Pointer to DMA handle.
- descriptor – descriptor to submit.

status_t DMA_SubmitChannelTransfer(*dma_handle_t* *handle, *dma_channel_config_t* *config)

Submits the DMA channel transfer request.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time. It is used for the case:

- a. for the single transfer, application doesn't need to allocate descriptor table, the head descriptor can be used for it.

```
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,NULL);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

- b. for the linked transfer, application should responsible for link descriptor, for example, if 4 transfer is required, then application should prepare three descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);
DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, NULL);
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,
↪ nextDesc0);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

- c. for the ping pong case, application should responsible for link descriptor, for example, application should prepare two descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc0);
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,
↪ nextDesc0);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.

Return values

- `kStatus_DMA_Success` – It means submit transfer request succeed.
- `kStatus_DMA_QueueFull` – It means TCD queue is full. Submit transfer request is not allowed.
- `kStatus_DMA_Busy` – It means the given channel is busy, need to submit request later.

`void DMA_StartTransfer(dma_handle_t *handle)`

DMA start transfer.

This function enables the channel request. User can call this function after submitting the transfer request. It will trigger transfer start with software trigger only when hardware trigger is not used.

Parameters

- `handle` – DMA handle pointer.

`void DMA_IRQHandle(DMA_Type *base)`

DMA IRQ handler for descriptor transfer complete.

This function clears the channel major interrupt flag and call the callback function if it is not NULL.

Parameters

- `base` – DMA base address.

`FSL_DMA_DRIVER_VERSION`

DMA driver version.

Version 2.5.3.

`_dma_transfer_status` DMA transfer status

Values:

enumerator `kStatus_DMA_Busy`

Channel is busy and can't handle the transfer request.

`_dma_addr_interleave_size` dma address interleave size

Values:

enumerator `kDMA_AddressInterleave0xWidth`

dma source/destination address no interleave

enumerator `kDMA_AddressInterleave1xWidth`

dma source/destination address interleave 1xwidth

enumerator `kDMA_AddressInterleave2xWidth`

dma source/destination address interleave 2xwidth

enumerator `kDMA_AddressInterleave4xWidth`

dma source/destination address interleave 3xwidth

`_dma_transfer_width` dma transfer width

Values:

enumerator `kDMA_Transfer8BitWidth`

dma channel transfer bit width is 8 bit

enumerator kDMA__Transfer16BitWidth
dma channel transfer bit width is 16 bit

enumerator kDMA__Transfer32BitWidth
dma channel transfer bit width is 32 bit

enum __dma__priority
DMA channel priority.

Values:

enumerator kDMA__ChannelPriority0
Highest channel priority - priority 0

enumerator kDMA__ChannelPriority1
Channel priority 1

enumerator kDMA__ChannelPriority2
Channel priority 2

enumerator kDMA__ChannelPriority3
Channel priority 3

enumerator kDMA__ChannelPriority4
Channel priority 4

enumerator kDMA__ChannelPriority5
Channel priority 5

enumerator kDMA__ChannelPriority6
Channel priority 6

enumerator kDMA__ChannelPriority7
Lowest channel priority - priority 7

enum __dma__int
DMA interrupt flags.

Values:

enumerator kDMA__IntA
DMA interrupt flag A

enumerator kDMA__IntB
DMA interrupt flag B

enumerator kDMA__IntError
DMA interrupt flag error

enum __dma__trigger__type
DMA trigger type.

Values:

enumerator kDMA__NoTrigger
Trigger is disabled

enumerator kDMA__LowLevelTrigger
Low level active trigger

enumerator kDMA__HighLevelTrigger
High level active trigger

enumerator kDMA_FallingEdgeTrigger

Falling edge active trigger

enumerator kDMA_RisingEdgeTrigger

Rising edge active trigger

_dma_burst_size DMA burst size

Values:

enumerator kDMA_BurstSize1

burst size 1 transfer

enumerator kDMA_BurstSize2

burst size 2 transfer

enumerator kDMA_BurstSize4

burst size 4 transfer

enumerator kDMA_BurstSize8

burst size 8 transfer

enumerator kDMA_BurstSize16

burst size 16 transfer

enumerator kDMA_BurstSize32

burst size 32 transfer

enumerator kDMA_BurstSize64

burst size 64 transfer

enumerator kDMA_BurstSize128

burst size 128 transfer

enumerator kDMA_BurstSize256

burst size 256 transfer

enumerator kDMA_BurstSize512

burst size 512 transfer

enumerator kDMA_BurstSize1024

burst size 1024 transfer

enum _dma_trigger_burst

DMA trigger burst.

Values:

enumerator kDMA_SingleTransfer

Single transfer

enumerator kDMA_LevelBurstTransfer

Burst transfer driven by level trigger

enumerator kDMA_EdgeBurstTransfer1

Perform 1 transfer by edge trigger

enumerator kDMA_EdgeBurstTransfer2

Perform 2 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer4

Perform 4 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer8
Perform 8 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer16
Perform 16 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer32
Perform 32 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer64
Perform 64 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer128
Perform 128 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer256
Perform 256 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer512
Perform 512 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer1024
Perform 1024 transfers by edge trigger

enum _dma_burst_wrap
DMA burst wrapping.

Values:

enumerator kDMA_NoWrap
Wrapping is disabled

enumerator kDMA_SrcWrap
Wrapping is enabled for source

enumerator kDMA_DstWrap
Wrapping is enabled for destination

enumerator kDMA_SrcAndDstWrap
Wrapping is enabled for source and destination

enum _dma_transfer_type
DMA transfer type.

Values:

enumerator kDMA_MemoryToMemory
Transfer from memory to memory (increment source and destination)

enumerator kDMA_PeripheralToMemory
Transfer from peripheral to memory (increment only destination)

enumerator kDMA_MemoryToPeripheral
Transfer from memory to peripheral (increment only source)

enumerator kDMA_StaticToStatic
Peripheral to static memory (do not increment source or destination)

typedef struct *_dma_descriptor* dma_descriptor_t
DMA descriptor structure.

typedef struct *_dma_xfercfg* dma_xfercfg_t
DMA transfer configuration.

typedef enum *_dma_priority* dma_priority_t

DMA channel priority.

typedef enum *_dma_irq* dma_irq_t

DMA interrupt flags.

typedef enum *_dma_trigger_type* dma_trigger_type_t

DMA trigger type.

typedef enum *_dma_trigger_burst* dma_trigger_burst_t

DMA trigger burst.

typedef enum *_dma_burst_wrap* dma_burst_wrap_t

DMA burst wrapping.

typedef enum *_dma_transfer_type* dma_transfer_type_t

DMA transfer type.

typedef struct *_dma_channel_trigger* dma_channel_trigger_t

DMA channel trigger.

typedef struct *_dma_channel_config* dma_channel_config_t

DMA channel trigger.

typedef struct *_dma_transfer_config* dma_transfer_config_t

DMA transfer configuration.

typedef void (*dma_callback)(struct *_dma_handle* *handle, void *userData, bool transferDone, uint32_t intmode)

Define Callback function for DMA.

typedef struct *_dma_handle* dma_handle_t

DMA transfer handle structure.

DMA_MAX_TRANSFER_COUNT

DMA max transfer size.

FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE

DMA channel numbers.

DMA head link descriptor table align size

DMA_ALLOCATE_HEAD_DESCRIPTOR(name, number)

DMA head descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

DMA_ALLOCATE_HEAD_DESCRIPTOR_AT_NONCACHEABLE(name, number)

DMA head descriptor table allocate macro at noncacheable section To simplify user interface, this macro will help allocate descriptor memory at noncacheable section, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

DMA_ALLOCATE_LINK_DESCRIPTOR(name, number)

DMA link descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate descriptor name.
- number – Number of descriptor to be allocated.

DMA_ALLOCATE_LINK_DESCRIPTOR_AT_NONCACHEABLE(name, number)

DMA link descriptor table allocate macro at noncacheable section To simplify user interface, this macro will help allocate descriptor memory at noncacheable section, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate descriptor name.
- number – Number of descriptor to be allocated.

DMA_ALLOCATE_DATA_TRANSFER_BUFFER(name, width)

DMA transfer buffer address need to align with the transfer width.

DMA_CHANNEL_GROUP(channel)

DMA_CHANNEL_INDEX(base, channel)

DMA_COMMON_REG_GET(base, channel, reg)

DMA linked descriptor address align size.

DMA_COMMON_CONST_REG_GET(base, channel, reg)

DMA_COMMON_REG_SET(base, channel, reg, value)

DMA_DESCRIPTOR_END_ADDRESS(start, inc, bytes, width)

DMA descriptor end address calculate.

Parameters

- start – start address
- inc – address interleave size
- bytes – transfer bytes
- width – transfer width

DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width, srcInc, dstInc, bytes)

struct _dma_descriptor

#include <fsl_dma.h> DMA descriptor structure.

Public Members

volatile uint32_t xfercfg

Transfer configuration

void *srcEndAddr

Last source address of DMA transfer

void *dstEndAddr

Last destination address of DMA transfer

```
void *linkToNextDesc
    Address of next DMA descriptor in chain
struct __dma_xfercfg
    #include <fsl_dma.h> DMA transfer configuration.
```

Public Members

```
bool valid
    Descriptor is ready to transfer
bool reload
    Reload channel configuration register after current descriptor is exhausted
bool swtrig
    Perform software trigger. Transfer if fired when 'valid' is set
bool clrtrig
    Clear trigger
bool intA
    Raises IRQ when transfer is done and set IRQA status register flag
bool intB
    Raises IRQ when transfer is done and set IRQB status register flag
uint8_t byteWidth
    Byte width of data to transfer
uint8_t srcInc
    Increment source address by 'srcInc' x 'byteWidth'
uint8_t dstInc
    Increment destination address by 'dstInc' x 'byteWidth'
uint16_t transferCount
    Number of transfers
struct __dma_channel_trigger
    #include <fsl_dma.h> DMA channel trigger.
```

Public Members

```
dma_trigger_type_t type
    Select hardware trigger as edge triggered or level triggered.
dma_trigger_burst_t burst
    Select whether hardware triggers cause a single or burst transfer.
dma_burst_wrap_t wrap
    Select wrap type, source wrap or dest wrap, or both.
struct __dma_channel_config
    #include <fsl_dma.h> DMA channel trigger.
```

Public Members

`void *srcStartAddr`
Source data address

`void *dstStartAddr`
Destination data address

`void *nextDesc`
Chain custom descriptor

`uint32_t xferCfg`
channel transfer configurations

`dma_channel_trigger_t *trigger`
DMA trigger type

`bool isPeriph`
select the request type

`struct _dma_transfer_config`
#include <fsl_dma.h> DMA transfer configuration.

Public Members

`uint8_t *srcAddr`
Source data address

`uint8_t *dstAddr`
Destination data address

`uint8_t *nextDesc`
Chain custom descriptor

`dma_xfercfg_t xfercfg`
Transfer options

`bool isPeriph`
DMA transfer is driven by peripheral

`struct _dma_handle`
#include <fsl_dma.h> DMA transfer handle structure.

Public Members

`dma_callback callback`
Callback function. Invoked when transfer of descriptor with interrupt flag finishes

`void *userData`
Callback function parameter

`DMA_Type *base`
DMA peripheral base address

`uint8_t channel`
DMA channel number

2.10 DMIC: Digital Microphone

2.11 DMIC DMA Driver

```
status_t DMIC_TransferCreateHandleDMA(DMIC_Type *base, dmic_dma_handle_t *handle,  
                                     dmic_dma_transfer_callback_t callback, void  
                                     *userData, dma_handle_t *rxDmaHandle)
```

Initializes the DMIC handle which is used in transactional functions.

Parameters

- base – DMIC peripheral base address.
- handle – Pointer to `dmic_dma_handle_t` structure.
- callback – Callback function.
- userData – User data.
- rxDmaHandle – User-requested DMA handle for RX DMA transfer.

```
status_t DMIC_TransferReceiveDMA(DMIC_Type *base, dmic_dma_handle_t *handle,  
                                dmic_transfer_t *xfer, uint32_t channel)
```

Receives data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – USART peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- xfer – DMIC DMA transfer structure. See `dmic_transfer_t`.
- channel – DMIC start channel number.

Return values

`kStatus_Success` –

```
void DMIC_TransferAbortReceiveDMA(DMIC_Type *base, dmic_dma_handle_t *handle)
```

Aborts the received data using DMA.

This function aborts the received data using DMA.

Parameters

- base – DMIC peripheral base address
- handle – Pointer to `dmic_dma_handle_t` structure

```
status_t DMIC_TransferGetReceiveCountDMA(DMIC_Type *base, dmic_dma_handle_t *handle,  
                                         uint32_t *count)
```

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – DMIC peripheral base address.
- handle – DMIC handle pointer.
- count – Receive bytes count.

Return values

- `kStatus_NoTransferInProgress` – No receive in progress.

- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
void DMIC__InstallDMADescriptorMemory(dmic_dma_handle_t *handle, void *linkAddr, size_t linkNum)
```

Install DMA descriptor memory.

This function used to register DMA descriptor memory for linked transfer, a typical case is ping pong transfer which will request more than one DMA descriptor memory space, it should be called after DMIC_TransferCreateHandleDMA. User should be take care about the address of DMA descriptor pool which required align with 16BYTE at least.

Parameters

- handle – Pointer to DMA channel transfer handle.
- linkAddr – DMA link descriptor address.
- linkNum – DMA link descriptor number.

```
FSL_DMIC_DMA_DRIVER_VERSION
```

DMIC DMA driver version 2.4.1.

```
typedef struct _dmic_transfer dmic_transfer_t
```

DMIC transfer structure.

```
typedef struct _dmic_dma_handle dmic_dma_handle_t
```

```
typedef void (*dmic_dma_transfer_callback_t)(DMIC_Type *base, dmic_dma_handle_t *handle, status_t status, void *userData)
```

DMIC transfer callback function.

```
struct _dmic_transfer
```

#include <fsl_dmic_dma.h> DMIC transfer structure.

Public Members

```
void *data
```

The buffer of data to be transfer.

```
uint8_t dataWidth
```

DMIC support 16bit/32bit

```
size_t dataSize
```

The byte count to be transfer.

```
uint8_t dataAddrInterleaveSize
```

destination address interleave size

```
struct _dmic_transfer *linkTransfer
```

use to support link transfer

```
struct _dmic_dma_handle
```

#include <fsl_dmic_dma.h> DMIC DMA handle.

Public Members

```
DMIC_Type *base
```

DMIC peripheral base address.

dma_handle_t *rxDmaHandle
The DMA RX channel used.

dmic_dma_transfer_callback_t callback
Callback function.

void *userData
DMIC callback function parameter.

size_t transferSize
Size of the data to receive.

volatile uint8_t state
Internal state of DMIC DMA transfer

uint32_t channel
DMIC channel used.

bool isChannelValid
DMIC channel initialization flag

dma_descriptor_t *desLink
descriptor pool pointer

size_t linkNum
number of descriptor in descriptors pool

2.12 DMIC Driver

uint32_t DMIC_GetInstance(DMIC_Type *base)
Get the DMIC instance from peripheral base address.

Parameters

- base – DMIC peripheral base address.

Returns

DMIC instance.

void DMIC_Init(DMIC_Type *base)
Turns DMIC Clock on.

Parameters

- base – : DMIC base

Returns

Nothing

void DMIC_DeInit(DMIC_Type *base)
Turns DMIC Clock off.

Parameters

- base – : DMIC base

Returns

Nothing

void DMIC_SetOperationMode(DMIC_Type *base, *operation_mode_t* mode)
Set DMIC operating mode.

Deprecated:

Do not use this function. It has been superseded by `DMIC_EnableChannelInterrupt`, `DMIC_EnableChannelDma`.

Parameters

- `base` – : The base address of DMIC interface
- `mode` – : DMIC mode

Returns

Nothing

```
void DMIC_Use2fs(DMIC_Type *base, bool use2fs)
```

Configure Clock scaling.

Parameters

- `base` – : The base address of DMIC interface
- `use2fs` – : clock scaling

Returns

Nothing

```
void DMIC_CfgChannelDc(DMIC_Type *base, dmic_channel_t channel, dc_removal_t  
dc_cut_level, uint32_t post_dc_gain_reduce, bool saturate16bit)
```

Configure DMIC channel.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : DMIC channel
- `dc_cut_level` – : `dc_removal_t`, Cut off Frequency
- `post_dc_gain_reduce` – : Fine gain adjustment in the form of a number of bits to downshift.
- `saturate16bit` – : If selects 16-bit saturation.

```
static inline void DMIC_EnableChannelSignExtend(DMIC_Type *base, dmic_channel_t channel,  
bool enable)
```

Enable channel sign extend which allows processing of 24bit audio data on 32bit machines.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : DMIC channel
- `enable` – : true is enable sign extend, false is disable sign extend

```
void DMIC_ConfigChannel(DMIC_Type *base, dmic_channel_t channel, stereo_side_t side,  
dmic_channel_config_t *channel_config)
```

Configure DMIC channel.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : DMIC channel
- `side` – : `stereo_side_t`, choice of left or right
- `channel_config` – : Channel configuration

Returns

Nothing

```
void DMIC__EnableChannnel(DMIC_Type *base, uint32_t channelmask)
```

Enable a particulr channel.

Parameters

- `base` – : The base address of DMIC interface
- `channelmask` – reference `_dmic_channel_mask`

Returns

Nothing

```
void DMIC__FifoChannel(DMIC_Type *base, uint32_t channel, uint32_t trig_level, uint32_t  
enable, uint32_t resetn)
```

Configure fifo settings for DMIC channel.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : DMIC channel
- `trig_level` – : FIFO trigger level
- `enable` – : FIFO level
- `resetn` – : FIFO reset

Returns

Nothing

```
static inline void DMIC__EnableChannelInterrupt(DMIC_Type *base, dmic_channel_t channel,  
bool enable)
```

Enable a particulr channel interrupt request.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : Channel selection
- `enable` – : true is enable, false is disable

```
static inline void DMIC__EnableChannelDma(DMIC_Type *base, dmic_channel_t channel, bool  
enable)
```

Enable a particulr channel dma request.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : Channel selection
- `enable` – : true is enable, false is disable

```
static inline void DMIC__EnableChannelFifo(DMIC_Type *base, dmic_channel_t channel, bool  
enable)
```

Enable a particulr channel fifo.

Parameters

- `base` – : The base address of DMIC interface
- `channel` – : Channel selection
- `enable` – : true is enable, false is disable

static inline void DMIC_DoFifoReset(DMIC_Type *base, *dmic_channel_t* channel)
Channel fifo reset.

Parameters

- base – : The base address of DMIC interface
- channel – : Channel selection

static inline uint32_t DMIC_FifoGetStatus(DMIC_Type *base, uint32_t channel)
Get FIFO status.

Parameters

- base – : The base address of DMIC interface
- channel – : DMIC channel

Returns

FIFO status

static inline void DMIC_FifoClearStatus(DMIC_Type *base, uint32_t channel, uint32_t mask)
Clear FIFO status.

Parameters

- base – : The base address of DMIC interface
- channel – : DMIC channel
- mask – : Bits to be cleared

Returns

FIFO status

static inline uint32_t DMIC_FifoGetData(DMIC_Type *base, uint32_t channel)
Get FIFO data.

Parameters

- base – : The base address of DMIC interface
- channel – : DMIC channel

Returns

FIFO data

static inline uint32_t DMIC_FifoGetAddress(DMIC_Type *base, uint32_t channel)
Get FIFO address.

Parameters

- base – : The base address of DMIC interface
- channel – : DMIC channel

Returns

FIFO data

void DMIC_ResetChannelDecimator(DMIC_Type *base, uint32_t channelMask, bool reset)
DMIC channel Decimator reset.

Parameters

- base – : The base address of DMIC interface
- channelMask – : DMIC channel mask, reference `_dmic_channel_mask`
- reset – : true is reset decimator, false is release decimator.

```
static inline void DMIC__EnableChannelGlobalSync(DMIC_Type *base, uint32_t channelMask,
uint32_t syncCounter)
```

Enable DMIC channel global sync function.

Parameters

- base – : The base address of DMIC interface
- channelMask – : DMIC channel mask, reference `_dmic_channel_mask`
- syncCounter – : sync counter will trigger a pulse whenever count reaches `CCOUNTVAL`. If `CCOUNTVAL` is set to 0, there will be a pulse on every cycle

```
static inline void DMIC__DisableChannelGlobalSync(DMIC_Type *base, uint32_t channelMask)
```

Disbale DMIC channel global sync function.

Parameters

- base – : The base address of DMIC interface
- channelMask – : DMIC channel mask, reference `_dmic_channel_mask`

```
void DMIC__EnableIntCallback(DMIC_Type *base, dmic_callback_t cb)
```

Enable callback.

This function enables the interrupt for the selected DMIC peripheral. The callback function is not enabled until this function is called.

Parameters

- base – Base address of the DMIC peripheral.
- cb – callback Pointer to store callback function.

Return values

None. –

```
void DMIC__DisableIntCallback(DMIC_Type *base, dmic_callback_t cb)
```

Disable callback.

This function disables the interrupt for the selected DMIC peripheral.

Parameters

- base – Base address of the DMIC peripheral.
- cb – callback Pointer to store callback function..

Return values

None. –

```
static inline void DMIC__SetGainNoiseEstHwvad(DMIC_Type *base, uint32_t value)
```

Sets the gain value for the noise estimator.

Parameters

- base – DMIC base pointer
- value – gain value for the noise estimator.

Return values

None. –

```
static inline void DMIC__SetGainSignalEstHwvad(DMIC_Type *base, uint32_t value)
```

Sets the gain value for the signal estimator.

Parameters

- base – DMIC base pointer
- value – gain value for the signal estimator.

Return values

None. –

static inline void DMIC_SetFilterCtrlHwvad(DMIC_Type *base, uint32_t value)

Sets the hwvad filter cutoff frequency parameter.

Parameters

- base – DMIC base pointer
- value – cut off frequency value.

Return values

None. –

static inline void DMIC_SetInputGainHwvad(DMIC_Type *base, uint32_t value)

Sets the input gain of hwvad.

Parameters

- base – DMIC base pointer
- value – input gain value for hwvad.

Return values

None. –

static inline void DMIC_CtrlClrIntrHwvad(DMIC_Type *base, bool st10)

Clears hwvad internal interrupt flag.

Parameters

- base – DMIC base pointer
- st10 – bit value.

Return values

None. –

static inline void DMIC_FilterResetHwvad(DMIC_Type *base, bool rstt)

Resets hwvad filters.

Parameters

- base – DMIC base pointer
- rstt – Reset bit value.

Return values

None. –

static inline uint16_t DMIC_GetNoiseEnvlpEst(DMIC_Type *base)

Gets the value from output of the filter z7.

Parameters

- base – DMIC base pointer

Return values

output – of filter z7.

void DMIC_HwvadEnableIntCallback(DMIC_Type *base, *dmic_hwvad_callback_t* vadcb)

Enable hwvad callback.

This function enables the hwvad interrupt for the selected DMIC peripheral. The callback function is not enabled until this function is called.

Parameters

- base – Base address of the DMIC peripheral.

- `vadcb` – callback Pointer to store callback function.

Return values

None. –

`void DMIC_HwvadDisableIntCallback(DMIC_Type *base, dmic_hwvad_callback_t vadcb)`

Disable callback.

This function disables the hwvad interrupt for the selected DMIC peripheral.

Parameters

- `base` – Base address of the DMIC peripheral.
- `vadcb` – callback Pointer to store callback function..

Return values

None. –

`FSL_DMIC_DRIVER_VERSION`

DMIC driver version 2.3.3.

`_dmic_status` DMIC transfer status.

Values:

enumerator `kStatus_DMIC_Busy`

DMIC is busy

enumerator `kStatus_DMIC_Idle`

DMIC is idle

enumerator `kStatus_DMIC_OverRunError`

DMIC over run Error

enumerator `kStatus_DMIC_UnderRunError`

DMIC under run Error

enum `_operation_mode`

DMIC different operation modes.

Values:

enumerator `kDMIC_OperationModeInterrupt`

Interrupt mode

enumerator `kDMIC_OperationModeDma`

DMA mode

enum `_stereo_side`

DMIC left/right values.

Values:

enumerator `kDMIC_Left`

Left Stereo channel

enumerator `kDMIC_Right`

Right Stereo channel

enum `pdm_div_t`

DMIC Clock pre-divider values.

Values:

enumerator kDMIC_PdmDiv1

DMIC pre-divider set in divide by 1

enumerator kDMIC_PdmDiv2

DMIC pre-divider set in divide by 2

enumerator kDMIC_PdmDiv3

DMIC pre-divider set in divide by 3

enumerator kDMIC_PdmDiv4

DMIC pre-divider set in divide by 4

enumerator kDMIC_PdmDiv6

DMIC pre-divider set in divide by 6

enumerator kDMIC_PdmDiv8

DMIC pre-divider set in divide by 8

enumerator kDMIC_PdmDiv12

DMIC pre-divider set in divide by 12

enumerator kDMIC_PdmDiv16

DMIC pre-divider set in divide by 16

enumerator kDMIC_PdmDiv24

DMIC pre-divider set in divide by 24

enumerator kDMIC_PdmDiv32

DMIC pre-divider set in divide by 32

enumerator kDMIC_PdmDiv48

DMIC pre-divider set in divide by 48

enumerator kDMIC_PdmDiv64

DMIC pre-divider set in divide by 64

enumerator kDMIC_PdmDiv96

DMIC pre-divider set in divide by 96

enumerator kDMIC_PdmDiv128

DMIC pre-divider set in divide by 128

enum _compensation

Pre-emphasis Filter coefficient value for 2FS and 4FS modes.

Values:

enumerator kDMIC_CompValueZero

Compensation 0

enumerator kDMIC_CompValueNegativePoint16

Compensation -0.16

enumerator kDMIC_CompValueNegativePoint15

Compensation -0.15

enumerator kDMIC_CompValueNegativePoint13

Compensation -0.13

enum _dc_removal

DMIC DC filter control values.

Values:

enumerator kDMIC_DcNoRemove

Flat response no filter

enumerator kDMIC_DcCut155

Cut off Frequency is 155 Hz

enumerator kDMIC_DcCut78

Cut off Frequency is 78 Hz

enumerator kDMIC_DcCut39

Cut off Frequency is 39 Hz

enum _dmic_channel

DMIC Channel number.

Values:

enumerator kDMIC_Channel0

DMIC channel 0

enumerator kDMIC_Channel1

DMIC channel 1

enumerator kDMIC_ChannelMAX

Maximum number of DMIC channels

_dmic_channel_mask DMIC Channel mask.

Values:

enumerator kDMIC_EnableChannel0

DMIC channel 0 mask

enumerator kDMIC_EnableChannel1

DMIC channel 1 mask

enum _dmic_phy_sample_rate

DMIC and decimator sample rates.

Values:

enumerator kDMIC_PhyFullSpeed

Decimator gets one sample per each chosen clock edge of PDM interface

enumerator kDMIC_PhyHalfSpeed

PDM clock to Microphone is halved, decimator receives each sample twice

typedef enum _operation_mode operation_mode_t

DMIC different operation modes.

typedef enum _stereo_side stereo_side_t

DMIC left/right values.

typedef enum _compensation compensation_t

Pre-emphasis Filter coefficient value for 2FS and 4FS modes.

typedef enum _dc_removal dc_removal_t

DMIC DC filter control values.

typedef enum _dmic_channel dmic_channel_t

DMIC Channel number.

```
typedef enum _dmic_phy_sample_rate dmic_phy_sample_rate_t
```

DMIC and decimator sample rates.

```
typedef struct _dmic_channel_config dmic_channel_config_t
```

DMIC Channel configuration structure.

```
typedef void (*dmic_callback_t)(void)
```

DMIC Callback function.

```
typedef void (*dmic_hwvad_callback_t)(void)
```

HWVAD Callback function.

```
struct _dmic_channel_config
```

#include <fsl_dmic.h> DMIC Channel configuration structure.

Public Members

pdm_div_t divhfclk

DMIC Clock pre-divider values

uint32_t osr

oversampling rate(CIC decimation rate) for PCM

uint32_t gainshft

4FS PCM data gain control

compensation_t preac2coef

Pre-emphasis Filter coefficient value for 2FS

compensation_t preac4coef

Pre-emphasis Filter coefficient value for 4FS

dc_removal_t dc_cut_level

DMIC DC filter control values.

uint32_t post_dc_gain_reduce

Fine gain adjustment in the form of a number of bits to downshift

dmic_phy_sample_rate_t sample_rate

DMIC and decimator sample rates

bool saturate16bit

Selects 16-bit saturation. 0 means results roll over if out range and do not saturate. 1 means if the result overflows, it saturates at 0xFFFF for positive overflow and 0x8000 for negative overflow.

bool enableSignExtend

sign extend feature which allows processing of 24bit audio data on 32bit machine

2.13 ENET: Ethernet MAC Driver

```
void ENET_GetDefaultConfig(enet_config_t *config)
```

Gets the ENET default configuration structure.

The purpose of this API is to get the default ENET MAC controller configure structure for ENET_Init(). User may use the initialized structure unchanged in ENET_Init(), or modify some fields of the structure before calling ENET_Init(). Example:

```
enet_config_t config;  
ENET_GetDefaultConfig(&config);
```

Parameters

- config – The ENET mac controller configuration structure pointer.

status_t ENET_Up(ENET_Type *base, *enet_handle_t* *handle, const *enet_config_t* *config, const *enet_buffer_config_t* *bufferConfig, uint8_t *macAddr, uint32_t srcClock_Hz)

Initializes the ENET module.

This function initializes the module with the ENET configuration.

Note: ENET has two buffer descriptors legacy buffer descriptors and enhanced IEEE 1588 buffer descriptors. The legacy descriptor is used by default. To use the IEEE 1588 feature, use the enhanced IEEE 1588 buffer descriptor by defining “ENET_ENHANCEDBUFFERDESCRIPTOR_MODE” and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling ENET_Up().

Parameters

- base – ENET peripheral base address.
- handle – ENET handler pointer.
- config – ENET mac configuration structure pointer. The “enet_config_t” type mac configuration return from ENET_GetDefaultConfig can be used directly. It is also possible to verify the Mac configuration using other methods.
- bufferConfig – ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization. It is the start address of “ringNum” *enet_buffer_config* structures. To support added multi-ring features in some soc and compatible with the previous enet driver version. For single ring supported, this bufferConfig is a buffer configure structure pointer; for multi-ring supported and used case, this bufferConfig pointer should be a buffer configure structure array pointer.
- macAddr – ENET mac address of Ethernet device. This MAC address should be provided.
- srcClock_Hz – The internal module clock source for MII clock.

Return values

- kStatus_Success – Succeed to initialize the ethernet driver.
- kStatus_ENET_InitMemoryFail – Init fails since buffer memory is not enough.

status_t ENET_Init(ENET_Type *base, *enet_handle_t* *handle, const *enet_config_t* *config, const *enet_buffer_config_t* *bufferConfig, uint8_t *macAddr, uint32_t srcClock_Hz)

Initializes the ENET module.

This function ungates the module clock and initializes it with the ENET configuration.

Note: ENET has two buffer descriptors legacy buffer descriptors and enhanced IEEE 1588 buffer descriptors. The legacy descriptor is used by default. To use the IEEE 1588 feature, use the enhanced IEEE 1588 buffer descriptor by defining “ENET_ENHANCEDBUFFERDESCRIPTOR_MODE” and calling ENET_Ptp1588Configure() to configure the 1588 feature and related buffers after calling ENET_Init().

Parameters

- `base` – ENET peripheral base address.
- `handle` – ENET handler pointer.
- `config` – ENET mac configuration structure pointer. The “enet_config_t” type mac configuration return from `ENET_GetDefaultConfig` can be used directly. It is also possible to verify the Mac configuration using other methods.
- `bufferConfig` – ENET buffer configuration structure pointer. The buffer configuration should be prepared for ENET Initialization. It is the start address of “ringNum” `enet_buffer_config` structures. To support added multi-ring features in some soc and compatible with the previous enet driver version. For single ring supported, this `bufferConfig` is a buffer configure structure pointer; for multi-ring supported and used case, this `bufferConfig` pointer should be a buffer configure structure array pointer.
- `macAddr` – ENET mac address of Ethernet device. This MAC address should be provided.
- `srcClock_Hz` – The internal module clock source for MII clock.

Return values

- `kStatus_Success` – Succeed to initialize the ethernet driver.
- `kStatus_ENET_InitMemoryFail` – Init fails since buffer memory is not enough.

`void ENET_Down(ENET_Type *base)`

Stops the ENET module.

This function disables the ENET module.

Parameters

- `base` – ENET peripheral base address.

`void ENET_Deinit(ENET_Type *base)`

Deinitializes the ENET module.

This function gates the module clock, clears ENET interrupts, and disables the ENET module.

Parameters

- `base` – ENET peripheral base address.

`static inline void ENET_Reset(ENET_Type *base)`

Resets the ENET module.

This function restores the ENET module to reset state. Note that this function sets all registers to reset state. As a result, the ENET module can't work after calling this function.

Parameters

- `base` – ENET peripheral base address.

`void ENET_ResetHardware(void)`

Resets the ENET hardware.

This function resets ENET related resources in the hardware.

`void ENET_SetMII(ENET_Type *base, enet_mii_speed_t speed, enet_mii_duplex_t duplex)`

Sets the ENET MII speed and duplex.

This API is provided to dynamically change the speed and duplex for MAC.

Parameters

- base – ENET peripheral base address.
- speed – The speed of the RMII mode.
- duplex – The duplex of the RMII mode.

void ENET_SetSMI(ENET_Type *base, uint32_t srcClock_Hz, bool isPreambleDisabled)

Sets the ENET SMI(serial management interface)- MII management interface.

Parameters

- base – ENET peripheral base address.
- srcClock_Hz – This is the ENET module clock frequency. See clock distribution.
- isPreambleDisabled – The preamble disable flag.
 - true Enables the preamble.
 - false Disables the preamble.

static inline bool ENET_GetSMI(ENET_Type *base)

Gets the ENET SMI- MII management interface configuration.

This API is used to get the SMI configuration to check whether the MII management interface has been set.

Parameters

- base – ENET peripheral base address.

Returns

The SMI setup status true or false.

static inline uint32_t ENET_ReadSMIData(ENET_Type *base)

Reads data from the PHY register through an SMI interface.

Parameters

- base – ENET peripheral base address.

Returns

The data read from PHY

static inline void ENET_StartSMIWrite(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, *enet_mii_write_t* operation, uint16_t data)

Sends the MDIO IEEE802.3 Clause 22 format write command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOWrite() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- phyAddr – The PHY address. Range from 0 ~ 31.
- regAddr – The PHY register address. Range from 0 ~ 31.
- operation – The write operation.
- data – The data written to PHY.

```
static inline void ENET_StartSMIRRead(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr,
                                     enet_mii_read_t operation)
```

Sends the MDIO IEEE802.3 Clause 22 format read command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated `ENET_MDIORead()` can be called. For customized requirements, implement with combining separated APIs.

Parameters

- `base` – ENET peripheral base address.
- `phyAddr` – The PHY address. Range from 0 ~ 31.
- `regAddr` – The PHY register address. Range from 0 ~ 31.
- `operation` – The read operation.

```
status_t ENET_MDIOWrite(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t data)
MDIO write with IEEE802.3 Clause 22 format.
```

Parameters

- `base` – ENET peripheral base address.
- `phyAddr` – The PHY address. Range from 0 ~ 31.
- `regAddr` – The PHY register. Range from 0 ~ 31.
- `data` – The data written to PHY.

Returns

`kStatus_Success` MDIO access succeeds.

Returns

`kStatus_Timeout` MDIO access timeout.

```
status_t ENET_MDIORead(ENET_Type *base, uint8_t phyAddr, uint8_t regAddr, uint16_t
                        *pData)
```

MDIO read with IEEE802.3 Clause 22 format.

Parameters

- `base` – ENET peripheral base address.
- `phyAddr` – The PHY address. Range from 0 ~ 31.
- `regAddr` – The PHY register. Range from 0 ~ 31.
- `pData` – The data read from PHY.

Returns

`kStatus_Success` MDIO access succeeds.

Returns

`kStatus_Timeout` MDIO access timeout.

```
static inline void ENET_StartExtC45SMIWriteReg(ENET_Type *base, uint8_t portAddr, uint8_t
                                              devAddr, uint16_t regAddr)
```

Sends the MDIO IEEE802.3 Clause 45 format write register command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated `ENET_MDIOC45Write()/ENET_MDIOC45Read()` can be called. For customized requirements, implement with combining separated APIs.

Parameters

- `base` – ENET peripheral base address.

- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- regAddr – The PHY register address.

```
static inline void ENET_StartExtC45SMIWriteData(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t data)
```

Sends the MDIO IEEE802.3 Clause 45 format write data command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOC45Write() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- data – The data written to PHY.

```
static inline void ENET_StartExtC45SMIReadData(ENET_Type *base, uint8_t portAddr, uint8_t devAddr)
```

Sends the MDIO IEEE802.3 Clause 45 format read data command.

After calling this function, need to check whether the transmission is over then do next MDIO operation. For ease of use, encapsulated ENET_MDIOC45Read() can be called. For customized requirements, implement with combining separated APIs.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.

```
status_t ENET_MDIOC45Write(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t regAddr, uint16_t data)
```

MDIO write with IEEE802.3 Clause 45 format.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).
- devAddr – The device address.
- regAddr – The PHY register address.
- data – The data written to PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

```
status_t ENET_MDIOC45Read(ENET_Type *base, uint8_t portAddr, uint8_t devAddr, uint16_t regAddr, uint16_t *pData)
```

MDIO read with IEEE802.3 Clause 45 format.

Parameters

- base – ENET peripheral base address.
- portAddr – The MDIO port address(PHY address).

- devAddr – The device address.
- regAddr – The PHY register address.
- pData – The data read from PHY.

Returns

kStatus_Success MDIO access succeeds.

Returns

kStatus_Timeout MDIO access timeout.

```
static inline void ENET_SetRGMIIClockDelay(ENET_Type *base, bool txEnabled, bool rxEnabled)
```

Control the usage of the delayed tx/rx RGMII clock.

Parameters

- base – ENET peripheral base address.
- txEnabled – Enable or disable to generate the delayed version of RGMII_TXC.
- rxEnabled – Enable or disable to use the delayed version of RGMII_RXC.

```
void ENET_SetMacAddr(ENET_Type *base, uint8_t *macAddr)
```

Sets the ENET module Mac address.

Parameters

- base – ENET peripheral base address.
- macAddr – The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

```
void ENET_GetMacAddr(ENET_Type *base, uint8_t *macAddr)
```

Gets the ENET module Mac address.

Parameters

- base – ENET peripheral base address.
- macAddr – The six-byte Mac address pointer. The pointer is allocated by application and input into the API.

```
void ENET_AddMulticastGroup(ENET_Type *base, uint8_t *address)
```

Adds the ENET device to a multicast group.

Parameters

- base – ENET peripheral base address.
- address – The six-byte multicast group address which is provided by application.

```
void ENET_LeaveMulticastGroup(ENET_Type *base, uint8_t *address)
```

Moves the ENET device from a multicast group.

Parameters

- base – ENET peripheral base address.
- address – The six-byte multicast group address which is provided by application.

```
static inline void ENET_ActiveRead(ENET_Type *base)
```

Activates frame reception for multiple rings.

This function is to active the enet read process.

Note: This must be called after the MAC configuration and state are ready. It must be called after the ENET_Init(). This should be called when the frame reception is required.

Parameters

- base – ENET peripheral base address.

static inline void ENET_EnableSleepMode(ENET_Type *base, bool enable)

Enables/disables the MAC to enter sleep mode. This function is used to set the MAC enter sleep mode. When entering sleep mode, the magic frame wakeup interrupt should be enabled to wake up MAC from the sleep mode and reset it to normal mode.

Parameters

- base – ENET peripheral base address.
- enable – True enable sleep mode, false disable sleep mode.

static inline void ENET_GetAccelFunction(ENET_Type *base, uint32_t *txAccelOption, uint32_t *rxAccelOption)

Gets ENET transmit and receive accelerator functions from MAC controller.

Parameters

- base – ENET peripheral base address.
- txAccelOption – The transmit accelerator option. The “enet_tx_accelerator_t” is recommended to be used to as the mask to get the exact the accelerator option.
- rxAccelOption – The receive accelerator option. The “enet_rx_accelerator_t” is recommended to be used to as the mask to get the exact the accelerator option.

static inline void ENET_EnableInterrupts(ENET_Type *base, uint32_t mask)

Enables the ENET interrupt.

This function enables the ENET interrupt according to the provided mask. The mask is a logical OR of enumeration members. See enet_interrupt_enable_t. For example, to enable the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_EnableInterrupts(ENET, kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
```

Parameters

- base – ENET peripheral base address.
- mask – ENET interrupts to enable. This is a logical OR of the enumeration enet_interrupt_enable_t.

static inline void ENET_DisableInterrupts(ENET_Type *base, uint32_t mask)

Disables the ENET interrupt.

This function disables the ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See enet_interrupt_enable_t. For example, to disable the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_DisableInterrupts(ENET, kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
```

Parameters

- base – ENET peripheral base address.

- `mask` – ENET interrupts to disable. This is a logical OR of the enumeration `enet_interrupt_enable_t`.

static inline uint32_t ENET_GetInterruptStatus(ENET_Type *base)

Gets the ENET interrupt status flag.

Parameters

- `base` – ENET peripheral base address.

Returns

The event status of the interrupt source. This is the logical OR of members of the enumeration `enet_interrupt_enable_t`.

static inline void ENET_ClearInterruptStatus(ENET_Type *base, uint32_t mask)

Clears the ENET interrupt events status flag.

This function clears enabled ENET interrupts according to the provided mask. The mask is a logical OR of enumeration members. See the `enet_interrupt_enable_t`. For example, to clear the TX frame interrupt and RX frame interrupt, do the following.

```
ENET_ClearInterruptStatus(ENET, kENET_TxFrameInterrupt | kENET_RxFrameInterrupt);
```

Parameters

- `base` – ENET peripheral base address.
- `mask` – ENET interrupt source to be cleared. This is the logical OR of members of the enumeration `enet_interrupt_enable_t`.

void ENET_SetRxISRHandler(ENET_Type *base, *enet_isr_t* ISRHandler)

Set the second level Rx IRQ handler.

Parameters

- `base` – ENET peripheral base address.
- `ISRHandler` – The handler to install.

void ENET_SetTxISRHandler(ENET_Type *base, *enet_isr_t* ISRHandler)

Set the second level Tx IRQ handler.

Parameters

- `base` – ENET peripheral base address.
- `ISRHandler` – The handler to install.

void ENET_SetErrISRHandler(ENET_Type *base, *enet_isr_t* ISRHandler)

Set the second level Err IRQ handler.

Parameters

- `base` – ENET peripheral base address.
- `ISRHandler` – The handler to install.

void ENET_GetRxErrBeforeReadFrame(*enet_handle_t* *handle, *enet_data_error_stats_t* *eErrorStatic, uint8_t ringId)

Gets the error statistics of a received frame for ENET specified ring.

This API must be called after the `ENET_GetRxFramSize` and before the `ENET_ReadFrame()`. If the `ENET_GetRxFramSize` returns `kStatus_ENET_RxFrameError`, the `ENET_GetRxErrBeforeReadFrame` can be used to get the exact error statistics. This is an example.

```
status = ENET_GetRxFrameSize(&g_handle, &length, 0);
if (status == kStatus_ENET_RxFrameError)
{
    Comments: Get the error information of the received frame.
    ENET_GetRxErrBeforeReadFrame(&g_handle, &eErrStatic, 0);
    Comments: update the receive buffer.
    ENET_ReadFrame(EXAMPLE_ENET, &g_handle, NULL, 0);
}
```

Parameters

- handle – The ENET handler structure pointer. This is the same handler pointer used in the ENET_Init.
- eErrorStatic – The error statistics structure pointer.
- ringId – The ring index, range from 0 ~ (FSL_FEATURE_ENET_INSTANCE_QUEUEEn(x) - 1).

void ENET_EnableStatistics(ENET_Type *base, bool enable)

Enables/disables collection of transfer statistics.

Note that this function does not reset any of the already collected data, use the function ENET_ResetStatistics to clear the transfer statistics if needed.

Parameters

- base – ENET peripheral base address.
- enable – True enable statistics collection, false disable statistics collection.

void ENET_GetStatistics(ENET_Type *base, *enet_transfer_stats_t* *statistics)

Gets transfer statistics.

Copies the actual value of hardware counters into the provided structure. Calling this function does not reset the counters in hardware.

Parameters

- base – ENET peripheral base address.
- statistics – The statistics structure pointer.

void ENET_ResetStatistics(ENET_Type *base)

Resets transfer statistics.

Sets the value of hardware transfer counters to zero.

Parameters

- base – ENET peripheral base address.

status_t ENET_GetRxFrameSize(*enet_handle_t* *handle, uint32_t *length, uint8_t ringId)

Gets the size of the read frame for specified ring.

This function gets a received frame size from the ENET buffer descriptors.

Note: The FCS of the frame is automatically removed by MAC and the size is the length without the FCS. After calling ENET_GetRxFrameSize, ENET_ReadFrame() should be called to receive frame and update the BD if the result is not “kStatus_ENET_RxFrameEmpty”.

Parameters

- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.

- length – The length of the valid frame received.
- ringId – The ring index or ring number.

Return values

- kStatus_ENET_RxFrameEmpty – No frame received. Should not call ENET_ReadFrame to read frame.
- kStatus_ENET_RxFrameError – Data error happens. ENET_ReadFrame should be called with NULL data and NULL length to update the receive buffers.
- kStatus_Success – Receive a frame Successfully then the ENET_ReadFrame should be called with the right data buffer and the captured data length input.

status_t ENET_ReadFrame(ENET_Type *base, *enet_handle_t* *handle, uint8_t *data, uint32_t length, uint8_t ringId, uint32_t *ts)

Reads a frame from the ENET device. This function reads a frame (both the data and the length) from the ENET buffer descriptors. User can get timestamp through ts pointer if the ts is not NULL.

Note: It doesn't store the timestamp in the receive timestamp queue. The ENET_GetRxFrameSize should be used to get the size of the prepared data buffer. This API uses memcpy to copy data from DMA buffer to application buffer, 4 bytes aligned data buffer in 32 bits platforms provided by user may let compiler use optimization instruction to reduce time consumption. This is an example:

```
uint32_t length;
enet_handle_t g_handle;
Comments: Get the received frame size firstly.
status = ENET_GetRxFrameSize(&g_handle, &length, 0);
if (length != 0)
{
    Comments: Allocate memory here with the size of "length"
    uint8_t *data = memory allocate interface;
    if (!data)
    {
        ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0, NULL);
        Comments: Add the console warning log.
    }
    else
    {
        status = ENET_ReadFrame(ENET, &g_handle, data, length, 0, NULL);
        Comments: Call stack input API to deliver the data to stack
    }
}
else if (status == kStatus_ENET_RxFrameError)
{
    Comments: Update the received buffer when a error frame is received.
    ENET_ReadFrame(ENET, &g_handle, NULL, 0, 0, NULL);
}
```

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler structure. This is the same handler pointer used in the ENET_Init.
- data – The data buffer provided by user to store the frame which memory size should be at least "length".

- `length` – The size of the data buffer which is still the length of the received frame.
- `ringId` – The ring index or ring number.
- `ts` – The timestamp address to store received timestamp.

Returns

The execute status, successful or failure.

`status_t` ENET_SendFrame(ENET_Type *base, *enet_handle_t* *handle, const uint8_t *data, uint32_t length, uint8_t ringId, bool tsFlag, void *context)

Transmits an ENET frame for specified ring.

Note: The CRC is automatically appended to the data. Input the data to send without the CRC. This API uses memcpy to copy data from DMA buffer to application buffer, 4 bytes aligned data buffer in 32 bits platforms provided by user may let compiler use optimization instruction to reduce time consumption.

Parameters

- `base` – ENET peripheral base address.
- `handle` – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- `data` – The data buffer provided by user to send.
- `length` – The length of the data to send.
- `ringId` – The ring index or ring number.
- `tsFlag` – Timestamp enable flag.
- `context` – Used by user to handle some events after transmit over.

Return values

- `kStatus_Success` – Send frame succeed.
- `kStatus_ENET_TxFrameBusy` – Transmit buffer descriptor is busy under transmission. The transmit busy happens when the data send rate is over the MAC capacity. The waiting mechanism is recommended to be added after each call return with `kStatus_ENET_TxFrameBusy`.

`status_t` ENET_SetTxReclaim(*enet_handle_t* *handle, bool isEnabled, uint8_t ringId)

Enable or disable tx descriptors reclaim mechanism.

Note: This function must be called when no pending send frame action. Set enable if you want to reclaim context or timestamp in interrupt.

Parameters

- `handle` – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- `isEnabled` – Enable or disable flag.
- `ringId` – The ring index or ring number.

Return values

- `kStatus_Success` – Succeed to enable/disable Tx reclaim.
- `kStatus_Fail` – Fail to enable/disable Tx reclaim.

`void ENET_ReclaimTxDescriptor(ENET_Type *base, enet_handle_t *handle, uint8_t ringId)`

Reclaim tx descriptors. This function is used to update the tx descriptor status and store the tx timestamp when the 1588 feature is enabled. This is called by the transmit interrupt IRQ handler after the complete of a frame transmission.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- ringId – The ring index or ring number.

`status_t ENET_GetRxFrame(ENET_Type *base, enet_handle_t *handle, enet_rx_frame_struct_t *rxFrame, uint8_t ringId)`

Receives one frame in specified BD ring with zero copy.

This function uses the user-defined allocation and free callbacks. Every time application gets one frame through this function, driver stores the buffer address(es) in `enet_buffer_struct_t` and allocate new buffer(s) for the BD(s). If there's no memory buffer in the pool, this function drops current one frame to keep the Rx frame in BD ring is as fresh as possible.

Note: Application must provide a memory pool including at least BD number + n buffers in order for this function to work properly, because each BD must always take one buffer while driver is running, then other extra n buffer(s) can be taken by application. Here n is the `ceil(max_frame_length(set by RCR) / bd_rx_size(set by MRBR))`. Application must also provide an array structure in `rxFrame->rxBuffArray` with n index to receive one complete frame in any case.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- rxFrame – The received frame information structure provided by user.
- ringId – The ring index or ring number.

Return values

- `kStatus_Success` – Succeed to get one frame and allocate new memory for Rx buffer.
- `kStatus_ENET_RxFrameEmpty` – There's no Rx frame in the BD.
- `kStatus_ENET_RxFrameError` – There's issue in this receiving.
- `kStatus_ENET_RxFrameDrop` – There's no new buffer memory for BD, drop this frame.

`status_t ENET_StartTxFrame(ENET_Type *base, enet_handle_t *handle, enet_tx_frame_struct_t *txFrame, uint8_t ringId)`

Sends one frame in specified BD ring with zero copy.

This function supports scattered buffer transmit, user needs to provide the buffer array.

Note: Tx reclaim should be enabled to ensure the Tx buffer ownership can be given back to application after Tx is over.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer. This is the same handler pointer used in the ENET_Init.
- txFrame – The Tx frame structure.
- ringId – The ring index or ring number.

Return values

- kStatus_Success – Succeed to send one frame.
- kStatus_ENET_TxFrameBusy – The BD is not ready for Tx or the reclaim operation still not finishes.
- kStatus_ENET_TxFrameOverLen – The Tx frame length is over max ethernet frame length.

void ENET_TransmitIRQHandler(ENET_Type *base, *enet_handle_t* *handle)

The transmit IRQ handler.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer.

void ENET_ReceiveIRQHandler(ENET_Type *base, *enet_handle_t* *handle)

The receive IRQ handler.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer.

void ENET_ErrorIRQHandler(ENET_Type *base, *enet_handle_t* *handle)

Some special IRQ handler including the error, mii, wakeup irq handler.

Parameters

- base – ENET peripheral base address.
- handle – The ENET handler pointer.

void ENET_Ptp1588IRQHandler(ENET_Type *base)

the common IRQ handler for the 1588 irq handler.

This is used for the 1588 timer interrupt.

Parameters

- base – ENET peripheral base address.

void ENET_CommonFrame0IRQHandler(ENET_Type *base)

the common IRQ handler for the tx/rx/error etc irq handler.

This is used for the combined tx/rx/error interrupt for single/mutli-ring (frame 0).

Parameters

- base – ENET peripheral base address.

FSL_ENET_DRIVER_VERSION

Defines the driver version.

ENET_BUFFDESCRIPTOR_RX_EMPTY_MASK

Empty bit mask.

ENET_BUFFDESCRIPTOR_RX_SOFTOWNER1_MASK

Software owner one mask.

ENET_BUFFDESCRIPTOR_RX_WRAP_MASK

Next buffer descriptor is the start address.

ENET_BUFFDESCRIPTOR_RX_SOFTOWNER2_Mask

Software owner two mask.

ENET_BUFFDESCRIPTOR_RX_LAST_MASK

Last BD of the frame mask.

ENET_BUFFDESCRIPTOR_RX_MISS_MASK

Received because of the promiscuous mode.

ENET_BUFFDESCRIPTOR_RX_BROADCAST_MASK

Broadcast packet mask.

ENET_BUFFDESCRIPTOR_RX_MULTICAST_MASK

Multicast packet mask.

ENET_BUFFDESCRIPTOR_RX_LENVIOLATE_MASK

Length violation mask.

ENET_BUFFDESCRIPTOR_RX_NOOCTET_MASK

Non-octet aligned frame mask.

ENET_BUFFDESCRIPTOR_RX_CRC_MASK

CRC error mask.

ENET_BUFFDESCRIPTOR_RX_OVERRUN_MASK

FIFO overrun mask.

ENET_BUFFDESCRIPTOR_RX_TRUNC_MASK

Frame is truncated mask.

ENET_BUFFDESCRIPTOR_TX_READY_MASK

Ready bit mask.

ENET_BUFFDESCRIPTOR_TX_SOFTOWENER1_MASK

Software owner one mask.

ENET_BUFFDESCRIPTOR_TX_WRAP_MASK

Wrap buffer descriptor mask.

ENET_BUFFDESCRIPTOR_TX_SOFTOWENER2_MASK

Software owner two mask.

ENET_BUFFDESCRIPTOR_TX_LAST_MASK

Last BD of the frame mask.

ENET_BUFFDESCRIPTOR_TX_TRANMITCRC_MASK

Transmit CRC mask.

ENET_FRAME_MAX_FRAMELEN

Default maximum Ethernet frame size without VLAN tag.

ENET_FRAME_VLAN_TAGLEN

Ethernet single VLAN tag size.

ENET_FRAME_CRC_LEN

CRC size in a frame.

ENET_FRAME_TX_LEN_LIMITATION(x)

ENET_FIFO_MIN_RX_FULL

ENET minimum receive FIFO full.

ENET_RX_MIN_BUFFERSIZE

ENET minimum buffer size.

ENET_PHY_MAXADDRESS

Maximum PHY address.

ENET_TX_INTERRUPT

Enet Tx interrupt flag.

ENET_RX_INTERRUPT

Enet Rx interrupt flag.

ENET_TS_INTERRUPT

Enet timestamp interrupt flag.

ENET_ERR_INTERRUPT

Enet error interrupt flag.

Defines the status return codes for transaction.

Values:

enumerator kStatus_ENET_InitMemoryFail

Init fails since buffer memory is not enough.

enumerator kStatus_ENET_RxFrameError

A frame received but data error happen.

enumerator kStatus_ENET_RxFrameFail

Failed to receive a frame.

enumerator kStatus_ENET_RxFrameEmpty

No frame arrive.

enumerator kStatus_ENET_RxFrameDrop

Rx frame is dropped since no buffer memory.

enumerator kStatus_ENET_TxFrameOverLen

Tx frame over length.

enumerator kStatus_ENET_TxFrameBusy

Tx buffer descriptors are under process.

enumerator kStatus_ENET_TxFrameFail

Transmit frame fail.

enum _enet_mii_mode

Defines the MII/RMII/RGMII mode for data interface between the MAC and the PHY.

Values:

enumerator kENET_MiiMode

MII mode for data interface.

enumerator kENET_RmiiMode

RMII mode for data interface.

enumerator kENET_RgmiiMode
RGMII mode for data interface.

enum _enet_mii_speed
Defines the 10/100/1000 Mbps speed for the MII data interface.
Notice: “kENET_MiiSpeed1000M” only supported when mii mode is “kENET_RgmiiMode”.
Values:

enumerator kENET_MiiSpeed10M
Speed 10 Mbps.

enumerator kENET_MiiSpeed100M
Speed 100 Mbps.

enumerator kENET_MiiSpeed1000M
Speed 1000M bps.

enum _enet_mii_duplex
Defines the half or full duplex for the MII data interface.
Values:

enumerator kENET_MiiHalfDuplex
Half duplex mode.

enumerator kENET_MiiFullDuplex
Full duplex mode.

enum _enet_mii_write
Define the MII opcode for normal MDIO_CLAUSES_22 Frame.
Values:

enumerator kENET_MiiWriteNoCompliant
Write frame operation, but not MII-compliant.

enumerator kENET_MiiWriteValidFrame
Write frame operation for a valid MII management frame.

enum _enet_mii_read
Defines the read operation for the MII management frame.
Values:

enumerator kENET_MiiReadValidFrame
Read frame operation for a valid MII management frame.

enumerator kENET_MiiReadNoCompliant
Read frame operation, but not MII-compliant.

enum _enet_mii_extend_opcode
Define the MII opcode for extended MDIO_CLAUSES_45 Frame.
Values:

enumerator kENET_MiiAddrWrite_C45
Address Write operation.

enumerator kENET_MiiWriteFrame_C45
Write frame operation for a valid MII management frame.

enumerator kENET_MiiReadFrame_C45
Read frame operation for a valid MII management frame.

enum _enet_special_control_flag

Defines a special configuration for ENET MAC controller.

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to mac-SpecialConfig in the enet_config_t. The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifo-FullThreshold and txFifoWatermark in the enet_config_t.

Values:

enumerator kENET_ControlFlowControlEnable

Enable ENET flow control: pause frame.

enumerator kENET_ControlRxPayloadCheckEnable

Enable ENET receive payload length check.

enumerator kENET_ControlRxPadRemoveEnable

Padding is removed from received frames.

enumerator kENET_ControlRxBroadCastRejectEnable

Enable broadcast frame reject.

enumerator kENET_ControlMacAddrInsert

Enable MAC address insert.

enumerator kENET_ControlStoreAndFwdDisable

Enable FIFO store and forward.

enumerator kENET_ControlSMIPreambleDisable

Enable SMI preamble.

enumerator kENET_ControlPromiscuousEnable

Enable promiscuous mode.

enumerator kENET_ControlMIILoopEnable

Enable ENET MII loop back.

enumerator kENET_ControlVLANTagEnable

Enable normal VLAN (single vlan tag).

enumerator kENET_ControlSVLANEnable

Enable S-VLAN.

enumerator kENET_ControlVLANUseSecondTag

Enable extracting the second vlan tag for further processing.

enum _enet_interrupt_enable

List of interrupts supported by the peripheral. This enumeration uses one-bit encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

Values:

enumerator kENET_BabrInterrupt

Babbling receive error interrupt source

enumerator kENET_BabtInterrupt

Babbling transmit error interrupt source

enumerator kENET_GraceStopInterrupt

Graceful stop complete interrupt source

enumerator kENET_TxFrameInterrupt
TX FRAME interrupt source

enumerator kENET_TxBufferInterrupt
TX BUFFER interrupt source

enumerator kENET_RxFrameInterrupt
RX FRAME interrupt source

enumerator kENET_RxBufferInterrupt
RX BUFFER interrupt source

enumerator kENET_MiiInterrupt
MII interrupt source

enumerator kENET_EBusERInterrupt
Ethernet bus error interrupt source

enumerator kENET_LateCollisionInterrupt
Late collision interrupt source

enumerator kENET_RetryLimitInterrupt
Collision Retry Limit interrupt source

enumerator kENET_UnderrunInterrupt
Transmit FIFO underrun interrupt source

enumerator kENET_PayloadRxInterrupt
Payload Receive error interrupt source

enumerator kENET_WakeupInterrupt
WAKEUP interrupt source

enumerator kENET_TsAvailInterrupt
TS AVAIL interrupt source for PTP

enumerator kENET_TsTimerInterrupt
TS WRAP interrupt source for PTP

enum _enet_event

Defines the common interrupt event for callback use.

Values:

enumerator kENET_RxEvent
Receive event.

enumerator kENET_TxEvent
Transmit event.

enumerator kENET_ErrEvent
Error event: BABR/BABT/EBERR/LC/RL/UN/PLR .

enumerator kENET_WakeUpEvent
Wake up from sleep mode event.

enumerator kENET_TimeStampEvent
Time stamp event.

enumerator kENET_TimeStampAvailEvent
Time stamp available event.

enum _enet_idle_slope

Defines certain idle slope for bandwidth fraction.

Values:

enumerator kENET_IdleSlope1

The bandwidth fraction is about 0.002.

enumerator kENET_IdleSlope2

The bandwidth fraction is about 0.003.

enumerator kENET_IdleSlope4

The bandwidth fraction is about 0.008.

enumerator kENET_IdleSlope8

The bandwidth fraction is about 0.02.

enumerator kENET_IdleSlope16

The bandwidth fraction is about 0.03.

enumerator kENET_IdleSlope32

The bandwidth fraction is about 0.06.

enumerator kENET_IdleSlope64

The bandwidth fraction is about 0.11.

enumerator kENET_IdleSlope128

The bandwidth fraction is about 0.20.

enumerator kENET_IdleSlope256

The bandwidth fraction is about 0.33.

enumerator kENET_IdleSlope384

The bandwidth fraction is about 0.43.

enumerator kENET_IdleSlope512

The bandwidth fraction is about 0.50.

enumerator kENET_IdleSlope640

The bandwidth fraction is about 0.56.

enumerator kENET_IdleSlope768

The bandwidth fraction is about 0.60.

enumerator kENET_IdleSlope896

The bandwidth fraction is about 0.64.

enumerator kENET_IdleSlope1024

The bandwidth fraction is about 0.67.

enumerator kENET_IdleSlope1152

The bandwidth fraction is about 0.69.

enumerator kENET_IdleSlope1280

The bandwidth fraction is about 0.71.

enumerator kENET_IdleSlope1408

The bandwidth fraction is about 0.73.

enumerator kENET_IdleSlope1536

The bandwidth fraction is about 0.75.

enum `_enet_tx_accelerator`

Defines the transmit accelerator configuration.

Note that the hardware does not insert ICMPv6 protocol checksums as mentioned in errata ERR052152.

Values:

enumerator `kENET_TxAccelIsShift16Enabled`

Transmit FIFO shift-16.

enumerator `kENET_TxAccelIpCheckEnabled`

Insert IP header checksum.

enumerator `kENET_TxAccelProtoCheckEnabled`

Insert protocol checksum (TCP, UDP, ICMPv4).

enum `_enet_rx_accelerator`

Defines the receive accelerator configuration.

Note that the hardware does not validate ICMPv6 protocol checksums as mentioned in errata ERR052152.

Values:

enumerator `kENET_RxAccelPadRemoveEnabled`

Padding removal for short IP frames.

enumerator `kENET_RxAccelIpCheckEnabled`

Discard with wrong IP header checksum.

enumerator `kENET_RxAccelProtoCheckEnabled`

Discard with wrong protocol checksum (TCP, UDP, ICMPv4).

enumerator `kENET_RxAccelMacCheckEnabled`

Discard with Mac layer errors.

enumerator `kENET_RxAccelIsShift16Enabled`

Receive FIFO shift-16.

typedef enum `_enet_mii_mode` `enet_mii_mode_t`

Defines the MII/RMII/RGMII mode for data interface between the MAC and the PHY.

typedef enum `_enet_mii_speed` `enet_mii_speed_t`

Defines the 10/100/1000 Mbps speed for the MII data interface.

Notice: “`kENET_MiiSpeed1000M`” only supported when mii mode is “`kENET_RgmiiMode`”.

typedef enum `_enet_mii_duplex` `enet_mii_duplex_t`

Defines the half or full duplex for the MII data interface.

typedef enum `_enet_mii_write` `enet_mii_write_t`

Define the MII opcode for normal MDIO_CLAUSES_22 Frame.

typedef enum `_enet_mii_read` `enet_mii_read_t`

Defines the read operation for the MII management frame.

typedef enum `_enet_mii_extend_opcode` `enet_mii_extend_opcode`

Define the MII opcode for extended MDIO_CLAUSES_45 Frame.

typedef enum *_enet_special_control_flag* enet_special_control_flag_t

Defines a special configuration for ENET MAC controller.

These control flags are provided for special user requirements. Normally, these control flags are unused for ENET initialization. For special requirements, set the flags to mac-SpecialConfig in the enet_config_t. The kENET_ControlStoreAndFwdDisable is used to disable the FIFO store and forward. FIFO store and forward means that the FIFO read/send is started when a complete frame is stored in TX/RX FIFO. If this flag is set, configure rxFifo-FullThreshold and txFifoWatermark in the enet_config_t.

typedef enum *_enet_interrupt_enable* enet_interrupt_enable_t

List of interrupts supported by the peripheral. This enumeration uses one-bit encoding to allow a logical OR of multiple members. Members usually map to interrupt enable bits in one or more peripheral registers.

typedef enum *_enet_event* enet_event_t

Defines the common interrupt event for callback use.

typedef enum *_enet_idle_slope* enet_idle_slope_t

Defines certain idle slope for bandwidth fraction.

typedef enum *_enet_tx_accelerator* enet_tx_accelerator_t

Defines the transmit accelerator configuration.

Note that the hardware does not insert ICMPv6 protocol checksums as mentioned in errata ERR052152.

typedef enum *_enet_rx_accelerator* enet_rx_accelerator_t

Defines the receive accelerator configuration.

Note that the hardware does not validate ICMPv6 protocol checksums as mentioned in errata ERR052152.

typedef struct *_enet_rx_bd_struct* enet_rx_bd_struct_t

Defines the receive buffer descriptor structure for the little endian system.

typedef struct *_enet_tx_bd_struct* enet_tx_bd_struct_t

Defines the enhanced transmit buffer descriptor structure for the little endian system.

typedef struct *_enet_data_error_stats* enet_data_error_stats_t

Defines the ENET data error statistics structure.

typedef struct *_enet_rx_frame_error* enet_rx_frame_error_t

Defines the Rx frame error structure.

typedef struct *_enet_transfer_stats* enet_transfer_stats_t

Defines the ENET transfer statistics structure.

typedef struct *enet_frame_info* enet_frame_info_t

Defines the frame info structure.

typedef struct *_enet_tx_dirty_ring* enet_tx_dirty_ring_t

Defines the ENET transmit dirty addresses ring/queue structure.

typedef void (*enet_rx_alloc_callback_t)(ENET_Type *base, void *userData, uint8_t ringId)

Defines the ENET Rx memory buffer alloc function pointer.

typedef void (*enet_rx_free_callback_t)(ENET_Type *base, void *buffer, void *userData, uint8_t ringId)

Defines the ENET Rx memory buffer free function pointer.

```
typedef struct _enet_buffer_config enet_buffer_config_t
```

Defines the receive buffer descriptor configuration structure.

Note that for the internal DMA requirements, the buffers have a corresponding alignment requirements.

- a. The aligned receive and transmit buffer size must be evenly divisible by ENET_BUFF_ALIGNMENT. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of "ENET_BUFF_ALIGNMENT" and the cache line size.
- b. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned. However, it's recommended to be evenly divisible by ENET_BUFF_ALIGNMENT. buffer descriptors should be put in non-cacheable region when cache is enabled.
- c. The aligned transmit and receive data buffer start address must be evenly divisible by ENET_BUFF_ALIGNMENT. Receive buffers should be continuous with the total size equal to "rxBdNumber * rxBuffSizeAlign". Transmit buffers should be continuous with the total size equal to "txBdNumber * txBuffSizeAlign". when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of "ENET_BUFF_ALIGNMENT" and the cache line size.

```
typedef struct _enet_intcoalesce_config enet_intcoalesce_config_t
```

Defines the interrupt coalescing configure structure.

```
typedef struct _enet_avb_config enet_avb_config_t
```

Defines the ENET AVB Configure structure.

This is used for to configure the extended ring 1 and ring 2.

- a. The classification match format is $(CMP3 \ll 12) \mid (CMP2 \ll 8) \mid (CMP1 \ll 4) \mid CMP0$. composed of four 3-bit compared VLAN priority field cmp0~cmp3, cm0 ~ cmp3 are used in parallel.

If CMP1,2,3 are not unused, please set them to the same value as CMP0.

- a. The idleSlope is used to calculate the Band Width fraction, BW fraction = $1 / (1 + 512/idleSlope)$. For avb configuration, the BW fraction of Class 1 and Class 2 combined must not exceed 0.75.

```
typedef struct _enet_handle enet_handle_t
```

```
typedef void (*enet_callback_t)(ENET_Type *base, enet_handle_t *handle, enet_event_t event,
enet_frame_info_t *frameInfo, void *userData)
```

ENET callback function.

```
typedef struct _enet_config enet_config_t
```

Defines the basic configuration structure for the ENET device.

Note:

- a. macSpecialConfig is used for a special control configuration, A logical OR of "enet_special_control_flag_t". For a special configuration for MAC, set this parameter to 0.
- b. txWatermark is used for a cut-through operation. It is in steps of 64 bytes: 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO 3 - 192 bytes written to TX FIFO The maximum of txWatermark is 0x2F - 4032 bytes written to TX FIFO txWatermark allows minimizing the transmit latency to set the txWatermark to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
- c. rxFifoFullThreshold is similar to the txWatermark for cut-through operation in RX. It is in 64-bit words. The minimum is ENET_FIFO_MIN_RX_FULL and the maximum is

0xFF. If the end of the frame is stored in FIFO and the frame size is smaller than the txWatermark, the frame is still transmitted. The rule is the same for rxFifoFullThreshold in the receive direction.

- d. When “kENET_ControlFlowControlEnable” is set in the macSpecialConfig, ensure that the pauseDuration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.
- e. When “kENET_ControlStoreAndFwdDisabled” is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
- f. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The “enet_tx_accelerator_t” and “enet_rx_accelerator_t” are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.
- g. The intCoalesceCfg can be used in the rx or tx enabled cases to decrease the CPU loading.

```
typedef struct _enet_tx_bd_ring enet_tx_bd_ring_t
```

Defines the ENET transmit buffer descriptor ring/queue structure.

```
typedef struct _enet_rx_bd_ring enet_rx_bd_ring_t
```

Defines the ENET receive buffer descriptor ring/queue structure.

```
typedef struct _enet_buffer_struct enet_buffer_struct_t
```

```
typedef struct _enet_rx_frame_attribute_struct enet_rx_frame_attribute_t
```

```
typedef struct _enet_rx_frame_struct enet_rx_frame_struct_t
```

```
typedef struct _enet_tx_frame_struct enet_tx_frame_struct_t
```

```
typedef void (*enet_isr_t)(ENET_Type *base, enet_handle_t *handle)
```

Define interrupt IRQ handler.

```
const clock_ip_name_t s_enetClock[]
```

Pointers to enet clocks for each instance.

```
const clock_ip_name_t s_enetExtraClock[]
```

```
uint32_t ENET_GetInstance(ENET_Type *base)
```

Get the ENET instance from peripheral base address.

Parameters

- base – ENET peripheral base address.

Returns

ENET instance.

```
ENET_BUFFDESCRIPTOR_RX_ERR_MASK
```

Defines the receive error status flag mask.

```
struct _enet_rx_bd_struct
```

#include <fsl_enet.h> Defines the receive buffer descriptor structure for the little endian system.

Public Members

```
uint16_t length
```

Buffer descriptor data length.

uint16_t control

Buffer descriptor control and status.

uint32_t buffer

Data buffer pointer.

struct __enet_tx_bd_struct

#include <fsl_enet.h> Defines the enhanced transmit buffer descriptor structure for the little endian system.

Public Members

uint16_t length

Buffer descriptor data length.

uint16_t control

Buffer descriptor control and status.

uint32_t buffer

Data buffer pointer.

struct __enet_data_error_stats

#include <fsl_enet.h> Defines the ENET data error statistics structure.

Public Members

uint32_t statsRxLenGreaterErr

Receive length greater than RCR[MAX_FL].

uint32_t statsRxAlignErr

Receive non-octet alignment/

uint32_t statsRxFcsErr

Receive CRC error.

uint32_t statsRxOverRunErr

Receive over run.

uint32_t statsRxTruncateErr

Receive truncate.

struct __enet_rx_frame_error

#include <fsl_enet.h> Defines the Rx frame error structure.

Public Members

bool statsRxTruncateErr

Receive truncate.

bool statsRxOverRunErr

Receive over run.

bool statsRxFcsErr

Receive CRC error.

bool statsRxAlignErr

Receive non-octet alignment.

bool statsRxLenGreaterErr
Receive length greater than RCR[MAX_FL].

struct __enet__transfer_stats
#include <fsl_enet.h> Defines the ENET transfer statistics structure.

Public Members

uint32_t statsRxFrameCount
Rx frame number.

uint32_t statsRxFrameOk
Good Rx frame number.

uint32_t statsRxCrcErr
Rx frame number with CRC error.

uint32_t statsRxAlignErr
Rx frame number with alignment error.

uint32_t statsRxDropInvalidSFD
Dropped frame number due to invalid SFD.

uint32_t statsRxFifoOverflowErr
Rx FIFO overflow count.

uint32_t statsTxFrameCount
Tx frame number.

uint32_t statsTxFrameOk
Good Tx frame number.

uint32_t statsTxCrcAlignErr
The transmit frame is error.

uint32_t statsTxFifoUnderRunErr
Tx FIFO underrun count.

struct enet__frame__info
#include <fsl_enet.h> Defines the frame info structure.

Public Members

void *context
User specified data

struct __enet__tx__dirty__ring
#include <fsl_enet.h> Defines the ENET transmit dirty addresses ring/queue structure.

Public Members

enet_frame_info_t *txDirtyBase
Dirty buffer descriptor base address pointer.

uint16_t txGenIdx
tx generate index.

uint16_t txConsumIdx
tx consume index.

uint16_t txRingLen

tx ring length.

bool isFull

tx ring is full flag.

struct `_enet_buffer_config`

`#include <fsl_enet.h>` Defines the receive buffer descriptor configuration structure.

Note that for the internal DMA requirements, the buffers have a corresponding alignment requirements.

- a. The aligned receive and transmit buffer size must be evenly divisible by `ENET_BUFF_ALIGNMENT`. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of “`ENET_BUFF_ALIGNMENT`” and the cache line size.
- b. The aligned transmit and receive buffer descriptor start address must be at least 64 bit aligned. However, it's recommended to be evenly divisible by `ENET_BUFF_ALIGNMENT`. buffer descriptors should be put in non-cacheable region when cache is enabled.
- c. The aligned transmit and receive data buffer start address must be evenly divisible by `ENET_BUFF_ALIGNMENT`. Receive buffers should be continuous with the total size equal to “`rxBdNumber * rxBuffSizeAlign`”. Transmit buffers should be continuous with the total size equal to “`txBdNumber * txBuffSizeAlign`”. when the data buffers are in cacheable region when cache is enabled, all those size should be aligned to the maximum value of “`ENET_BUFF_ALIGNMENT`” and the cache line size.

Public Members

uint16_t rxBdNumber

Receive buffer descriptor number.

uint16_t txBdNumber

Transmit buffer descriptor number.

uint16_t rxBuffSizeAlign

Aligned receive data buffer size.

uint16_t txBuffSizeAlign

Aligned transmit data buffer size.

volatile `enet_rx_bd_struct_t` *rxBdStartAddrAlign

Aligned receive buffer descriptor start address: should be non-cacheable.

volatile `enet_tx_bd_struct_t` *txBdStartAddrAlign

Aligned transmit buffer descriptor start address: should be non-cacheable.

uint8_t *rxBufferAlign

Receive data buffer start address.

uint8_t *txBufferAlign

Transmit data buffer start address.

bool rxMaintainEnable

Receive buffer cache maintain.

bool txMaintainEnable

Transmit buffer cache maintain.

enet_frame_info_t *txFrameInfo

Transmit frame information start address.

struct __enet_intcoalesce_config

#include <fsl_enet.h> Defines the interrupt coalescing configure structure.

Public Members

uint8_t txCoalesceFrameCount[1]

Transmit interrupt coalescing frame count threshold.

uint16_t txCoalesceTimeCount[1]

Transmit interrupt coalescing timer count threshold.

uint8_t rxCoalesceFrameCount[1]

Receive interrupt coalescing frame count threshold.

uint16_t rxCoalesceTimeCount[1]

Receive interrupt coalescing timer count threshold.

struct __enet_avb_config

#include <fsl_enet.h> Defines the ENET AVB Configure structure.

This is used for to configure the extended ring 1 and ring 2.

- a. The classification match format is $(CMP3 \ll 12) \mid (CMP2 \ll 8) \mid (CMP1 \ll 4) \mid CMP0$. composed of four 3-bit compared VLAN priority field cmp0~cmp3, cm0 ~ cmp3 are used in parallel.

If CMP1,2,3 are not unused, please set them to the same value as CMP0.

- a. The idleSlope is used to calculate the Band Width fraction, $BW \text{ fraction} = 1 / (1 + 512/idleSlope)$. For avb configuration, the BW fraction of Class 1 and Class 2 combined must not exceed 0.75.

Public Members

uint16_t rxClassifyMatch[1 - 1]

The classification match value for the ring.

enet_idle_slope_t idleSlope[1 - 1]

The idle slope for certian bandwidth fraction.

struct __enet_config

#include <fsl_enet.h> Defines the basic configuration structure for the ENET device.

Note:

- a. macSpecialConfig is used for a special control configuration, A logical OR of “enet_special_control_flag_t”. For a special configuration for MAC, set this parameter to 0.
- b. txWatermark is used for a cut-through operation. It is in steps of 64 bytes: 0/1 - 64 bytes written to TX FIFO before transmission of a frame begins. 2 - 128 bytes written to TX FIFO 3 - 192 bytes written to TX FIFO The maximum of txWatermark is 0x2F - 4032 bytes written to TX FIFO txWatermark allows minimizing the transmit latency to set the txWatermark to 0 or 1 or for larger bus access latency 3 or larger due to contention for the system bus.
- c. rxFifoFullThreshold is similar to the txWatermark for cut-through operation in RX. It is in 64-bit words. The minimum is ENET_FIFO_MIN_RX_FULLL and the maximum is 0xFF. If the end of the frame is stored in FIFO and the frame size if smaller than the

txWatermark, the frame is still transmitted. The rule is the same for rxFifoFullThreshold in the receive direction.

- d. When “kENET_ControlFlowControlEnable” is set in the macSpecialConfig, ensure that the pauseDuration, rxFifoEmptyThreshold, and rxFifoStatEmptyThreshold are set for flow control enabled case.
- e. When “kENET_ControlStoreAndFwdDisabled” is set in the macSpecialConfig, ensure that the rxFifoFullThreshold and txFifoWatermark are set for store and forward disable.
- f. The rxAccelerConfig and txAccelerConfig default setting with 0 - accelerator are disabled. The “enet_tx_accelerator_t” and “enet_rx_accelerator_t” are recommended to be used to enable the transmit and receive accelerator. After the accelerators are enabled, the store and forward feature should be enabled. As a result, kENET_ControlStoreAndFwdDisabled should not be set.
- g. The intCoalesceCfg can be used in the rx or tx enabled cases to decrease the CPU loading.

Public Members

uint32_t macSpecialConfig

Mac special configuration. A logical OR of “enet_special_control_flag_t”.

uint32_t interrupt

Mac interrupt source. A logical OR of “enet_interrupt_enable_t”.

uint16_t rxMaxFrameLen

Receive maximum frame length.

enet_mii_mode_t miiMode

MII mode.

enet_mii_speed_t miiSpeed

MII Speed.

enet_mii_duplex_t miiDuplex

MII duplex.

uint8_t rxAccelerConfig

Receive accelerator, A logical OR of “enet_rx_accelerator_t”.

uint8_t txAccelerConfig

Transmit accelerator, A logical OR of “enet_tx_accelerator_t”.

uint16_t pauseDuration

For flow control enabled case: Pause duration.

uint8_t rxFifoEmptyThreshold

For flow control enabled case: when RX FIFO level reaches this value, it makes MAC generate XOFF pause frame.

uint8_t rxFifoStatEmptyThreshold

For flow control enabled case: number of frames in the receive FIFO, independent of size, that can be accept. If the limit is reached, reception continues and a pause frame is triggered.

uint8_t rxFifoFullThreshold

For store and forward disable case, the data required in RX FIFO to notify the MAC receive ready status.

uint8_t txFifoWatermark

For store and forward disable case, the data required in TX FIFO before a frame transmit start.

enet_intcoalesce_config_t *intCoalesceCfg

If the interrupt coalesce is not required in the ring n(0,1,2), please set to NULL.

uint8_t ringNum

Number of used rings. default with 1 — single ring.

enet_rx_alloc_callback_t rxBuffAlloc

Callback function to alloc memory, must be provided for zero-copy Rx.

enet_rx_free_callback_t rxBuffFree

Callback function to free memory, must be provided for zero-copy Rx.

enet_callback_t callback

General callback function.

void *userData

Callback function parameter.

struct __enet_tx_bd_ring

#include <fsl_enet.h> Defines the ENET transmit buffer descriptor ring/queue structure.

Public Members

volatile enet_tx_bd_struct_t *txBdBase

Buffer descriptor base address pointer.

uint16_t txGenIdx

The current available transmit buffer descriptor pointer.

uint16_t txConsumeIdx

Transmit consume index.

volatile uint16_t txDescUsed

Transmit descriptor used number.

uint16_t txRingLen

Transmit ring length.

struct __enet_rx_bd_ring

#include <fsl_enet.h> Defines the ENET receive buffer descriptor ring/queue structure.

Public Members

volatile enet_rx_bd_struct_t *rxBdBase

Buffer descriptor base address pointer.

uint16_t rxGenIdx

The current available receive buffer descriptor pointer.

uint16_t rxRingLen

Receive ring length.

struct __enet_handle

#include <fsl_enet.h> Defines the ENET handler structure.

Public Members

```

enet_rx_bd_ring_t rxBdRing[1]
    Receive buffer descriptor.
enet_tx_bd_ring_t txBdRing[1]
    Transmit buffer descriptor.
uint16_t rxBuffSizeAlign[1]
    Receive buffer size alignment.
uint16_t txBuffSizeAlign[1]
    Transmit buffer size alignment.
bool rxMaintainEnable[1]
    Receive buffer cache maintain.
bool txMaintainEnable[1]
    Transmit buffer cache maintain.
uint8_t ringNum
    Number of used rings.
enet_callback_t callback
    Callback function.
void *userData
    Callback function parameter.
enet_tx_dirty_ring_t txDirtyRing[1]
    Ring to store tx frame information.
bool txReclaimEnable[1]
    Tx reclaim enable flag.
enet_rx_alloc_callback_t rxBuffAlloc
    Callback function to alloc memory for zero copy Rx.
enet_rx_free_callback_t rxBuffFree
    Callback function to free memory for zero copy Rx.
uint8_t multicastCount[64]
    Multicast collisions counter
struct __enet_buffer_struct
#include <fsl_enet.h>

```

Public Members

```

void *buffer
    The buffer store the whole or partial frame.
uint16_t length
    The byte length of this buffer.
struct __enet_rx_frame_attribute_struct
#include <fsl_enet.h>

```

Public Members

bool promiscuous

This frame is received because of promiscuous mode.

```
struct __enet_rx_frame_struct  
#include <fsl_enet.h>
```

Public Members

enet_buffer_struct_t *rxBuffArray
Rx frame buffer structure.

uint16_t totLen
Rx frame total length.

enet_rx_frame_attribute_t rxAttribute
Rx frame attribute structure.

enet_rx_frame_error_t rxFrameError
Rx frame error.

```
struct __enet_tx_frame_struct  
#include <fsl_enet.h>
```

Public Members

enet_buffer_struct_t *txBuffArray
Tx frame buffer structure.

uint32_t txBuffNum
Buffer number of this Tx frame.

void *context
Driver reclaims and gives it in Tx over callback, usually store network packet header.

2.14 FLEXCOMM: FLEXCOMM Driver

2.15 FLEXCOMM Driver

FSL_FLEXCOMM_DRIVER_VERSION
FlexCOMM driver version 2.0.2.

enum FLEXCOMM_PERIPH_T
FLEXCOMM peripheral modes.

Values:

enumerator FLEXCOMM_PERIPH_NONE
No peripheral

enumerator FLEXCOMM_PERIPH_USART
USART peripheral

enumerator FLEXCOMM_PERIPH_SPI
SPI Peripheral

```

enumerator FLEXCOMM_PERIPH_I2C
    I2C Peripheral
enumerator FLEXCOMM_PERIPH_I2S_TX
    I2S TX Peripheral
enumerator FLEXCOMM_PERIPH_I2S_RX
    I2S RX Peripheral
typedef void (*flexcomm_irq_handler_t)(void *base, void *handle)
    Typedef for interrupt handler.
IRQn_Type const kFlexcommIrqs[]
    Array with IRQ number for each FLEXCOMM module.
uint32_t FLEXCOMM_GetInstance(void *base)
    Returns instance number for FLEXCOMM module with given base address.
status_t FLEXCOMM_Init(void *base, FLEXCOMM_PERIPH_T periph)
    Initializes FLEXCOMM and selects peripheral mode according to the second parameter.
void FLEXCOMM_SetIRQHandler(void *base, flexcomm_irq_handler_t handler, void
    *flexcommHandle)
    Sets IRQ handler for given FLEXCOMM module. It is used by drivers register IRQ handler
    according to FLEXCOMM mode.

```

2.16 FLEXSPI: Flexible Serial Peripheral Interface Driver

```

uint32_t FLEXSPI_GetInstance(FLEXSPI_Type *base)
    Get the instance number for FLEXSPI.

```

Parameters

- base – FLEXSPI base pointer.

```

status_t FLEXSPI_CheckAndClearError(FLEXSPI_Type *base, uint32_t status)
    Check and clear IP command execution errors.

```

Parameters

- base – FLEXSPI base pointer.
- status – interrupt status.

```

void FLEXSPI_Init(FLEXSPI_Type *base, const flexspi_config_t *config)
    Initializes the FLEXSPI module and internal state.

```

This function enables the clock for FLEXSPI and also configures the FLEXSPI with the input configure parameters. Users should call this function before any FLEXSPI operations.

Parameters

- base – FLEXSPI peripheral base address.
- config – FLEXSPI configure structure.

```

void FLEXSPI_GetDefaultConfig(flexspi_config_t *config)
    Gets default settings for FLEXSPI.

```

Parameters

- config – FLEXSPI configuration structure.

`void FLEXSPI_Deinit(FLEXSPI_Type *base)`

Deinitializes the FLEXSPI module.

Clears the FLEXSPI state and FLEXSPI module registers.

Parameters

- `base` – FLEXSPI peripheral base address.

`void FLEXSPI_UpdateDllValue(FLEXSPI_Type *base, flexspi_device_config_t *config, flexspi_port_t port)`

Update FLEXSPI DLL value depending on currently flexspi root clock.

Parameters

- `base` – FLEXSPI peripheral base address.
- `config` – Flash configuration parameters.
- `port` – FLEXSPI Operation port.

`void FLEXSPI_SetFlashConfig(FLEXSPI_Type *base, flexspi_device_config_t *config, flexspi_port_t port)`

Configures the connected device parameter.

This function configures the connected device relevant parameters, such as the size, command, and so on. The flash configuration value cannot have a default value. The user needs to configure it according to the connected device.

Parameters

- `base` – FLEXSPI peripheral base address.
- `config` – Flash configuration parameters.
- `port` – FLEXSPI Operation port.

`void FLEXSPI_SoftwareReset(FLEXSPI_Type *base)`

Software reset for the FLEXSPI logic.

This function sets the software reset flags for both AHB and buffer domain and resets both AHB buffer and also IP FIFOs.

Parameters

- `base` – FLEXSPI peripheral base address.

`static inline void FLEXSPI_Enable(FLEXSPI_Type *base, bool enable)`

Enables or disables the FLEXSPI module.

Parameters

- `base` – FLEXSPI peripheral base address.
- `enable` – True means enable FLEXSPI, false means disable.

`static inline void FLEXSPI_EnableInterrupts(FLEXSPI_Type *base, uint32_t mask)`

Enables the FLEXSPI interrupts.

Parameters

- `base` – FLEXSPI peripheral base address.
- `mask` – FLEXSPI interrupt source.

`static inline void FLEXSPI_DisableInterrupts(FLEXSPI_Type *base, uint32_t mask)`

Disable the FLEXSPI interrupts.

Parameters

- `base` – FLEXSPI peripheral base address.

- mask – FLEXSPI interrupt source.

static inline void FLEXSPI_EnableTxDMA(FLEXSPI_Type *base, bool enable)

Enables or disables FLEXSPI IP Tx FIFO DMA requests.

Parameters

- base – FLEXSPI peripheral base address.
- enable – Enable flag for transmit DMA request. Pass true for enable, false for disable.

static inline void FLEXSPI_EnableRxDMA(FLEXSPI_Type *base, bool enable)

Enables or disables FLEXSPI IP Rx FIFO DMA requests.

Parameters

- base – FLEXSPI peripheral base address.
- enable – Enable flag for receive DMA request. Pass true for enable, false for disable.

static inline uint32_t FLEXSPI_GetTxFifoAddress(FLEXSPI_Type *base)

Gets FLEXSPI IP tx fifo address for DMA transfer.

Parameters

- base – FLEXSPI peripheral base address.

Return values

The – tx fifo address.

static inline uint32_t FLEXSPI_GetRxFifoAddress(FLEXSPI_Type *base)

Gets FLEXSPI IP rx fifo address for DMA transfer.

Parameters

- base – FLEXSPI peripheral base address.

Return values

The – rx fifo address.

static inline void FLEXSPI_ResetFifos(FLEXSPI_Type *base, bool txFifo, bool rxFifo)

Clears the FLEXSPI IP FIFO logic.

Parameters

- base – FLEXSPI peripheral base address.
- txFifo – Pass true to reset TX FIFO.
- rxFifo – Pass true to reset RX FIFO.

static inline void FLEXSPI_GetFifoCounts(FLEXSPI_Type *base, size_t *txCount, size_t *rxCount)

Gets the valid data entries in the FLEXSPI FIFOs.

Parameters

- base – FLEXSPI peripheral base address.
- txCount – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- rxCount – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

```
static inline uint32_t FLEXSPI_GetInterruptStatusFlags(FLEXSPI_Type *base)
```

Get the FLEXSPI interrupt status flags.

Parameters

- base – FLEXSPI peripheral base address.

Return values

interrupt – status flag, use status flag to AND flexspi_flags_t could get the related status.

```
static inline void FLEXSPI_ClearInterruptStatusFlags(FLEXSPI_Type *base, uint32_t mask)
```

Get the FLEXSPI interrupt status flags.

Parameters

- base – FLEXSPI peripheral base address.
- mask – FLEXSPI interrupt source.

```
static inline void FLEXSPI_GetDataLearningPhase(FLEXSPI_Type *base, uint8_t *portAPhase,  
                                                uint8_t *portBPhase)
```

Gets the sampling clock phase selection after Data Learning.

Parameters

- base – FLEXSPI peripheral base address.
- portAPhase – Pointer to a uint8_t type variable to receive the selected clock phase on PORTA.
- portBPhase – Pointer to a uint8_t type variable to receive the selected clock phase on PORTB.

```
static inline flexspi_arb_command_source_t FLEXSPI_GetArbitratorCommandSource(FLEXSPI_Type  
                                                                              *base)
```

Gets the trigger source of current command sequence granted by arbitrator.

Parameters

- base – FLEXSPI peripheral base address.

Return values

trigger – source of current command sequence.

```
static inline flexspi_ip_error_code_t FLEXSPI_GetIPCommandErrorCode(FLEXSPI_Type *base,  
                                                                    uint8_t *index)
```

Gets the error code when IP command error detected.

Parameters

- base – FLEXSPI peripheral base address.
- index – Pointer to a uint8_t type variable to receive the sequence index when error detected.

Return values

error – code when IP command error detected.

```
static inline flexspi_ahb_error_code_t FLEXSPI_GetAHBCommandErrorCode(FLEXSPI_Type  
                                                                    *base, uint8_t  
                                                                    *index)
```

Gets the error code when AHB command error detected.

Parameters

- base – FLEXSPI peripheral base address.

- `index` – Pointer to a `uint8_t` type variable to receive the sequence index when error detected.

Return values

`error` – code when AHB command error detected.

`static inline bool FLEXSPI_GetBusIdleStatus(FLEXSPI_Type *base)`

Returns whether the bus is idle.

Parameters

- `base` – FLEXSPI peripheral base address.

Return values

- `true` – Bus is idle.
- `false` – Bus is busy.

`void FLEXSPI_UpdateRxSampleClock(FLEXSPI_Type *base, flexspi_read_sample_clock_t clockSource)`

Update read sample clock source.

Parameters

- `base` – FLEXSPI peripheral base address.
- `clockSource` – clockSource of type `flexspi_read_sample_clock_t`

`void FLEXSPI_UpdateLUT(FLEXSPI_Type *base, uint32_t index, const uint32_t *cmd, uint32_t count)`

Updates the LUT table.

Parameters

- `base` – FLEXSPI peripheral base address.
- `index` – From which index start to update. It could be any index of the LUT table, which also allows user to update command content inside a command. Each command consists of up to 8 instructions and occupy 4*32-bit memory.
- `cmd` – Command sequence array.
- `count` – Number of sequences.

`static inline void FLEXSPI_WriteData(FLEXSPI_Type *base, uint32_t data, uint8_t fifoIndex)`

Writes data into FIFO.

Parameters

- `base` – FLEXSPI peripheral base address
- `data` – The data bytes to send
- `fifoIndex` – Destination fifo index.

`static inline uint32_t FLEXSPI_ReadData(FLEXSPI_Type *base, uint8_t fifoIndex)`

Receives data from data FIFO.

Parameters

- `base` – FLEXSPI peripheral base address
- `fifoIndex` – Source fifo index.

Returns

The data in the FIFO.

status_t FLEXSPI_WriteBlocking(FLEXSPI_Type *base, uint8_t *buffer, size_t size)

Sends a buffer of data bytes using blocking method.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- base – FLEXSPI peripheral base address
- buffer – The data bytes to send
- size – The number of data bytes to send

Return values

- kStatus_Success – write success without error
- kStatus_FLEXSPI_SequenceExecutionTimeout – sequence execution timeout
- kStatus_FLEXSPI_IpCommandSequenceError – IP command sequence error detected
- kStatus_FLEXSPI_IpCommandGrantTimeout – IP command grant timeout detected

status_t FLEXSPI_ReadBlocking(FLEXSPI_Type *base, uint8_t *buffer, size_t size)

Receives a buffer of data bytes using a blocking method.

Note: This function blocks via polling until all bytes have been sent.

Parameters

- base – FLEXSPI peripheral base address
- buffer – The data bytes to send
- size – The number of data bytes to receive

Return values

- kStatus_Success – read success without error
- kStatus_FLEXSPI_SequenceExecutionTimeout – sequence execution timeout
- kStatus_FLEXSPI_IpCommandSequenceError – IP command sequence error detected
- kStatus_FLEXSPI_IpCommandGrantTimeout – IP command grant timeout detected

status_t FLEXSPI_TransferBlocking(FLEXSPI_Type *base, flexspi_transfer_t *xfer)

Execute command to transfer a buffer data bytes using a blocking method.

Parameters

- base – FLEXSPI peripheral base address
- xfer – pointer to the transfer structure.

Return values

- kStatus_Success – command transfer success without error

- `kStatus_FLEXSPI_SequenceExecutionTimeout` – sequence execution timeout
- `kStatus_FLEXSPI_IpCommandSequenceError` – IP command sequence error detected
- `kStatus_FLEXSPI_IpCommandGrantTimeout` – IP command grant timeout detected

`void FLEXSPI_TransferCreateHandle(FLEXSPI_Type *base, flexspi_handle_t *handle, flexspi_transfer_callback_t callback, void *userData)`

Initializes the FLEXSPI handle which is used in transactional functions.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure to store the transfer state.
- `callback` – pointer to user callback function.
- `userData` – user parameter passed to the callback function.

`status_t FLEXSPI_TransferNonBlocking(FLEXSPI_Type *base, flexspi_handle_t *handle, flexspi_transfer_t *xfer)`

Performs a interrupt non-blocking transfer on the FLEXSPI bus.

Note: Calling the API returns immediately after transfer initiates. The user needs to call `FLEXSPI_GetTransferCount` to poll the transfer status to check whether the transfer is finished. If the return status is not `kStatus_FLEXSPI_Busy`, the transfer is finished. For `FLEXSPI_Read`, the `dataSize` should be multiple of rx watermark level, or FLEXSPI could not read data properly.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure which stores the transfer state.
- `xfer` – pointer to `flexspi_transfer_t` structure.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_FLEXSPI_Busy` – Previous transmission still not finished.

`status_t FLEXSPI_TransferGetCount(FLEXSPI_Type *base, flexspi_handle_t *handle, size_t *count)`

Gets the master transfer status during a interrupt non-blocking transfer.

Parameters

- `base` – FLEXSPI peripheral base address.
- `handle` – pointer to `flexspi_handle_t` structure which stores the transfer state.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – count is Invalid.
- `kStatus_Success` – Successfully return the count.

void FLEXSPI_TransferAbort(FLEXSPI_Type *base, *flexspi_handle_t* *handle)

Aborts an interrupt non-blocking transfer early.

Note: This API can be called at any time when an interrupt non-blocking transfer initiates to abort the transfer early.

Parameters

- base – FLEXSPI peripheral base address.
- handle – pointer to *flexspi_handle_t* structure which stores the transfer state

void FLEXSPI_TransferHandleIRQ(FLEXSPI_Type *base, *flexspi_handle_t* *handle)

Master interrupt handler.

Parameters

- base – FLEXSPI peripheral base address.
- handle – pointer to *flexspi_handle_t* structure.

FSL_FLEXSPI_DRIVER_VERSION

FLEXSPI driver version.

Status structure of FLEXSPI.

Values:

enumerator kStatus_FLEXSPI_Busy

FLEXSPI is busy

enumerator kStatus_FLEXSPI_SequenceExecutionTimeout

Sequence execution timeout error occurred during FLEXSPI transfer.

enumerator kStatus_FLEXSPI_IpCommandSequenceError

IP command Sequence execution timeout error occurred during FLEXSPI transfer.

enumerator kStatus_FLEXSPI_IpCommandGrantTimeout

IP command grant timeout error occurred during FLEXSPI transfer.

CMD definition of FLEXSPI, use to form LUT instruction, *_flexspi_command*.

Values:

enumerator kFLEXSPI_Command_STOP

Stop execution, deassert CS.

enumerator kFLEXSPI_Command_SDR

Transmit Command code to Flash, using SDR mode.

enumerator kFLEXSPI_Command_RADDR_SDR

Transmit Row Address to Flash, using SDR mode.

enumerator kFLEXSPI_Command_CADDR_SDR

Transmit Column Address to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE1_SDR

Transmit 1-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE2_SDR
Transmit 2-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE4_SDR
Transmit 4-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_MODE8_SDR
Transmit 8-bit Mode bits to Flash, using SDR mode.

enumerator kFLEXSPI_Command_WRITE_SDR
Transmit Programming Data to Flash, using SDR mode.

enumerator kFLEXSPI_Command_READ_SDR
Receive Read Data from Flash, using SDR mode.

enumerator kFLEXSPI_Command_LEARN_SDR
Receive Read Data or Preamble bit from Flash, SDR mode.

enumerator kFLEXSPI_Command_DATSZ_SDR
Transmit Read/Program Data size (byte) to Flash, SDR mode.

enumerator kFLEXSPI_Command_DUMMY_SDR
Leave data lines undriven by FlexSPI controller.

enumerator kFLEXSPI_Command_DUMMY_RWDS_SDR
Leave data lines undriven by FlexSPI controller, dummy cycles decided by RWDS.

enumerator kFLEXSPI_Command_DDR
Transmit Command code to Flash, using DDR mode.

enumerator kFLEXSPI_Command_RADDR_DDR
Transmit Row Address to Flash, using DDR mode.

enumerator kFLEXSPI_Command_CADDR_DDR
Transmit Column Address to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE1_DDR
Transmit 1-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE2_DDR
Transmit 2-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE4_DDR
Transmit 4-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_MODE8_DDR
Transmit 8-bit Mode bits to Flash, using DDR mode.

enumerator kFLEXSPI_Command_WRITE_DDR
Transmit Programming Data to Flash, using DDR mode.

enumerator kFLEXSPI_Command_READ_DDR
Receive Read Data from Flash, using DDR mode.

enumerator kFLEXSPI_Command_LEARN_DDR
Receive Read Data or Preamble bit from Flash, DDR mode.

enumerator kFLEXSPI_Command_DATSZ_DDR
Transmit Read/Program Data size (byte) to Flash, DDR mode.

enumerator kFLEXSPI_Command_DUMMY_DDR
Leave data lines undriven by FlexSPI controller.

enumerator kFLEXSPI_Command_DUMMY_RWDS_DDR

Leave data lines undriven by FlexSPI controller, dummy cycles decided by RWDS.

enumerator kFLEXSPI_Command_JUMP_ON_CS

Stop execution, deassert CS and save operand[7:0] as the instruction start pointer for next sequence

enum _flexspi_pad

pad definition of FLEXSPI, use to form LUT instruction.

Values:

enumerator kFLEXSPI_1PAD

Transmit command/address and transmit/receive data only through DATA0/DATA1.

enumerator kFLEXSPI_2PAD

Transmit command/address and transmit/receive data only through DATA[1:0].

enumerator kFLEXSPI_4PAD

Transmit command/address and transmit/receive data only through DATA[3:0].

enumerator kFLEXSPI_8PAD

Transmit command/address and transmit/receive data only through DATA[7:0].

enum _flexspi_flags

FLEXSPI interrupt status flags.

Values:

enumerator kFLEXSPI_SequenceExecutionTimeoutFlag

Sequence execution timeout.

enumerator kFLEXSPI_AhbBusTimeoutFlag

AHB Bus timeout.

enumerator kFLEXSPI_SckStoppedBecauseTxEmptyFlag

SCK is stopped during command sequence because Async TX FIFO empty.

enumerator kFLEXSPI_SckStoppedBecauseRxFullFlag

SCK is stopped during command sequence because Async RX FIFO full.

enumerator kFLEXSPI_DataLearningFailedFlag

Data learning failed.

enumerator kFLEXSPI_IpTxFifoWatermarkEmptyFlag

IP TX FIFO WaterMark empty.

enumerator kFLEXSPI_IpRxFifoWatermarkAvailableFlag

IP RX FIFO WaterMark available.

enumerator kFLEXSPI_AhbCommandSequenceErrorFlag

AHB triggered Command Sequences Error.

enumerator kFLEXSPI_IpCommandSequenceErrorFlag

IP triggered Command Sequences Error.

enumerator kFLEXSPI_AhbCommandGrantTimeoutFlag

AHB triggered Command Sequences Grant Timeout.

enumerator kFLEXSPI_IpCommandGrantTimeoutFlag

IP triggered Command Sequences Grant Timeout.

enumerator kFLEXSPI_IpCommandExecutionDoneFlag

IP triggered Command Sequences Execution finished.

enumerator kFLEXSPI_AllInterruptFlags

All flags.

enum _flexspi_read_sample_clock

FLEXSPI sample clock source selection for Flash Reading.

Values:

enumerator kFLEXSPI_ReadSampleClkLoopbackInternally

Dummy Read strobe generated by FlexSPI Controller and loopback internally.

enumerator kFLEXSPI_ReadSampleClkLoopbackFromDqsPad

Dummy Read strobe generated by FlexSPI Controller and loopback from DQS pad.

enumerator kFLEXSPI_ReadSampleClkLoopbackFromSckPad

SCK output clock and loopback from SCK pad.

enumerator kFLEXSPI_ReadSampleClkExternalInputFromDqsPad

Flash provided Read strobe and input from DQS pad.

enum _flexspi_cs_interval_cycle_unit

FLEXSPI interval unit for flash device select.

Values:

enumerator kFLEXSPI_CsIntervalUnit1SckCycle

Chip selection interval: CSINTERVAL * 1 serial clock cycle.

enumerator kFLEXSPI_CsIntervalUnit256SckCycle

Chip selection interval: CSINTERVAL * 256 serial clock cycle.

enum _flexspi_ahb_write_wait_unit

FLEXSPI AHB wait interval unit for writing.

Values:

enumerator kFLEXSPI_AhbWriteWaitUnit2AhbCycle

AWRWAIT unit is 2 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit8AhbCycle

AWRWAIT unit is 8 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit32AhbCycle

AWRWAIT unit is 32 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit128AhbCycle

AWRWAIT unit is 128 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit512AhbCycle

AWRWAIT unit is 512 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit2048AhbCycle

AWRWAIT unit is 2048 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit8192AhbCycle

AWRWAIT unit is 8192 ahb clock cycle.

enumerator kFLEXSPI_AhbWriteWaitUnit32768AhbCycle

AWRWAIT unit is 32768 ahb clock cycle.

enum _flexspi_ip_error_code

Error Code when IP command Error detected.

Values:

enumerator kFLEXSPI_IpCmdErrorNoError

No error.

enumerator kFLEXSPI_IpCmdErrorJumpOnCsInIpCmd

IP command with JMP_ON_CS instruction used.

enumerator kFLEXSPI_IpCmdErrorUnknownOpCode

Unknown instruction opcode in the sequence.

enumerator kFLEXSPI_IpCmdErrorSdrDummyInDdrSequence

Instruction DUMMY_SDR/DUMMY_RWDS_SDR used in DDR sequence.

enumerator kFLEXSPI_IpCmdErrorDdrDummyInSdrSequence

Instruction DUMMY_DDR/DUMMY_RWDS_DDR used in SDR sequence.

enumerator kFLEXSPI_IpCmdErrorInvalidAddress

Flash access start address exceed the whole flash address range (A1/A2/B1/B2).

enumerator kFLEXSPI_IpCmdErrorSequenceExecutionTimeout

Sequence execution timeout.

enumerator kFLEXSPI_IpCmdErrorFlashBoundaryAcrosss

Flash boundary crossed.

enum _flexspi_ahb_error_code

Error Code when AHB command Error detected.

Values:

enumerator kFLEXSPI_AhbCmdErrorNoError

No error.

enumerator kFLEXSPI_AhbCmdErrorJumpOnCsInWriteCmd

AHB Write command with JMP_ON_CS instruction used in the sequence.

enumerator kFLEXSPI_AhbCmdErrorUnknownOpCode

Unknown instruction opcode in the sequence.

enumerator kFLEXSPI_AhbCmdErrorSdrDummyInDdrSequence

Instruction DUMMY_SDR/DUMMY_RWDS_SDR used in DDR sequence.

enumerator kFLEXSPI_AhbCmdErrorDdrDummyInSdrSequence

Instruction DUMMY_DDR/DUMMY_RWDS_DDR used in SDR sequence.

enumerator kFLEXSPI_AhbCmdSequenceExecutionTimeout

Sequence execution timeout.

enum _flexspi_port

FLEXSPI operation port select.

Values:

enumerator kFLEXSPI_PortA1

Access flash on A1 port.

enumerator kFLEXSPI_PortA2

Access flash on A2 port.

enumerator kFLEXSPI_PortB1

Access flash on B1 port.

enumerator kFLEXSPI_PortB2

Access flash on B2 port.

enumerator kFLEXSPI_PortCount

enum _flexspi_arb_command_source

Trigger source of current command sequence granted by arbitrator.

Values:

enumerator kFLEXSPI_AhbReadCommand

enumerator kFLEXSPI_AhbWriteCommand

enumerator kFLEXSPI_IpCommand

enumerator kFLEXSPI_SuspendedCommand

enum _flexspi_command_type

Command type.

Values:

enumerator kFLEXSPI_Command

FlexSPI operation: Only command, both TX and Rx buffer are ignored.

enumerator kFLEXSPI_Config

FlexSPI operation: Configure device mode, the TX fifo size is fixed in LUT.

enumerator kFLEXSPI_Read

enumerator kFLEXSPI_Write

typedef enum _flexspi_pad flexspi_pad_t

pad definition of FLEXSPI, use to form LUT instruction.

typedef enum _flexspi_flags flexspi_flags_t

FLEXSPI interrupt status flags.

typedef enum _flexspi_read_sample_clock flexspi_read_sample_clock_t

FLEXSPI sample clock source selection for Flash Reading.

typedef enum _flexspi_cs_interval_cycle_unit flexspi_cs_interval_cycle_unit_t

FLEXSPI interval unit for flash device select.

typedef enum _flexspi_ahb_write_wait_unit flexspi_ahb_write_wait_unit_t

FLEXSPI AHB wait interval unit for writing.

typedef enum _flexspi_ip_error_code flexspi_ip_error_code_t

Error Code when IP command Error detected.

typedef enum _flexspi_ahb_error_code flexspi_ahb_error_code_t

Error Code when AHB command Error detected.

typedef enum _flexspi_port flexspi_port_t

FLEXSPI operation port select.

typedef enum _flexspi_arb_command_source flexspi_arb_command_source_t

Trigger source of current command sequence granted by arbitrator.

typedef enum _flexspi_command_type flexspi_command_type_t

Command type.

typedef struct _flexspi_ahbBuffer_config flexspi_ahbBuffer_config_t

typedef struct _flexspi_config flexspi_config_t

FLEXSPI configuration structure.

```
typedef struct _flexspi_device_config flexspi_device_config_t
```

External device configuration items.

```
typedef struct _flexspi_transfer flexspi_transfer_t
```

Transfer structure for FLEXSPI.

```
typedef struct _flexspi_handle flexspi_handle_t
```

```
typedef void (*flexspi_transfer_callback_t)(FLEXSPI_Type *base, flexspi_handle_t *handle,  
status_t status, void *userData)
```

FLEXSPI transfer callback function.

```
typedef struct _flexspi_addr_map_config flexspi_addr_map_config_t
```

Address mapping configuration structure.

```
FSL_FEATURE_FLEXSPI_AHB_BUFFER_COUNT
```

```
FLEXSPI_LUT_SEQ(cmd0, pad0, op0, cmd1, pad1, op1)
```

Formula to form FLEXSPI instructions in LUT table.

```
struct _flexspi_ahbBuffer_config
```

```
#include <fsl_flexspi.h>
```

Public Members

```
uint8_t priority
```

This priority for AHB Master Read which this AHB RX Buffer is assigned.

```
uint8_t masterIndex
```

AHB Master ID the AHB RX Buffer is assigned.

```
uint16_t bufferSize
```

AHB buffer size in byte.

```
bool enablePrefetch
```

AHB Read Prefetch Enable for current AHB RX Buffer corresponding Master, allows prefetch disable/enable separately for each master.

```
struct _flexspi_config
```

```
#include <fsl_flexspi.h> FLEXSPI configuration structure.
```

Public Members

```
flexspi_read_sample_clock_t rxSampleClock
```

Sample Clock source selection for Flash Reading.

```
bool enableSckFreeRunning
```

Enable/disable SCK output free-running.

```
bool enableCombination
```

Enable/disable combining PORT A and B Data Pins (SIOA[3:0] and SIOB[3:0]) to support Flash Octal mode.

```
bool enableDoze
```

Enable/disable doze mode support.

```
bool enableHalfSpeedAccess
```

Enable/disable divide by 2 of the clock for half speed commands.

flexspi_read_sample_clock_t rxSampleClockPortB
Sample Clock source_b selection for Flash Reading.

bool rxSampleClockDiff
Sample Clock source or source_b selection for Flash Reading.

bool enableSckBDiffOpt
Enable/disable SCKB pad use as SCKA differential clock output, when enable, Port B flash access is not available.

bool enableSameConfigForAll
Enable/disable same configuration for all connected devices when enabled, same configuration in FLASHA1CRx is applied to all.

uint16_t seqTimeoutCycle
Timeout wait cycle for command sequence execution, timeout after ahbGrantTimeoutCycle*1024 serial root clock cycles.

uint8_t ipGrantTimeoutCycle
Timeout wait cycle for IP command grant, timeout after ipGrantTimeoutCycle*1024 AHB clock cycles.

uint8_t txWatermark
FLEXSPI IP transmit watermark value.

uint8_t rxWatermark
FLEXSPI receive watermark value.

struct __flexspi_device_config
#include <fsl_flexspi.h> External device configuration items.

Public Members

uint32_t flexspiRootClk
FLEXSPI serial root clock.

bool isSck2Enabled
FLEXSPI use SCK2.

uint32_t flashSize
Flash size in KByte.

bool addressShift
Address shift.

flexspi_cs_interval_cycle_unit_t CSIntervalUnit
CS interval unit, 1 or 256 cycle.

uint16_t CSInterval
CS line assert interval, multiply CS interval unit to get the CS line assert interval cycles.

uint8_t CSHoldTime
CS line hold time.

uint8_t CSSetupTime
CS line setup time.

uint8_t dataValidTime
Data valid time for external device.

uint8_t columnSpace

Column space size.

bool enableWordAddress

If enable word address.

uint8_t AWRSeqIndex

Sequence ID for AHB write command.

uint8_t AWRSeqNumber

Sequence number for AHB write command.

uint8_t ARDSeqIndex

Sequence ID for AHB read command.

uint8_t ARDSeqNumber

Sequence number for AHB read command.

flexspi_ahb_write_wait_unit_t AHBWriteWaitUnit

AHB write wait unit.

uint16_t AHBWriteWaitInterval

AHB write wait interval, multiply AHB write interval unit to get the AHB write wait cycles.

bool enableWriteMask

Enable/Disable FLEXSPI drive DQS pin as write mask when writing to external device.

struct *_flexspi_transfer*

#include <fsl_flexspi.h> Transfer structure for FLEXSPI.

Public Members

uint32_t deviceAddress

Operation device address.

flexspi_port_t port

Operation port.

flexspi_command_type_t cmdType

Execution command type.

uint8_t seqIndex

Sequence ID for command.

uint8_t SeqNumber

Sequence number for command.

uint32_t *data

Data buffer.

size_t dataSize

Data size in bytes.

struct *_flexspi_handle*

#include <fsl_flexspi.h> Transfer handle structure for FLEXSPI.

Public Members

uint32_t state

Internal state for FLEXSPI transfer

uint8_t *data

Data buffer.

size_t dataSize

Remaining Data size in bytes.

size_t transferTotalSize

Total Data size in bytes.

flexspi_transfer_callback_t completionCallback

Callback for users while transfer finish or error occurred

void *userData

FLEXSPI callback function parameter.

struct *_flexspi_addr_map_config*

#include <fsl_flexspi.h> Address mapping configuration structure.

Public Members

uint32_t addrStart

Remapping start address.

uint32_t addrEnd

Remapping end address.

uint32_t addrOffset

Address offset.

bool remapEnable

Enable address remapping.

struct ahbConfig

Public Members

uint8_t ahbGrantTimeoutCycle

Timeout wait cycle for AHB command grant, timeout after ahbGrantTimeoutCycle*1024 AHB clock cycles.

uint16_t ahbBusTimeoutCycle

Timeout wait cycle for AHB read/write access, timeout after ahbBusTimeoutCycle*1024 AHB clock cycles.

uint8_t resumeWaitCycle

Wait cycle for idle state before suspended command sequence resume, timeout after ahbBusTimeoutCycle AHB clock cycles.

flexspi_ahbBuffer_config_t buffer[FSL_FEATURE_FLEXSPI_AHB_BUFFER_COUNTn(0)]

AHB buffer size.

bool enableClearAHBBufferOpt

Enable/disable automatically clean AHB RX Buffer and TX Buffer when FLEXSPI returns STOP mode ACK.

`bool enableReadAddressOpt`

Enable/disable remove AHB read burst start address alignment limitation. when enable, there is no AHB read burst start address alignment limitation.

`bool enableAHBPrefetch`

Enable/disable AHB read prefetch feature, when enabled, FLEXSPI will fetch more data than current AHB burst.

`bool enableAHBBufferable`

Enable/disable AHB bufferable write access support, when enabled, FLEXSPI return before waiting for command execution finished.

`bool enableAHBCachable`

Enable AHB bus cachable read access support.

2.17 FLEXSPI DMA Driver

```
void FLEXSPI_TransferCreateHandleDMA(FLEXSPI_Type *base, flexspi_dma_handle_t *handle,
                                     flexspi_dma_callback_t callback, void *userData,
                                     dma_handle_t *txDmaHandle, dma_handle_t
                                     *rxDmaHandle)
```

Initializes the FLEXSPI handle for transfer which is used in transactional functions and set the callback.

Parameters

- `base` – FLEXSPI peripheral base address
- `handle` – Pointer to `flexspi_dma_handle_t` structure
- `callback` – FLEXSPI callback, NULL means no callback.
- `userData` – User callback function data.
- `txDmaHandle` – User requested DMA handle for TX DMA transfer.
- `rxDmaHandle` – User requested DMA handle for RX DMA transfer.

```
void FLEXSPI_TransferUpdateSizeDMA(FLEXSPI_Type *base, flexspi_dma_handle_t *handle,
                                   flexspi_dma_transfer_nsize_t nsize)
```

Update FLEXSPI DMA transfer source data transfer size(SSIZE) and destination data transfer size(DSIZE).

See also:

`flexspi_dma_transfer_nsize_t`.

Parameters

- `base` – FLEXSPI peripheral base address
- `handle` – Pointer to `flexspi_dma_handle_t` structure
- `nsize` – FLEXSPI DMA transfer data transfer size(SSIZE/DSIZE), by default the size is `kFLEXPSI_DMAAnSize1Bytes`(one byte).

```
status_t FLEXSPI_TransferDMA(FLEXSPI_Type *base, flexspi_dma_handle_t *handle,
                             flexspi_transfer_t *xfer)
```

Transfers FLEXSPI data using an dma non-blocking method.

This function writes/receives data to/from the FLEXSPI transmit/receive FIFO. This function is non-blocking.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to flexspi_dma_handle_t structure
- xfer – FLEXSPI transfer structure.

Return values

- kStatus_FLEXSPI_Busy – FLEXSPI is busy transfer.
- kStatus_InvalidArgument – The watermark configuration is invalid, the watermark should be power of 2 to do successfully DMA transfer.
- kStatus_Success – FLEXSPI successfully start dma transfer.

void FLEXSPI_TransferAbortDMA(FLEXSPI_Type *base, flexspi_dma_handle_t *handle)

Aborts the transfer data using dma.

This function aborts the transfer data using dma.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to flexspi_dma_handle_t structure

status_t FLEXSPI_TransferGetTransferCountDMA(FLEXSPI_Type *base, flexspi_dma_handle_t *handle, size_t *count)

Gets the transferred counts of transfer.

Parameters

- base – FLEXSPI peripheral base address.
- handle – Pointer to flexspi_dma_handle_t structure.
- count – Bytes transfer.

Return values

- kStatus_Success – Succeed get the transfer count.
- kStatus_NoTransferInProgress – There is not a non-blocking transaction currently in progress.

FSL_FLEXSPI_DMA_DRIVER_VERSION

FLEXSPI DMA driver version 2.2.1.

enum flexspi_dma_ntransfer_size

dma transfer configuration

Values:

enumerator kFLEXPSI_DMAnSize1Bytes

Source/Destination data transfer size is 1 byte every time

enumerator kFLEXPSI_DMAnSize2Bytes

Source/Destination data transfer size is 2 bytes every time

enumerator kFLEXPSI_DMAnSize4Bytes

Source/Destination data transfer size is 4 bytes every time

typedef struct flexspi_dma_handle flexspi_dma_handle_t

typedef void (*flexspi_dma_callback_t)(FLEXSPI_Type *base, flexspi_dma_handle_t *handle, status_t status, void *userData)

FLEXSPI dma transfer callback function for finish and error.

```
typedef enum _flexspi_dma_ntransfer_size flexspi_dma_transfer_nsize_t
    dma transfer configuration

struct _flexspi_dma_handle
    #include <fsl_flexspi_dma.h> FLEXSPI DMA transfer handle, users should not touch the content of the handle.
```

Public Members

```
dma_handle_t *txDmaHandle
    dma handler for FLEXSPI Tx.

dma_handle_t *rxDmaHandle
    dma handler for FLEXSPI Rx.

size_t transferSize
    Bytes need to transfer.

flexspi_dma_transfer_nsize_t nsize
    dma SSIZE/DSIZE in each transfer.

uint8_t nbytes
    dma minor byte transfer count initially configured.

uint8_t count
    The transfer data count in a DMA request.

uint32_t state
    Internal state for FLEXSPI dma transfer.

flexspi_dma_callback_t completionCallback
    A callback function called after the dma transfer is finished.

void *userData
    User callback parameter
```

2.18 FMEAS: Frequency Measure Driver

```
static inline void FMEAS_StartMeasure(FMEAS_SYSCON_Type *base)
    Starts a frequency measurement cycle.
```

Parameters

- base – : SYSCON peripheral base address.

```
static inline bool FMEAS_IsMeasureComplete(FMEAS_SYSCON_Type *base)
    Indicates when a frequency measurement cycle is complete.
```

Parameters

- base – : SYSCON peripheral base address.

Returns

true if a measurement cycle is active, otherwise false.

```
uint32_t FMEAS_GetFrequency(FMEAS_SYSCON_Type *base, uint32_t refClockRate)
    Returns the computed value for a frequency measurement cycle.
```

Parameters

- base – : SYSCON peripheral base address.

- `refClockRate` – : Reference clock rate used during the frequency measurement cycle.

Returns

Frequency in Hz.

`FSL_FMEAS_DRIVER_VERSION`

Defines LPC Frequency Measure driver version 2.1.1.

`typedef` `FREQME_Type` `FMEAS_SYSCON_Type`

`FMEAS_SYSCON_FREQMECTRL_CAPVAL_MASK`

`FMEAS_SYSCON_FREQMECTRL_CAPVAL_SHIFT`

`FMEAS_SYSCON_FREQMECTRL_CAPVAL`

`FMEAS_SYSCON_FREQMECTRL_PROG_MASK`

`FMEAS_SYSCON_FREQMECTRL_PROG_SHIFT`

`FMEAS_SYSCON_FREQMECTRL_PROG`

2.19 GDMA: General DMA(GDMA) Driver

`void` `GDMA_Init`(`GDMA_Type *base`)

Initializes GDMA peripheral.

It ungates the GDMA access clock, after this function, the GDMA module is ready to be used.

Parameters

- `base` – GDMA peripheral base address.

`void` `GDMA_Deinit`(`GDMA_Type *base`)

Deinitializes GDMA peripheral.

Parameters

- `base` – GDMA peripheral base address.

`static inline void` `GDMA_SetChannelSourceAddress`(`GDMA_Type *base`, `uint8_t channel`, `uint32_t addr`)

Set GDMA channel source address.

Parameters

- `base` – GDMA peripheral base address.
- `channel` – GDMA channel number.
- `addr` – Source address.

`static inline void` `GDMA_SetChannelDestAddress`(`GDMA_Type *base`, `uint8_t channel`, `uint32_t addr`)

Set GDMA channel destination address.

Parameters

- `base` – GDMA peripheral base address.
- `channel` – GDMA channel number.
- `addr` – Destination address.

static inline void GDMA_StartChannel(GDMA_Type *base, uint8_t channel)

Start GDMA channel to work.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.

static inline void GDMA_StopChannel(GDMA_Type *base, uint8_t channel)

Stop GDMA channel.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.

static inline bool GDMA_IsChannelBusy(GDMA_Type *base, uint8_t channel)

Return whether GDMA channel is processing transfer.

When GDMA_StopChannel is called, if the channel is on service, it does not stop immediately, application could call this API to check whether the channel is stopped.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.

Returns

True if the channel is busy, false if not.

static inline void GDMA_EnableChannelInterrupts(GDMA_Type *base, uint8_t channel, uint32_t interrupts)

Enables the interrupt for the GDMA transfer.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.
- interrupts – The interrupts to enable, it is OR'ed value of `_gdma_interrupt_enable`.

static inline void GDMA_DisableChannelInterrupts(GDMA_Type *base, uint8_t channel, uint32_t interrupts)

Disables the interrupt for the GDMA transfer.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.
- interrupts – The interrupts to disable, it is OR'ed value of `_gdma_interrupt_enable`.

static inline uint32_t GDMA_GetChannelInterruptFlags(GDMA_Type *base, uint8_t channel)

Get the GDMA channel interrupt flags.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.

Returns

The interrupt flags, it is OR'ed value of `_gdma_interrupt_flags`.

```
static inline void GDMA_ClearChannelInterruptFlags(GDMA_Type *base, uint8_t channel,  
                                                    uint32_t flags)
```

Clear the GDMA channel interrupt flags.

The kGDMA_ChannelInterruptFlag is OR'ed status of all other unmasked interrupt flags, it could not be clear directly, it should be cleared by clear all other flags.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.
- flags – The interrupt flags to clear, it is OR'ed value of `_gdma_interrupt_flags`.

```
static inline uint32_t GDMA_GetChannelFinishedDescriptorNumber(GDMA_Type *base, uint8_t  
                                                                channel)
```

Get the number of finished descriptor.

The counter increases when an item of descriptor is done in linklist mode.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.

Returns

Number of finished descriptor.

```
static inline void GDMA_ClearChannelFinishedDescriptorNumber(GDMA_Type *base, uint8_t  
                                                             channel)
```

Clear the number of finished descriptor.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.

```
static inline void GDMA_SetChannelPriority(GDMA_Type *base, uint8_t channel,  
                                           gdma_priority_t priority)
```

Set priority of channel.

Parameters

- base – GDMA peripheral base address.
- channel – GDMA channel number.
- priority – Channel priority value.

```
status_t GDMA_SetChannelTransferConfig(GDMA_Type *base, uint8_t channel, const  
                                        gdma_channel_xfer_config_t *config)
```

Set channel transfer configuration..

This function configures the channel transfer, after configured, GDMA_StartChannel could be called to start the transfer.

This function must be called when previous transfer finished. Application can use GDMA_IsChannelBusy to check whether the channel has finished the previous work.

Note: The transfer configuration must follow the requirements:

- SRCBSIZE * SRCWIDTH == DESTBSIZE * DESTWIDTH
- If wrap not used, the address should align with WIDTH

- If wrap used, the address should align with $\text{WIDTH} * \text{BURST_SIZE}$.
-

Parameters

- base – GDMA base address.
- channel – GDMA channel number. @config Pointer to the transfer configuration.

Return values

- kStatus_Fail – GDMA is busy with previous transfer.
- kStatus_Success – Configuration set successfully.
- kStatus_InvalidArgument – Configuration wrong.

void GDMA_CreateHandle(*gdma_handle_t* *handle, GDMA_Type *base, uint8_t channel)

Creates the GDMA handle.

This function is called if using transaction API for GDMA. This function initializes the internal state of GDMA handle.

Parameters

- handle – GDMA handle pointer. It stores callback function and parameters.
- base – GDMA peripheral base address.
- channel – GDMA channel number.

void GDMA_SetCallback(*gdma_handle_t* *handle, *gdma_callback_t* callback, void *userData)

Installs a callback function for the GDMA transfer.

This callback is called in GDMA IRQ handler to inform user the interrupt status.

Parameters

- handle – GDMA handle pointer.
- callback – GDMA callback function pointer.
- userData – Parameter for callback function.

status_t GDMA_SubmitTransfer(*gdma_handle_t* *handle, *gdma_channel_xfer_config_t* *config)

Submits the GDMA channel transfer request.

After this function, user could call GDMA_StartTransfer to start GDMA transfer.

This function must be called when previous transfer finished. Application can use GDMA_IsChannelBusy to check whether the channel has finished the previous work.

Note: The transfer configuration must follow the requirements:

- $\text{SRCBSIZE} * \text{SRCWIDTH} == \text{DESTBSIZE} * \text{DESTWIDTH}$
 - If wrap not used, the address should align with WIDTH
 - If wrap used, the address should align with $\text{WIDTH} * \text{BURST_SIZE}$.
-

Parameters

- handle – GDMA handle pointer.
- config – Pointer to GDMA transfer configuration structure.

Return values

- kStatus_Fail – GDMA is busy with previous transfer.

- kStatus_Success – Configuration set successfully.
- kStatus_InvalidArgument – Configuration wrong.

void GDMA_StartTransfer(*gdma_handle_t* *handle)

GDMA start transfer.

User can call this function after GDMA_SubmitTransfer.

Parameters

- handle – GDMA handle pointer.

void GDMA_AbortTransfer(*gdma_handle_t* *handle)

Abort running transfer by handle.

When this function is called, if the channel is on service, it only stops when service finished.

Parameters

- handle – GDMA handle pointer.

void GDMA_IRQHandle(GDMA_Type *base)

GDMA IRQ handler.

This function checks all GDMA channel interrupts and inform application the interrupt flags through user registered callback.

Parameters

- base – GDMA peripheral.

FSL_GDMA_DRIVER_VERSION

enum __gdma_transfer_width

GDMA transfer width.

Values:

enumerator kGDMA_TransferWidth1Byte
1 byte.

enumerator kGDMA_TransferWidth2Byte
2 bytes.

enumerator kGDMA_TransferWidth4Byte
4 bytes.

enum __gdma_burst_size

GDMA burst size.

Values:

enumerator kGDMA_BurstSize1
Burst 1.

enumerator kGDMA_BurstSize4
Burst 4.

enumerator kGDMA_BurstSize8
Burst 8.

enumerator kGDMA_BurstSize16
Burst 16.

enumerator kGDMA_BurstSizeWrap4
Wrap 4.

enumerator kGDMA__BurstSizeWrap8
Wrap 8.

enumerator kGDMA__BurstSizeWrap16
Wrap 16.

enum _gdma_ahb_prot
GDMA AHB HPROT flags. .

Values:

enumerator kGDMA__ProtUserMode
The access is in user mode.

enumerator kGDMA__ProtPrivilegedMode
The access is in privileged mode.

enumerator kGDMA__ProtUnbufferable
The access is not bufferable.

enumerator kGDMA__ProtBufferable
The access is bufferable.

enumerator kGDMA__ProtUncacheable
The access is not cacheable.

enumerator kGDMA__ProtCacheable
The access is cacheable.

enum _gdma_priority
GDMA channel priority.

Values:

enumerator kGDMA__ChannelPriority0
Lowest channel priority - priority 0

enumerator kGDMA__ChannelPriority1
Channel priority 1

enumerator kGDMA__ChannelPriority2
Channel priority 2

enumerator kGDMA__ChannelPriority3
Channel priority 3

enumerator kGDMA__ChannelPriority4
Channel priority 4

enumerator kGDMA__ChannelPriority5
Channel priority 5

enumerator kGDMA__ChannelPriority6
Channel priority 6

enumerator kGDMA__ChannelPriority7
Channel priority 7

enumerator kGDMA__ChannelPriority8
Channel priority 8

enumerator kGDMA__ChannelPriority9
Channel priority 9

enumerator kGDMA_ChannelPriority10

Channel priority 10

enumerator kGDMA_ChannelPriority11

Channel priority 11

enumerator kGDMA_ChannelPriority12

Channel priority 12

enumerator kGDMA_ChannelPriority13

Channel priority 13

enumerator kGDMA_ChannelPriority14

Channel priority 14

enumerator kGDMA_ChannelPriority15

Highest channel priority - priority 15

enum _gdma_interrupt_enable

GDMA interrupts to enable .

Values:

enumerator kGDMA_DescriptorTransferDoneInterruptEnable

Descriptor transfer done interrupt. This happens when the descriptor is configured to generate interrupt when transfer done.

enumerator kGDMA_AddressErrorInterruptEnable

Channel source or destination address is not aligned to corresponding transfer width.

enumerator kGDMA_BusErrorInterruptEnable

AHB bus interrupt.

enumerator kGDMA_TransferDoneInterruptEnable

DMA transfer done interrupt.

enumerator kGDMA_BlockTransferDoneInterruptEnable

DMA block single/burst transfer done interrupt.

enumerator kGDMA_AllInterruptEnable

All interrupt enable.

enum _gdma_interrupt_flags

GDMA interrupt status flags. .

Values:

enumerator kGDMA_DescriptorTransferDoneFlag

Descriptor transfer done interrupt. This happens when the descriptor is configured to generate interrupt when transfer done.

enumerator kGDMA_ChannelInterruptFlag

OR of the content of the respective unmasked interrupt of channel.

enumerator kGDMA_AddressErrorFlag

Channel source or destination address is not aligned to corresponding transfer width.

enumerator kGDMA_BusErrorFlag

AHB bus interrupt.

enumerator kGDMA_TransferDoneFlag

DMA transfer done interrupt.

enumerator kGDMA_BlockTransferDoneFlag

DMA block single/burst transfer done interrupt.

enumerator kGDMA_AllInterruptFlag

All interrupt flags.

typedef enum *_gdma_transfer_width* gdma_transfer_width_t

GDMA transfer width.

typedef enum *_gdma_burst_size* gdma_burst_size_t

GDMA burst size.

typedef enum *_gdma_priority* gdma_priority_t

GDMA channel priority.

typedef struct *_gdma_channel_xfer_config* gdma_channel_xfer_config_t

GDMA channel transfer configuration.

Note: The transfer configuration must follow the requirements:

- SRCBSIZE * SRCWIDTH == DESTBSIZE * DESTWIDTH
 - If wrap not used, the address should align with WIDTH
 - If wrap used, the address should align with WIDTH * BURST_SIZE.
-

typedef void (*gdma_callback_t)(struct *_gdma_handle* *handle, void *userData, uint32_t interrupts)

Define Callback function for GDMA.

handle: Pointer to the GDMA driver handle. userData: The userData registered using GDMA_SetCallback. interrupts: The interrupts flags of the specific channel.

typedef struct *_gdma_handle* gdma_handle_t

GDMA transfer handle structure.

gdma_descriptor_t

struct __ALIGNED (16) *_gdma_descriptor*

GDMA channel link list descriptor structure.

GDMA_DESC_LLI(linkListAddr, stopAfterDescFinished, enableDescInterrupt)

Macro for GDMA link list descriptor LLI.

This macro constructs gdma_descriptor_t::lli.

Parameters

- linkListAddr – Address of next link list descriptor item.
- stopAfterDescFinished – Stop or not after this descriptor transfer done.
- enableDescInterrupt – Generate interrupt after this descriptor transfer done.

GDMA_DESC_CTRL(ahbProt, srcAddrInc, destAddrInc, srcWidth, destWidth, srcBurstSize, destBurstSize, length)

struct *_gdma_channel_xfer_config*

#include <fsl_gdma.h> GDMA channel transfer configuration.

Note: The transfer configuration must follow the requirements:

- SRCBSIZE * SRCWIDTH == DESTBSIZE * DESTWIDTH

- If wrap not used, the address should align with WIDTH
 - If wrap used, the address should align with WIDTH * BURST_SIZE.
-

Public Members

uint32_t srcAddr

Source data address

uint32_t destAddr

Destination data address

uint8_t ahbProt

GDMA AHB HPROT flags, it could be OR'ed value of _gdma_ahb_prot.

gdma_burst_size_t srcBurstSize

Source address burst size.

gdma_burst_size_t destBurstSize

Destination address burst size.

gdma_transfer_width_t srcWidth

Source transfer width.

gdma_transfer_width_t destWidth

Destination transfer width.

bool srcAddrInc

Increase source address on each successive access.

bool destAddrInc

Increase destination address on each successive access.

uint16_t transferLen

Transfer length in bytes, max value is 8 * 1024 - 1, should align with transfer size.

bool enableLinkList

Enable link list or not.

bool enableDescInterrupt

Generate interrupt when descriptor transfer finished, only used when enableLinkList is true.

bool stopAfterDescFinished

Stop channel when descriptor transfer finished, only used when enableLinkList is true.

uint32_t linkListAddr

Link list address, only used when enableLinkList is true.

struct _gdma_handle

#include <fsl_gdma.h> GDMA transfer handle structure.

Public Members

GDMA_Type *gdma

GDMA peripheral base address

uint8_t channel

GDMA channel number

gdma_callback_t callback

Callback function. Invoked interrupt happens.

void *userData

Callback function parameter

2.20 I2C: Inter-Integrated Circuit Driver

2.21 I2C DMA Driver

```
void I2C_MasterTransferCreateHandleDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,  
                                       i2c_master_dma_transfer_callback_t callback, void  
                                       *userData, dma_handle_t *dmaHandle)
```

Init the I2C handle which is used in transactional functions.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- callback – pointer to user callback function
- userData – user param passed to the callback function
- dmaHandle – DMA handle pointer

```
status_t I2C_MasterTransferDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,  
                               i2c_master_transfer_t *xfer)
```

Performs a master dma non-blocking transfer on the I2C bus.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- xfer – pointer to transfer structure of i2c_master_transfer_t

Return values

- kStatus_Success – Sucesssully complete the data transmission.
- kStatus_I2C_Busy – Previous transmission still not finished.
- kStatus_I2C_Timeout – Transfer error, wait signal timeout.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStataus_I2C_Nak – Transfer error, receive Nak during transfer.

```
status_t I2C_MasterTransferGetCountDMA(I2C_Type *base, i2c_master_dma_handle_t *handle,  
                                       size_t *count)
```

Get master transfer status during a dma non-blocking transfer.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure
- count – Number of bytes transferred so far by the non-blocking transaction.

```
void I2C_MasterTransferAbortDMA(I2C_Type *base, i2c_master_dma_handle_t *handle)
```

Abort a master dma non-blocking transfer in a early time.

Parameters

- base – I2C peripheral base address
- handle – pointer to i2c_master_dma_handle_t structure

```
FSL_I2C_DMA_DRIVER_VERSION
```

I2C DMA driver version.

```
typedef struct _i2c_master_dma_handle i2c_master_dma_handle_t
```

I2C master dma handle typedef.

```
typedef void (*i2c_master_dma_transfer_callback_t)(I2C_Type *base, i2c_master_dma_handle_t *handle, status_t status, void *userData)
```

I2C master dma transfer callback typedef.

```
typedef void (*flexcomm_i2c_dma_master_irq_handler_t)(I2C_Type *base, i2c_master_dma_handle_t *handle)
```

Typedef for master dma handler.

```
I2C_MAX_DMA_TRANSFER_COUNT
```

Maximum lenght of single DMA transfer (determined by capability of the DMA engine)

```
struct _i2c_master_dma_handle
```

#include <fsl_i2c_dma.h> I2C master dma transfer structure.

Public Members

```
uint8_t state
```

Transfer state machine current state.

```
uint32_t transferCount
```

Indicates progress of the transfer

```
uint32_t remainingBytesDMA
```

Remaining byte count to be transferred using DMA.

```
uint8_t *buf
```

Buffer pointer for current state.

```
bool checkAddrNack
```

Whether to check the nack signal is detected during addressing.

```
dma_handle_t *dmaHandle
```

The DMA handler used.

```
i2c_master_transfer_t transfer
```

Copy of the current transfer info.

```
i2c_master_dma_transfer_callback_t completionCallback
```

Callback function called after dma transfer finished.

```
void *userData
```

Callback parameter passed to callback function.

2.22 I2C Driver

FSL_I2C_DRIVER_VERSION

I2C driver version.

I2C status return codes.

Values:

enumerator kStatus_I2C_Busy

The master is already performing a transfer.

enumerator kStatus_I2C_Idle

The slave driver is idle.

enumerator kStatus_I2C_Nak

The slave device sent a NAK in response to a byte.

enumerator kStatus_I2C_InvalidParameter

Unable to proceed due to invalid parameter.

enumerator kStatus_I2C_BitError

Transferred bit was not seen on the bus.

enumerator kStatus_I2C_ArbitrationLost

Arbitration lost error.

enumerator kStatus_I2C_NoTransferInProgress

Attempt to abort a transfer when one is not in progress.

enumerator kStatus_I2C_DmaRequestFail

DMA request failed.

enumerator kStatus_I2C_StartStopError

Start and stop error.

enumerator kStatus_I2C_UnexpectedState

Unexpected state.

enumerator kStatus_I2C_Timeout

Timeout when waiting for I2C master/slave pending status to set to continue transfer.

enumerator kStatus_I2C_Addr_Nak

NAK received for Address

enumerator kStatus_I2C_EventTimeout

Timeout waiting for bus event.

enumerator kStatus_I2C_SclLowTimeout

Timeout SCL signal remains low.

enum _i2c_status_flags

I2C status flags.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingFlag

The I2C module is waiting for software interaction. bit 0

enumerator kI2C_MasterArbitrationLostFlag

The arbitration of the bus was lost. There was collision on the bus. bit 4

enumerator kI2C_MasterStartStopErrorFlag

There was an error during start or stop phase of the transaction. bit 6

enumerator kI2C_MasterIdleFlag

The I2C master idle status. bit 5

enumerator kI2C_MasterRxReadyFlag

The I2C master rx ready status. bit 1

enumerator kI2C_MasterTxReadyFlag

The I2C master tx ready status. bit 2

enumerator kI2C_MasterAddrNackFlag

The I2C master address nack status. bit 7

enumerator kI2C_MasterDataNackFlag

The I2C master data nack status. bit 3

enumerator kI2C_SlavePendingFlag

The I2C module is waiting for software interaction. bit 8

enumerator kI2C_SlaveNotStretching

Indicates whether the slave is currently stretching clock (0 = yes, 1 = no). bit 11

enumerator kI2C_SlaveSelected

Indicates whether the slave is selected by an address match. bit 14

enumerator kI2C_SlaveDeselected

Indicates that slave was previously deselected (deselect event took place, w1c). bit 15

enumerator kI2C_SlaveAddressedFlag

One of the I2C slave's 4 addresses is matched. bit 22

enumerator kI2C_SlaveReceiveFlag

Slave receive data available. bit 9

enumerator kI2C_SlaveTransmitFlag

Slave data can be transmitted. bit 10

enumerator kI2C_SlaveAddress0MatchFlag

Slave address0 match. bit 20

enumerator kI2C_SlaveAddress1MatchFlag

Slave address1 match. bit 12

enumerator kI2C_SlaveAddress2MatchFlag

Slave address2 match. bit 13

enumerator kI2C_SlaveAddress3MatchFlag

Slave address3 match. bit 21

enumerator kI2C_MonitorReadyFlag

The I2C monitor ready interrupt. bit 16

enumerator kI2C_MonitorOverflowFlag

The monitor data overrun interrupt. bit 17

enumerator kI2C_MonitorActiveFlag

The monitor is active. bit 18

enumerator kI2C_MonitorIdleFlag

The monitor idle interrupt. bit 19

enumerator kI2C_EventTimeoutFlag

The bus event timeout interrupt. bit 24

enumerator kI2C_SclTimeoutFlag

The SCL timeout interrupt. bit 25

enumerator kI2C_MasterAllClearFlags

enumerator kI2C_SlaveAllClearFlags

enumerator kI2C_CommonAllClearFlags

enum _i2c_interrupt_enable

I2C interrupt enable.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2C_MasterPendingInterruptEnable

The I2C master communication pending interrupt.

enumerator kI2C_MasterArbitrationLostInterruptEnable

The I2C master arbitration lost interrupt.

enumerator kI2C_MasterStartStopErrorInterruptEnable

The I2C master start/stop timing error interrupt.

enumerator kI2C_SlavePendingInterruptEnable

The I2C slave communication pending interrupt.

enumerator kI2C_SlaveNotStretchingInterruptEnable

The I2C slave not stretching interrupt, deep-sleep mode can be entered only when this interrupt occurs.

enumerator kI2C_SlaveDeselectedInterruptEnable

The I2C slave deselection interrupt.

enumerator kI2C_MonitorReadyInterruptEnable

The I2C monitor ready interrupt.

enumerator kI2C_MonitorOverflowInterruptEnable

The monitor data overrun interrupt.

enumerator kI2C_MonitorIdleInterruptEnable

The monitor idle interrupt.

enumerator kI2C_EventTimeoutInterruptEnable

The bus event timeout interrupt.

enumerator kI2C_SclTimeoutInterruptEnable

The SCL timeout interrupt.

enumerator kI2C_MasterAllInterruptEnable

enumerator kI2C_SlaveAllInterruptEnable

enumerator kI2C_CommonAllInterruptEnable

I2C_RETRY_TIMES

Retry times for waiting flag.

I2C_MASTER_TRANSMIT_IGNORE_LAST_NACK

Whether to ignore the nack signal of the last byte during master transmit.

I2C_STAT_MSTCODE_IDLE

Master Idle State Code

I2C_STAT_MSTCODE_RXREADY

Master Receive Ready State Code

I2C_STAT_MSTCODE_TXREADY

Master Transmit Ready State Code

I2C_STAT_MSTCODE_NACKADR

Master NACK by slave on address State Code

I2C_STAT_MSTCODE_NACKDAT

Master NACK by slave on data State Code

I2C_STAT_SLVST_ADDR

I2C_STAT_SLVST_RX

I2C_STAT_SLVST_TX

2.23 I2C Master Driver

`void I2C_MasterGetDefaultConfig(i2c_master_config_t *masterConfig)`

Provides a default configuration for the I2C master peripheral.

This function provides the following default configuration for the I2C master peripheral:

```
masterConfig->enableMaster      = true;
masterConfig->baudRate_Bps      = 100000U;
masterConfig->enableTimeout     = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I2C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `i2c_master_config_t`.

`void I2C_MasterInit(I2C_Type *base, const i2c_master_config_t *masterConfig, uint32_t srcClock_Hz)`

Initializes the I2C master peripheral.

This function enables the peripheral clock and initializes the I2C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The I2C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `I2C_MasterGetDefaultConfig()` to get a set of defaults that you can override.

- `srcClock_Hz` – Frequency in Hertz of the I2C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`void I2C_MasterDeinit(I2C_Type *base)`

Deinitializes the I2C master peripheral.

This function disables the I2C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

`uint32_t I2C_GetInstance(I2C_Type *base)`

Returns an instance number given a base address.

If an invalid base address is passed, debug builds will assert. Release builds will just return instance number 0.

Parameters

- `base` – The I2C peripheral base address.

Returns

I2C instance number starting from 0.

`static inline void I2C_MasterReset(I2C_Type *base)`

Performs a software reset.

Restores the I2C master peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

`static inline void I2C_MasterEnable(I2C_Type *base, bool enable)`

Enables or disables the I2C module as master.

Parameters

- `base` – The I2C peripheral base address.
- `enable` – Pass true to enable or false to disable the specified I2C as master.

`uint32_t I2C_GetStatusFlags(I2C_Type *base)`

Gets the I2C status flags.

A bit mask with the state of all I2C status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i2c_status_flags`.

Parameters

- `base` – The I2C peripheral base address.

Returns

State of the status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I2C_ClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C status flag state.

Refer to `kI2C_CommonAllClearStatusFlags`, `kI2C_MasterAllClearStatusFlags` and `kI2C_SlaveAllClearStatusFlags` to see the clearable flags. Attempts to clear other flags has no effect.

See also:

`_i2c_status_flags`, `_i2c_master_status_flags` and `_i2c_slave_status_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of the members in `kI2C_CommonAllClearStatusFlags`, `kI2C_MasterAllClearStatusFlags` and `kI2C_SlaveAllClearStatusFlags`. You may pass the result of a previous call to `I2C_GetStatusFlags()`.

```
static inline void I2C_MasterClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C master status flag state.

Deprecated:

Do not use this function. It has been superceded by `I2C_ClearStatusFlags`. The following status register flags can be cleared:

- `kI2C_MasterArbitrationLostFlag`
- `kI2C_MasterStartStopErrorFlag`

Attempts to clear other flags has no effect.

See also:

`_i2c_status_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i2c_status_flags` enumerators OR'd together. You may pass the result of a previous call to `I2C_GetStatusFlags()`.

```
static inline void I2C_EnableInterrupts(I2C_Type *base, uint32_t interruptMask)
```

Enables the I2C interrupt requests.

Parameters

- `base` – The I2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i2c_interrupt_enable` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I2C_DisableInterrupts(I2C_Type *base, uint32_t interruptMask)
```

Disables the I2C interrupt requests.

Parameters

- `base` – The I2C peripheral base address.

- `interruptMask` – Bit mask of interrupts to disable. See `_i2c_interrupt_enable` for the set of constants that should be OR'd together to form the bit mask.

`static inline uint32_t I2C_GetEnabledInterrupts(I2C_Type *base)`

Returns the set of currently enabled I2C interrupt requests.

Parameters

- `base` – The I2C peripheral base address.

Returns

A bitmask composed of `_i2c_interrupt_enable` enumerators OR'd together to indicate the set of enabled interrupts.

`void I2C_MasterSetBaudRate(I2C_Type *base, uint32_t baudRate_Bps, uint32_t srcClock_Hz)`

Sets the I2C bus frequency for master transactions.

The I2C master is automatically disabled and re-enabled as necessary to configure the baud rate. Do not call this function during a transfer, or the transfer is aborted.

Parameters

- `base` – The I2C peripheral base address.
- `srcClock_Hz` – I2C functional clock frequency in Hertz.
- `baudRate_Bps` – Requested bus frequency in bits per second.

`void I2C_MasterSetTimeoutValue(I2C_Type *base, uint8_t timeout_Ms, uint32_t srcClock_Hz)`

Sets the I2C bus timeout value.

If the SCL signal remains low or bus does not have event longer than the timeout value, `kI2C_SclTimeoutFlag` or `kI2C_EventTimeoutFlag` is set. This can indicate the bus is held by slave or any fault occurs to the I2C module.

Parameters

- `base` – The I2C peripheral base address.
- `timeout_Ms` – Timeout value in millisecond.
- `srcClock_Hz` – I2C functional clock frequency in Hertz.

`static inline bool I2C_MasterGetBusIdleState(I2C_Type *base)`

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The I2C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

`status_t I2C_MasterStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)`

Sends a START on the I2C bus.

This function is used to initiate a new master mode transfer by sending the START signal. The slave address is sent following the I2C START signal.

Parameters

- `base` – I2C peripheral base pointer
- `address` – 7-bit slave device address.
- `direction` – Master transfer directions(transmit/receive).

Return values

- `kStatus_Success` – Successfully send the start signal.
- `kStatus_I2C_Busy` – Current bus is busy.

`status_t I2C_MasterStop(I2C_Type *base)`

Sends a STOP signal on the I2C bus.

Return values

- `kStatus_Success` – Successfully send the stop signal.
- `kStatus_I2C_Timeout` – Send stop signal failed, timeout.

`static inline status_t I2C_MasterRepeatedStart(I2C_Type *base, uint8_t address, i2c_direction_t direction)`

Sends a REPEATED START on the I2C bus.

Parameters

- `base` – I2C peripheral base pointer
- `address` – 7-bit slave device address.
- `direction` – Master transfer directions(transmit/receive).

Return values

- `kStatus_Success` – Successfully send the start signal.
- `kStatus_I2C_Busy` – Current bus is busy but not occupied by current I2C master.

`status_t I2C_MasterWriteBlocking(I2C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)`

Performs a polling send transfer on the I2C bus.

Sends up to *txSize* number of bytes to the previously addressed slave device. The slave may reply with a NAK to any byte in order to terminate the transfer early. If this happens, this function returns `kStatus_I2C_Nak`.

Parameters

- `base` – The I2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.
- `flags` – Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use `kI2C_TransferDefaultFlag`

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_I2C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I2C_Nak` – The slave device sent a NAK in response to a byte.
- `kStatus_I2C_ArbitrationLost` – Arbitration lost error.

`status_t I2C_MasterReadBlocking(I2C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)`

Performs a polling receive transfer on the I2C bus.

Parameters

- `base` – The I2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.

- rxSize – The length in bytes of the data to be transferred.
- flags – Transfer control flag to control special behavior like suppressing start or stop, for normal transfers use kI2C_TransferDefaultFlag

Return values

- kStatus_Success – Data was received successfully.
- kStatus_I2C_Busy – Another master is currently utilizing the bus.
- kStatus_I2C_Nak – The slave device sent a NAK in response to a byte.
- kStatus_I2C_ArbitrationLost – Arbitration lost error.

status_t I2C_MasterTransferBlocking(I2C_Type *base, *i2c_master_transfer_t* *xfer)

Performs a master polling transfer on the I2C bus.

Note: The API does not return until the transfer succeeds or fails due to arbitration lost or receiving a NAK.

Parameters

- base – I2C peripheral base address.
- xfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Successfully complete the data transmission.
- kStatus_I2C_Busy – Previous transmission still not finished.
- kStatus_I2C_Timeout – Transfer error, wait signal timeout.
- kStatus_I2C_ArbitrationLost – Transfer error, arbitration lost.
- kStatus_I2C_Nak – Transfer error, receive NAK during transfer.
- kStatus_I2C_Addr_Nak – Transfer error, receive NAK during addressing.

void I2C_MasterTransferCreateHandle(I2C_Type *base, *i2c_master_handle_t* *handle, *i2c_master_transfer_callback_t* callback, *void* *userData)

Creates a new handle for the I2C master non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the I2C_MasterTransferAbort() API shall be called.

Parameters

- base – The I2C peripheral base address.
- handle – **[out]** Pointer to the I2C master driver handle.
- callback – User provided pointer to the asynchronous callback function.
- userData – User provided pointer to the application callback data.

status_t I2C_MasterTransferNonBlocking(I2C_Type *base, *i2c_master_handle_t* *handle, *i2c_master_transfer_t* *xfer)

Performs a non-blocking transaction on the I2C bus.

Parameters

- base – The I2C peripheral base address.
- handle – Pointer to the I2C master driver handle.
- xfer – The pointer to the transfer descriptor.

Return values

- `kStatus__Success` – The transaction was started successfully.
- `kStatus_I2C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

`status_t I2C_MasterTransferGetCount(I2C_Type *base, i2c_master_handle_t *handle, size_t *count)`

Returns number of bytes transferred so far.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.
- `count` – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus__Success` –
- `kStatus_I2C_Busy` –

`status_t I2C_MasterTransferAbort(I2C_Type *base, i2c_master_handle_t *handle)`

Terminates a non-blocking I2C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I2C peripheral's IRQ priority.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.

Return values

- `kStatus__Success` – A transaction was successfully aborted.
- `kStatus_I2C_Timeout` – Timeout during polling for flags.

`void I2C_MasterTransferHandleIRQ(I2C_Type *base, i2c_master_handle_t *handle)`

Reusable routine to handle master interrupts.

Note: This function does not need to be called unless you are reimplementing the non-blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to the I2C master driver handle.

`enum _i2c_direction`

Direction of master and slave transfers.

Values:

enumerator `kI2C_Write`

Master transmit.

enumerator kI2C_Read
Master receive.

enum _i2c_master_transfer_flags
Transfer option flags.

Note: These enumerations are intended to be OR'd together to form a bit mask of options for the `_i2c_master_transfer::flags` field.

Values:

enumerator kI2C_TransferDefaultFlag
Transfer starts with a start signal, stops with a stop signal.

enumerator kI2C_TransferNoStartFlag
Don't send a start condition, address, and sub address

enumerator kI2C_TransferRepeatedStartFlag
Send a repeated start condition

enumerator kI2C_TransferNoStopFlag
Don't send a stop condition.

enum _i2c_transfer_states
States for the state machine used by transactional APIs.

Values:

enumerator kIdleState

enumerator kTransmitSubaddrState

enumerator kTransmitDataState

enumerator kReceiveDataBeginState

enumerator kReceiveDataState

enumerator kReceiveLastDataState

enumerator kStartState

enumerator kStopState

enumerator kWaitForCompletionState

typedef enum _i2c_direction i2c_direction_t
Direction of master and slave transfers.

typedef struct _i2c_master_config i2c_master_config_t
Structure with settings to initialize the I2C master module.

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the `I2C_MasterGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef struct _i2c_master_transfer i2c_master_transfer_t
I2C master transfer typedef.

typedef struct _i2c_master_handle i2c_master_handle_t
I2C master handle typedef.

```
typedef void (*i2c_master_transfer_callback_t)(I2C_Type *base, i2c_master_handle_t *handle,
status_t completionStatus, void *userData)
```

Master completion callback function pointer type.

This callback is used only for the non-blocking master transfer API. Specify the callback you wish to use in the call to I2C_MasterTransferCreateHandle().

Param base

The I2C peripheral base address.

Param completionStatus

Either kStatus_Success or an error code describing how the transfer completed.

Param userData

Arbitrary pointer-sized value passed from the application.

```
struct _i2c_master_config
```

#include <fsl_i2c.h> Structure with settings to initialize the I2C master module.

This structure holds configuration settings for the I2C peripheral. To initialize this structure to reasonable defaults, call the I2C_MasterGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

bool enableMaster

Whether to enable master mode.

uint32_t baudRate_Bps

Desired baud rate in bits per second.

bool enableTimeout

Enable internal timeout function.

uint8_t timeout_Ms

Event timeout and SCL low timeout value.

```
struct _i2c_master_transfer
```

#include <fsl_i2c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the I2C_MasterTransferNonBlocking() API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration `_i2c_master_transfer_flags` for available options. Set to 0 or `kI2C_TransferDefaultFlag` for normal transfers.

uint8_t slaveAddress

The 7-bit slave address.

i2c_direction_t direction

Either `kI2C_Read` or `kI2C_Write`.

uint32_t subaddress

Sub address. Transferred MSB first.

size_t subaddressSize

Length of sub address to send in bytes. Maximum size is 4 bytes.

void *data

Pointer to data to transfer.

size_t dataSize

Number of bytes to transfer.

struct _i2c_master_handle

#include <fsl_i2c.h> Driver handle for master non-blocking APIs.

Note: The contents of this structure are private and subject to change.

Public Members

uint8_t state

Transfer state machine current state.

uint32_t transferCount

Indicates progress of the transfer

uint32_t remainingBytes

Remaining byte count in current state.

uint8_t *buf

Buffer pointer for current state.

bool checkAddrNack

Whether to check the nack signal is detected during addressing.

i2c_master_transfer_t transfer

Copy of the current transfer info.

i2c_master_transfer_callback_t completionCallback

Callback function pointer.

void *userData

Application data passed to callback.

2.24 I2C Slave Driver

void I2C_SlaveGetDefaultConfig(*i2c_slave_config_t* *slaveConfig)

Provides a default configuration for the I2C slave peripheral.

This function provides the following default configuration for the I2C slave peripheral:

```
slaveConfig->enableSlave = true;
slaveConfig->address0.disable = false;
slaveConfig->address0.address = 0u;
slaveConfig->address1.disable = true;
slaveConfig->address2.disable = true;
slaveConfig->address3.disable = true;
slaveConfig->busSpeed = kI2C_SlaveStandardMode;
```

After calling this function, override any settings to customize the configuration, prior to initializing the master driver with `I2C_SlaveInit()`. Be sure to override at least the *address0.address* member of the configuration structure with the desired slave address.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure that is set to default values. Refer to `i2c_slave_config_t`.

```
status_t I2C_SlaveInit(I2C_Type *base, const i2c_slave_config_t *slaveConfig, uint32_t  
srcClock_Hz)
```

Initializes the I2C slave peripheral.

This function enables the peripheral clock and initializes the I2C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The I2C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `I2C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `srcClock_Hz` – Frequency in Hertz of the I2C functional clock. Used to calculate CLKDIV value to provide enough data setup time for master when slave stretches the clock.

```
void I2C_SlaveSetAddress(I2C_Type *base, i2c_slave_address_register_t addressRegister, uint8_t  
address, bool addressDisable)
```

Configures Slave Address n register.

This function writes new value to Slave Address register.

Parameters

- `base` – The I2C peripheral base address.
- `addressRegister` – The module supports multiple address registers. The parameter determines which one shall be changed.
- `address` – The slave address to be stored to the address register for matching.
- `addressDisable` – Disable matching of the specified address register.

```
void I2C_SlaveDeinit(I2C_Type *base)
```

Deinitializes the I2C slave peripheral.

This function disables the I2C slave peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I2C peripheral base address.

```
static inline void I2C_SlaveEnable(I2C_Type *base, bool enable)
```

Enables or disables the I2C module as slave.

Parameters

- `base` – The I2C peripheral base address.
- `enable` – True to enable or false to disable.

```
static inline void I2C_SlaveClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C status flag state.

The following status register flags can be cleared:

- slave deselected flag

Attempts to clear other flags has no effect.

See also:

`_i2c_slave_flags`.

Parameters

- `base` – The I2C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i2c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I2C_SlaveGetStatusFlags()`.

status_t I2C_SlaveWriteBlocking(I2C_Type *base, const uint8_t *txBuff, size_t txSize)

Performs a polling send transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- `base` – The I2C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

`kStatus_Success` Data has been sent.

Returns

`kStatus_Fail_Unexpected_slave_state` (master data write while master read from slave is expected).

status_t I2C_SlaveReadBlocking(I2C_Type *base, uint8_t *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I2C bus.

The function executes blocking address phase and blocking data phase.

Parameters

- `base` – The I2C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

`kStatus_Success` Data has been received.

Returns

`kStatus_Fail_Unexpected_slave_state` (master data read while master write to slave is expected).

void I2C_SlaveTransferCreateHandle(I2C_Type *base, *i2c_slave_handle_t* *handle,
i2c_slave_transfer_callback_t callback, void *userData)

Creates a new handle for the I2C slave non-blocking APIs.

The creation of a handle is for use with the non-blocking APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I2C_SlaveTransferAbort()` API shall be called.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – **[out]** Pointer to the I2C slave driver handle.
- `callback` – User provided pointer to the asynchronous callback function.

- `userData` – User provided pointer to the application callback data.

`status_t I2C_SlaveTransferNonBlocking(I2C_Type *base, i2c_slave_handle_t *handle, uint32_t eventMask)`

Starts accepting slave transfers.

Call this API after calling `I2C_SlaveInit()` and `I2C_SlaveTransferCreateHandle()` to start processing transactions driven by an I2C master. The slave monitors the I2C bus and pass events to the callback that was passed into the call to `I2C_SlaveTransferCreateHandle()`. The callback is always invoked from the interrupt context.

If no slave Tx transfer is busy, a master read from slave request invokes `kI2C_SlaveTransmitEvent` callback. If no slave Rx transfer is busy, a master write to slave request invokes `kI2C_SlaveReceiveEvent` callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kI2C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`status_t I2C_SlaveSetSendBuffer(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, const void *txData, size_t txSize, uint32_t eventMask)`

Starts accepting master read from slave requests.

The function can be called in response to `kI2C_SlaveTransmitEvent` callback to start a new slave Tx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the *eventMask* parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kI2C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – Pointer to `i2c_slave_transfer_t` structure.
- `txData` – Pointer to data to send to master.
- `txSize` – Size of `txData` in bytes.

- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`status_t I2C_SlaveSetReceiveBuffer(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, void *rxData, size_t rxSize, uint32_t eventMask)`

Starts accepting master write to slave requests.

The function can be called in response to `kI2C_SlaveReceiveEvent` callback to start a new slave Rx transfer from within the transfer callback.

The set of events received by the callback is customizable. To do so, set the `eventMask` parameter to the OR'd combination of `i2c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI2C_SlaveTransmitEvent` and `kI2C_SlaveReceiveEvent` events are always enabled and do not need to be included in the mask. Alternatively, you can pass 0 to get a default set of only the transmit and receive events that are always enabled. In addition, the `kI2C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – Pointer to `i2c_slave_transfer_t` structure.
- `rxData` – Pointer to data to store data from master.
- `rxSize` – Size of `rxData` in bytes.
- `eventMask` – Bit mask formed by OR'ing together `i2c_slave_transfer_event_t` enumerators to specify which events to send to the callback. Other accepted values are 0 to get a default set of only the transmit and receive events, and `kI2C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I2C_Busy` – Slave transfers have already been started on this handle.

`static inline uint32_t I2C_SlaveGetReceivedAddress(I2C_Type *base, volatile i2c_slave_transfer_t *transfer)`

Returns the slave address sent by the I2C master.

This function should only be called from the address match event callback `kI2C_SlaveAddressMatchEvent`.

Parameters

- `base` – The I2C peripheral base address.
- `transfer` – The I2C slave transfer.

Returns

The 8-bit address matched by the I2C slave. Bit 0 contains the R/w direction bit, and the 7-bit slave address is in the upper 7 bits.

`void I2C_SlaveTransferAbort(I2C_Type *base, i2c_slave_handle_t *handle)`

Aborts the slave non-blocking transfers.

Note: This API could be called at any time to stop slave for handling the bus events.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.

Return values

- `kStatus_Success` –
- `kStatus_I2C_Idle` –

`status_t I2C_SlaveTransferGetCount(I2C_Type *base, i2c_slave_handle_t *handle, size_t *count)`

Gets the slave transfer remaining bytes during a interrupt non-blocking transfer.

Parameters

- `base` – I2C base pointer.
- `handle` – pointer to `i2c_slave_handle_t` structure.
- `count` – Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_InvalidArgument` – `count` is Invalid.
- `kStatus_Success` – Successfully return the count.

`void I2C_SlaveTransferHandleIRQ(I2C_Type *base, i2c_slave_handle_t *handle)`

Reusable routine to handle slave interrupts.

Note: This function does not need to be called unless you are reimplementing the non blocking API's interrupt handler routines to add special functionality.

Parameters

- `base` – The I2C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.

`enum _i2c_slave_address_register`

I2C slave address register.

Values:

enumerator `kI2C_SlaveAddressRegister0`

Slave Address 0 register.

enumerator `kI2C_SlaveAddressRegister1`

Slave Address 1 register.

enumerator `kI2C_SlaveAddressRegister2`

Slave Address 2 register.

enumerator `kI2C_SlaveAddressRegister3`

Slave Address 3 register.

enum `_i2c_slave_address_qual_mode`

I2C slave address match options.

Values:

enumerator `ki2C_QualModeMask`

The SLVQUAL0 field (`qualAddress`) is used as a logical mask for matching address0.

enumerator `ki2C_QualModeExtend`

The SLVQUAL0 (`qualAddress`) field is used to extend address 0 matching in a range of addresses.

enum `_i2c_slave_bus_speed`

I2C slave bus speed options.

Values:

enumerator `ki2C_SlaveStandardMode`

enumerator `ki2C_SlaveFastMode`

enumerator `ki2C_SlaveFastModePlus`

enumerator `ki2C_SlaveHsMode`

enum `_i2c_slave_transfer_event`

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

Values:

enumerator `ki2C_SlaveAddressMatchEvent`

Received the slave address after a start or repeated start.

enumerator `ki2C_SlaveTransmitEvent`

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator `ki2C_SlaveReceiveEvent`

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator `ki2C_SlaveCompletionEvent`

All data in the active transfer have been consumed.

enumerator `ki2C_SlaveDeselectedEvent`

The slave function has become deselected (SLVSEL flag changing from 1 to 0).

enumerator `ki2C_SlaveAllEvents`

Bit mask of all available events.

enum `_i2c_slave_fsm`

I2C slave software finite state machine states.

Values:

enumerator `ki2C_SlaveFsmAddressMatch`

enumerator `kI2C_SlaveFsmReceive`

enumerator `kI2C_SlaveFsmTransmit`

typedef enum `_i2c_slave_address_register` `i2c_slave_address_register_t`
I2C slave address register.

typedef struct `_i2c_slave_address` `i2c_slave_address_t`
Data structure with 7-bit Slave address and Slave address disable.

typedef enum `_i2c_slave_address_qual_mode` `i2c_slave_address_qual_mode_t`
I2C slave address match options.

typedef enum `_i2c_slave_bus_speed` `i2c_slave_bus_speed_t`
I2C slave bus speed options.

typedef struct `_i2c_slave_config` `i2c_slave_config_t`
Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the `I2C_SlaveGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

typedef enum `_i2c_slave_transfer_event` `i2c_slave_transfer_event_t`
Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

Note: These enumerations are meant to be OR'd together to form a bit mask of events.

typedef struct `_i2c_slave_handle` `i2c_slave_handle_t`
I2C slave handle typedef.

typedef struct `_i2c_slave_transfer` `i2c_slave_transfer_t`
I2C slave transfer structure.

typedef void (*`i2c_slave_transfer_callback_t`)(`I2C_Type` *`base`, volatile `i2c_slave_transfer_t` *`transfer`, void *`userData`)
Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the `I2C_SlaveSetCallback()` function after you have created a handle.

Param base

Base address for the I2C instance on which the event occurred.

Param transfer

Pointer to transfer descriptor containing values passed to and/or from the callback.

Param userData

Arbitrary pointer-sized value passed from the application.

typedef enum `_i2c_slave_fsm` `i2c_slave_fsm_t`
I2C slave software finite state machine states.

```
typedef void (*flexcomm_i2c_master_irq_handler_t)(I2C_Type *base, i2c_master_handle_t *handle)
```

Typedef for master interrupt handler.

```
typedef void (*flexcomm_i2c_slave_irq_handler_t)(I2C_Type *base, i2c_slave_handle_t *handle)
```

Typedef for slave interrupt handler.

```
struct _i2c_slave_address
```

#include <fsl_i2c.h> Data structure with 7-bit Slave address and Slave address disable.

Public Members

uint8_t address

7-bit Slave address SLVADR.

bool addressDisable

Slave address disable SADISABLE.

```
struct _i2c_slave_config
```

#include <fsl_i2c.h> Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the I2C_SlaveGetDefaultConfig() function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

Public Members

i2c_slave_address_t address0

Slave's 7-bit address and disable.

i2c_slave_address_t address1

Alternate slave 7-bit address and disable.

i2c_slave_address_t address2

Alternate slave 7-bit address and disable.

i2c_slave_address_t address3

Alternate slave 7-bit address and disable.

i2c_slave_address_qual_mode_t qualMode

Qualify mode for slave address 0.

uint8_t qualAddress

Slave address qualifier for address 0.

i2c_slave_bus_speed_t busSpeed

Slave bus speed mode. If the slave function stretches SCL to allow for software response, it must provide sufficient data setup time to the master before releasing the stretched clock. This is accomplished by inserting one clock time of CLKDIV at that point. The busSpeed value is used to configure CLKDIV such that one clock time is greater than the tSU;DAT value noted in the I2C bus specification for the I2C mode that is being used. If the busSpeed mode is unknown at compile time, use the longest data setup time kI2C_SlaveStandardMode (250 ns)

bool enableSlave

Enable slave mode.

```
struct _i2c_slave_transfer
```

#include <fsl_i2c.h> I2C slave transfer structure.

Public Members

i2c_slave_handle_t *handle

Pointer to handle that contains this transfer.

i2c_slave_transfer_event_t event

Reason the callback is being invoked.

uint8_t receivedAddress

Matching address send by master. 7-bits plus R/nW bit0

uint32_t eventMask

Mask of enabled events.

uint8_t *rxData

Transfer buffer for receive data

const uint8_t *txData

Transfer buffer for transmit data

size_t txSize

Transfer size

size_t rxSize

Transfer size

size_t transferredCount

Number of bytes transferred during this transfer.

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kI2C_SlaveCompletionEvent.

struct *_i2c_slave_handle*

#include <fsl_i2c.h> I2C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

volatile *i2c_slave_transfer_t* transfer

I2C slave transfer.

volatile bool isBusy

Whether transfer is busy.

volatile *i2c_slave_fsm_t* slaveFsm

slave transfer state machine.

i2c_slave_transfer_callback_t callback

Callback function called at transfer event.

void *userData

Callback parameter passed to callback.

2.25 I2S: I2S Driver

2.26 I2S_BRIDGE: I2S bridging and signal sharing configuration

enum _i2s_bridge_share_set_index

I2S Bridge share set.

Values:

enumerator kI2S_BRIDGE_OriginalSignal

Original FLEXCOMM I2S signals

enumerator kI2S_BRIDGE_ShareSet0

share set 0 signals

enumerator kI2S_BRIDGE_ShareSet1

share set 1 signals

enum _i2s_bridge_signal

I2S signal.

Values:

enumerator kI2S_BRIDGE_SignalSCK

SCK signal

enumerator kI2S_BRIDGE_SignalWS

WS signal

enumerator kI2S_BRIDGE_SignalDataIn

Data in signal

enumerator kI2S_BRIDGE_SignalDataOut

Data out signal

enum _i2s_bridge_share_src

I2S signal source.

Values:

enumerator kI2S_BRIDGE_Flexcomm0

Shared signal comes from FLEXCOMM0

enumerator kI2S_BRIDGE_Flexcomm1

Shared signal comes from FLEXCOMM1

enumerator kI2S_BRIDGE_Flexcomm2

Shared signal comes from FLEXCOMM2

enumerator kI2S_BRIDGE_Flexcomm3

Shared signal comes from FLEXCOMM3

enum _i2s_bridge_dataout_mask

I2S Bridge shared data out mask.

Values:

enumerator kI2S_BRIDGE_Flexcomm0DataOut

FLEXCOMM0 DATAOUT Output Enable

enumerator kI2S_BRIDGE_Flexcomm1DataOut

FLEXCOMM1 DATAOUT Output Enable

enumerator kI2S_BRIDGE_Flexcomm2DataOut

FLEXCOMM2 DATAOUT Output Enable

enumerator kI2S_BRIDGE_Flexcomm3DataOut

FLEXCOMM3 DATAOUT Output Enable

typedef enum *i2s_bridge_signal* i2s_bridge_signal_t
I2S signal.

FSL_I2S_BRIDGE_DRIVER_VERSION

Group I2S Bridge driver version for SDK.

Version 2.0.0.

void I2S_BRIDGE_SetFlexcommShareSet(uint32_t flexCommIndex, uint32_t sckSet, uint32_t wsSet, uint32_t dataInSet, uint32_t dataOutSet)

I2S Bridge share set selection for flexcomm instance.

Parameters

- flexCommIndex – index of flexcomm, refer to RM for supported FLEXCOMM instances.
- sckSet – share set for sck, refer to *_i2s_bridge_share_set_index*
- wsSet – share set for ws, refer to *_i2s_bridge_share_set_index*
- dataInSet – share set for data in, refer to *_i2s_bridge_share_set_index*
- dataOutSet – share set for data out, refer to *_i2s_bridge_share_set_index*

void I2S_BRIDGE_SetFlexcommSignalShareSet(uint32_t flexCommIndex, *i2s_bridge_signal_t* signal, uint32_t set)

I2S Bridge share set selection for a separate signal.

Parameters

- flexCommIndex – index of flexcomm, refer to RM for supported FLEXCOMM instances.
- signal – The signal need to be configured.
- set – share set for the signal, refer to *_i2s_bridge_share_set_index*

void I2S_BRIDGE_SetShareSetSrc(uint32_t setIndex, uint32_t sckShareSrc, uint32_t wsShareSrc, uint32_t dataInShareSrc, uint32_t dataOutShareSrc)

I2S Bridge share set source configure.

Parameters

- setIndex – index of share set, refer *_i2s_bridge_share_set_index*
- sckShareSrc – sck source for this share set, refer to *_i2s_bridge_share_src*
- wsShareSrc – ws source for this share set, refer to *_i2s_bridge_share_src*
- dataInShareSrc – data in source for this share set, refer to *_i2s_bridge_share_src*
- dataOutShareSrc – data out source for this share set, refer to *_i2s_bridge_dataout_mask*

```
void I2S_BRIDGE_SetShareSignalSrc(uint32_t setIndex, i2s_bridge_signal_t signal, uint32_t shareSrc)
```

I2S Bridge shared signal source selection for a share set.

Parameters

- `setIndex` – index of share set, refer to `_i2s_bridge_share_set_index`
- `signal` – the shared signal to be configured
- `shareSrc` – the signal selection, refer to `_i2s_bridge_share_src`.

2.27 I2S DMA Driver

```
void I2S_TxTransferCreateHandleDMA(I2S_Type *base, i2s_dma_handle_t *handle, dma_handle_t *dmaHandle, i2s_dma_transfer_callback_t callback, void *userData)
```

Initializes handle for transfer of audio data.

Parameters

- `base` – I2S base pointer.
- `handle` – pointer to handle structure.
- `dmaHandle` – pointer to dma handle structure.
- `callback` – function to be called back when transfer is done or fails.
- `userData` – pointer to data passed to callback.

```
status_t I2S_TxTransferSendDMA(I2S_Type *base, i2s_dma_handle_t *handle, i2s_transfer_t transfer)
```

Begins or queue sending of the given data.

Parameters

- `base` – I2S base pointer.
- `handle` – pointer to handle structure.
- `transfer` – data buffer.

Return values

- `kStatus_Success` –
- `kStatus_I2S_Busy` – if all queue slots are occupied with unsent buffers.

```
void I2S_TransferAbortDMA(I2S_Type *base, i2s_dma_handle_t *handle)
```

Aborts transfer of data.

Parameters

- `base` – I2S base pointer.
- `handle` – pointer to handle structure.

```
void I2S_RxTransferCreateHandleDMA(I2S_Type *base, i2s_dma_handle_t *handle, dma_handle_t *dmaHandle, i2s_dma_transfer_callback_t callback, void *userData)
```

Initializes handle for reception of audio data.

Parameters

- `base` – I2S base pointer.
- `handle` – pointer to handle structure.

- dmaHandle – pointer to dma handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

status_t I2S_RxTransferReceiveDMA(I2S_Type *base, *i2s_dma_handle_t* *handle, *i2s_transfer_t* transfer)

Begins or queue reception of data into given buffer.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- transfer – data buffer.

Return values

- kStatus_Success –
- kStatus_I2S_Busy – if all queue slots are occupied with buffers which are not full.

void I2S_DMACallback(*dma_handle_t* *handle, void *userData, bool transferDone, uint32_t tcds)

Invoked from DMA interrupt handler.

Parameters

- handle – pointer to DMA handle structure.
- userData – argument for user callback.
- transferDone – if transfer was done.
- tcds –

void I2S_TransferInstallLoopDMADescriptorMemory(*i2s_dma_handle_t* *handle, void *dmaDescriptorAddr, size_t dmaDescriptorNum)

Install DMA descriptor memory for loop transfer only.

This function used to register DMA descriptor memory for the i2s loop dma transfer.

It must be called before I2S_TransferSendLoopDMA/I2S_TransferReceiveLoopDMA and after I2S_RxTransferCreateHandleDMA/I2S_TxTransferCreateHandleDMA.

User should be take care about the address of DMA descriptor pool which required align with 16BYTE at least.

Parameters

- handle – Pointer to i2s DMA transfer handle.
- dmaDescriptorAddr – DMA descriptor start address.
- dmaDescriptorNum – DMA descriptor number.

status_t I2S_TransferSendLoopDMA(I2S_Type *base, *i2s_dma_handle_t* *handle, *i2s_transfer_t* *xfer, uint32_t loopTransferCount)

Send link transfer data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

This function support loop transfer, such as A->B->...->A, the loop transfer chain will be converted into a chain of descriptor and submit to dma. Application must be aware of that the more counts of the loop transfer, then more DMA descriptor memory required, user can use function I2S_InstallDMADescriptorMemory to register the dma descriptor memory.

As the DMA support maximum 1024 transfer count, so application must be aware of that this transfer function support maximum 1024 samples in each transfer; otherwise assert error or error status will be returned. Once the loop transfer start, application can use function `I2S_TransferAbortDMA` to stop the loop transfer.

Parameters

- `base` – I2S peripheral base address.
- `handle` – Pointer to `usart_dma_handle_t` structure.
- `xfer` – I2S DMA transfer structure. See `i2s_transfer_t`.
- `loopTransferCount` – loop count

Return values

`kStatus_Success` –

`status_t` `I2S_TransferReceiveLoopDMA(I2S_Type *base, i2s_dma_handle_t *handle, i2s_transfer_t *xfer, uint32_t loopTransferCount)`

Receive link transfer data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

This function support loop transfer, such as `A->B->...->A`, the loop transfer chain will be converted into a chain of descriptor and submit to dma. Application must be aware of that the more counts of the loop transfer, then more DMA descriptor memory required, user can use function `I2S_InstallDMADescriptorMemory` to register the dma descriptor memory.

As the DMA support maximum 1024 transfer count, so application must be aware of that this transfer function support maximum 1024 samples in each transfer; otherwise assert error or error status will be returned. Once the loop transfer start, application can use function `I2S_TransferAbortDMA` to stop the loop transfer.

Parameters

- `base` – I2S peripheral base address.
- `handle` – Pointer to `usart_dma_handle_t` structure.
- `xfer` – I2S DMA transfer structure. See `i2s_transfer_t`.
- `loopTransferCount` – loop count

Return values

`kStatus_Success` –

`FSL_I2S_DMA_DRIVER_VERSION`

I2S DMA driver version 2.3.3.

`typedef struct i2s_dma_handle i2s_dma_handle_t`

Members not to be accessed / modified outside of the driver.

`typedef void (*i2s_dma_transfer_callback_t)(I2S_Type *base, i2s_dma_handle_t *handle, status_t completionStatus, void *userData)`

Callback function invoked from DMA API on completion.

Param base

I2S base pointer.

Param handle

pointer to I2S transaction.

Param completionStatus

status of the transaction.

Param userData

optional pointer to user arguments data.

```
struct _i2s_dma_handle
#include <fsl_i2s_dma.h> i2s dma handle
```

Public Members

```
uint32_t state
    Internal state of I2S DMA transfer

uint8_t bytesPerFrame
    bytes per frame

i2s_dma_transfer_callback_t completionCallback
    Callback function pointer

void *userData
    Application data passed to callback

dma_handle_t *dmaHandle
    DMA handle

volatile i2s_transfer_t i2sQueue[(4U)]
    Transfer queue storing transfer buffers

volatile uint8_t queueUser
    Queue index where user's next transfer will be stored

volatile uint8_t queueDriver
    Queue index of buffer actually used by the driver

dma_descriptor_t *i2sLoopDMADescriptor
    descriptor pool pointer

size_t i2sLoopDMADescriptorNum
    number of descriptor in descriptors pool
```

2.28 I2S Driver

```
void I2S_TxInit(I2S_Type *base, const i2s_config_t *config)
```

Initializes the FLEXCOMM peripheral for I2S transmit functionality.

Ungates the FLEXCOMM clock and configures the module for I2S transmission using a configuration structure. The configuration structure can be custom filled or set with default values by I2S_TxGetDefaultConfig().

Note: This API should be called at the beginning of the application to use the I2S driver.

Parameters

- base – I2S base pointer.
- config – pointer to I2S configuration structure.

```
void I2S_RxInit(I2S_Type *base, const i2s_config_t *config)
```

Initializes the FLEXCOMM peripheral for I2S receive functionality.

Ungates the FLEXCOMM clock and configures the module for I2S receive using a configuration structure. The configuration structure can be custom filled or set with default values by I2S_RxGetDefaultConfig().

Note: This API should be called at the beginning of the application to use the I2S driver.

Parameters

- base – I2S base pointer.
- config – pointer to I2S configuration structure.

void I2S_TxGetDefaultConfig(*i2s_config_t* *config)

Sets the I2S Tx configuration structure to default values.

This API initializes the configuration structure for use in I2S_TxInit(). The initialized structure can remain unchanged in I2S_TxInit(), or it can be modified before calling I2S_TxInit(). Example:

```
i2s_config_t config;
I2S_TxGetDefaultConfig(&config);
```

Default values:

```
config->masterSlave = kI2S_MasterSlaveNormalMaster;
config->mode = kI2S_ModeI2sClassic;
config->rightLow = false;
config->leftJust = false;
config->pdmData = false;
config->sckPol = false;
config->wsPol = false;
config->divider = 1;
config->oneChannel = false;
config->dataLength = 16;
config->frameLength = 32;
config->position = 0;
config->watermark = 4;
config->txEmptyZero = true;
config->pack48 = false;
```

Parameters

- config – pointer to I2S configuration structure.

void I2S_RxGetDefaultConfig(*i2s_config_t* *config)

Sets the I2S Rx configuration structure to default values.

This API initializes the configuration structure for use in I2S_RxInit(). The initialized structure can remain unchanged in I2S_RxInit(), or it can be modified before calling I2S_RxInit(). Example:

```
i2s_config_t config;
I2S_RxGetDefaultConfig(&config);
```

Default values:

```
config->masterSlave = kI2S_MasterSlaveNormalSlave;
config->mode = kI2S_ModeI2sClassic;
config->rightLow = false;
config->leftJust = false;
config->pdmData = false;
config->sckPol = false;
config->wsPol = false;
config->divider = 1;
config->oneChannel = false;
```

(continues on next page)

(continued from previous page)

```

config->dataLength = 16;
config->frameLength = 32;
config->position = 0;
config->watermark = 4;
config->txEmptyZero = false;
config->pack48 = false;

```

Parameters

- config – pointer to I2S configuration structure.

void I2S_Deinit(I2S_Type *base)

De-initializes the I2S peripheral.

This API gates the FLEXCOMM clock. The I2S module can't operate unless I2S_TxInit or I2S_RxInit is called to enable the clock.

Parameters

- base – I2S base pointer.

void I2S_SetBitClockRate(I2S_Type *base, uint32_t sourceClockHz, uint32_t sampleRate, uint32_t bitWidth, uint32_t channelNumbers)

Transmitter/Receiver bit clock rate configurations.

Parameters

- base – SAI base pointer.
- sourceClockHz – bit clock source frequency.
- sampleRate – audio data sample rate.
- bitWidth – audio data bitWidth.
- channelNumbers – audio channel numbers.

void I2S_TxTransferCreateHandle(I2S_Type *base, *i2s_handle_t* *handle, *i2s_transfer_callback_t* callback, void *userData)

Initializes handle for transfer of audio data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

status_t I2S_TxTransferNonBlocking(I2S_Type *base, *i2s_handle_t* *handle, *i2s_transfer_t* transfer)

Begins or queue sending of the given data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- transfer – data buffer.

Return values

- kStatus__Success –
- kStatus_I2S_Busy – if all queue slots are occupied with unsent buffers.

void I2S_TxTransferAbort(I2S_Type *base, i2s_handle_t *handle)

Aborts sending of data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

void I2S_RxTransferCreateHandle(I2S_Type *base, i2s_handle_t *handle, i2s_transfer_callback_t callback, void *userData)

Initializes handle for reception of audio data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- callback – function to be called back when transfer is done or fails.
- userData – pointer to data passed to callback.

status_t I2S_RxTransferNonBlocking(I2S_Type *base, i2s_handle_t *handle, i2s_transfer_t transfer)

Begins or queue reception of data into given buffer.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- transfer – data buffer.

Return values

- kStatus_Success –
- kStatus_I2S_Busy – if all queue slots are occupied with buffers which are not full.

void I2S_RxTransferAbort(I2S_Type *base, i2s_handle_t *handle)

Aborts receiving of data.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

status_t I2S_TransferGetCount(I2S_Type *base, i2s_handle_t *handle, size_t *count)

Returns number of bytes transferred so far.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- count – **[out]** number of bytes transferred so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – there is no non-blocking transaction currently in progress.

status_t I2S_TransferGetErrorCount(I2S_Type *base, *i2s_handle_t* *handle, *size_t* *count)

Returns number of buffer underruns or overruns.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.
- count – **[out]** number of transmit errors encountered so far by the non-blocking transaction.

Return values

- kStatus_Success –
- kStatus_NoTransferInProgress – there is no non-blocking transaction currently in progress.

static inline void I2S_Enable(I2S_Type *base)

Enables I2S operation.

Parameters

- base – I2S base pointer.

void I2S_EnableSecondaryChannel(I2S_Type *base, *uint32_t* channel, bool oneChannel, *uint32_t* position)

Enables I2S secondary channel.

Parameters

- base – I2S base pointer.
- channel – secondary channel channel number, reference `_i2s_secondary_channel`.
- oneChannel – true is treated as single channel, functionality left channel for this pair.
- position – define the location within the frame of the data, should not bigger than 0x1FFU.

static inline void I2S_DisableSecondaryChannel(I2S_Type *base, *uint32_t* channel)

Disables I2S secondary channel.

Parameters

- base – I2S base pointer.
- channel – secondary channel channel number, reference `_i2s_secondary_channel`.

static inline void I2S_Disable(I2S_Type *base)

Disables I2S operation.

Parameters

- base – I2S base pointer.

static inline void I2S_EnableInterrupts(I2S_Type *base, *uint32_t* interruptMask)

Enables I2S FIFO interrupts.

Parameters

- base – I2S base pointer.
- interruptMask – bit mask of interrupts to enable. See `i2s_flags_t` for the set of constants that should be OR'd together to form the bit mask.

static inline void I2S_DisableInterrupts(I2S_Type *base, uint32_t interruptMask)
Disables I2S FIFO interrupts.

Parameters

- base – I2S base pointer.
- interruptMask – bit mask of interrupts to enable. See `i2s_flags_t` for the set of constants that should be OR'd together to form the bit mask.

static inline uint32_t I2S_GetEnabledInterrupts(I2S_Type *base)
Returns the set of currently enabled I2S FIFO interrupts.

Parameters

- base – I2S base pointer.

Returns

A bitmask composed of `i2s_flags_t` enumerators OR'd together to indicate the set of enabled interrupts.

status_t I2S_EmptyTxFifo(I2S_Type *base)
Flush the valid data in TX fifo.

Parameters

- base – I2S base pointer.

Returns

`kStatus_Fail` empty TX fifo failed, `kStatus_Success` empty tx fifo success.

void I2S_TxHandleIRQ(I2S_Type *base, *i2s_handle_t* *handle)
Invoked from interrupt handler when transmit FIFO level decreases.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

void I2S_RxHandleIRQ(I2S_Type *base, *i2s_handle_t* *handle)
Invoked from interrupt handler when receive FIFO level decreases.

Parameters

- base – I2S base pointer.
- handle – pointer to handle structure.

FSL_I2S_DRIVER_VERSION
I2S driver version 2.3.2.

`_i2s_status` I2S status codes.

Values:

enumerator `kStatus_I2S_BufferComplete`
Transfer from/into a single buffer has completed

enumerator `kStatus_I2S_Done`
All buffers transfers have completed

enumerator `kStatus_I2S_Busy`
Already performing a transfer and cannot queue another buffer

enum _i2s_flags
I2S flags.

Note: These enums are meant to be OR'd together to form a bit mask.

Values:

enumerator kI2S_TxErrorFlag
TX error interrupt

enumerator kI2S_TxLevelFlag
TX level interrupt

enumerator kI2S_RxErrorFlag
RX error interrupt

enumerator kI2S_RxLevelFlag
RX level interrupt

enum _i2s_master_slave
Master / slave mode.

Values:

enumerator kI2S_MasterSlaveNormalSlave
Normal slave

enumerator kI2S_MasterSlaveWsSyncMaster
WS synchronized master

enumerator kI2S_MasterSlaveExtSckMaster
Master using existing SCK

enumerator kI2S_MasterSlaveNormalMaster
Normal master

enum _i2s_mode
I2S mode.

Values:

enumerator kI2S_ModeI2sClassic
I2S classic mode

enumerator kI2S_ModeDspWs50
DSP mode, WS having 50% duty cycle

enumerator kI2S_ModeDspWsShort
DSP mode, WS having one clock long pulse

enumerator kI2S_ModeDspWsLong
DSP mode, WS having one data slot long pulse

_i2s_secondary_channel I2S secondary channel.

Values:

enumerator kI2S_SecondaryChannel1
secondary channel 1

enumerator `ki2S_SecondaryChannel2`
secondary channel 2

enumerator `ki2S_SecondaryChannel3`
secondary channel 3

typedef enum `_i2s_flags` `i2s_flags_t`
I2S flags.

Note: These enums are meant to be OR'd together to form a bit mask.

typedef enum `_i2s_master_slave` `i2s_master_slave_t`
Master / slave mode.

typedef enum `_i2s_mode` `i2s_mode_t`
I2S mode.

typedef struct `_i2s_config` `i2s_config_t`
I2S configuration structure.

typedef struct `_i2s_transfer` `i2s_transfer_t`
Buffer to transfer from or receive audio data into.

typedef struct `_i2s_handle` `i2s_handle_t`
Transactional state of the initialized transfer or receive I2S operation.

typedef void (`*i2s_transfer_callback_t`)(`I2S_Type *base`, `i2s_handle_t *handle`, `status_t` `completionStatus`, void `*userData`)

Callback function invoked from transactional API on completion of a single buffer transfer.

Param base
I2S base pointer.

Param handle
pointer to I2S transaction.

Param completionStatus
status of the transaction.

Param userData
optional pointer to user arguments data.

`I2S_NUM_BUFFERS`
Number of buffers .

struct `_i2s_config`
`#include <fsl_i2s.h>` I2S configuration structure.

Public Members

`i2s_master_slave_t` `masterSlave`
Master / slave configuration

`i2s_mode_t` `mode`
I2S mode

bool `rightLow`
Right channel data in low portion of FIFO

bool leftJust

Left justify data in FIFO

bool pdmData

Data source is the D-Mic subsystem

bool sckPol

SCK polarity

bool wsPol

WS polarity

uint16_t divider

Flexcomm function clock divider (1 - 4096)

bool oneChannel

true mono, false stereo

uint8_t dataLength

Data length (4 - 32)

uint16_t frameLength

Frame width (4 - 512)

uint16_t position

Data position in the frame

uint8_t watermark

FIFO trigger level

bool txEmptyZero

Transmit zero when buffer becomes empty or last item

bool pack48

Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)

struct `_i2s_transfer`

#include <fsl_i2s.h> Buffer to transfer from or receive audio data into.

Public Members

uint8_t *data

Pointer to data buffer.

size_t dataSize

Buffer size in bytes.

struct `_i2s_handle`

#include <fsl_i2s.h> Members not to be accessed / modified outside of the driver.

Public Members

volatile uint32_t state

State of transfer

i2s_transfer_callback_t completionCallback

Callback function pointer

`void *userData`
Application data passed to callback

`bool oneChannel`
true mono, false stereo

`uint8_t dataLength`
Data length (4 - 32)

`bool pack48`
Packing format for 48-bit data (false - 24 bit values, true - alternating 32-bit and 16-bit values)

`uint8_t watermark`
FIFO trigger level

`bool useFifo48H`
When dataLength 17-24: true use FIFOWR48H, false use FIFOWR

`volatile i2s_transfer_t i2sQueue[(4U)]`
Transfer queue storing transfer buffers

`volatile uint8_t queueUser`
Queue index where user's next transfer will be stored

`volatile uint8_t queueDriver`
Queue index of buffer actually used by the driver

`volatile uint32_t errorCount`
Number of buffer underruns/overruns

`volatile uint32_t transferCount`
Number of bytes transferred

2.29 IMU: Inter CPU Messaging Unit

`status_t IMU_Init(imu_link_t link)`

Initializes the IMU module.

This function sets IMU to initialized state, including:

- Flush the send FIFO.
- Unlock the send FIFO.
- Set the water mark to (IMU_MAX_MSG_FIFO_WATER_MARK)
- Flush the receive FIFO.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return `kStatus_Timeout` if flushing the receive FIFO takes too long.

Parameters

- `link` – IMU link.

Return values

- `kStatus_Success` – The IMU was initialized successfully.
- `kStatus_InvalidArgument` – Invalid link parameter.

- kStatus_Timeout – Timeout occurred while flushing the receive FIFO.

Returns

status_t

void IMU_Deinit(imu_link_t link)

De-initializes the IMU module.

Parameters

- link – IMU link.

static inline void IMU_WriteMsg(imu_link_t link, uint32_t msg)

Write one message to TX FIFO.

This function writes message to the TX FIFO, user need to make sure there is empty space in the TX FIFO, and TX FIFO not locked before calling this function.

Parameters

- link – IMU link.
- msg – The message to send.

static inline uint32_t IMU_ReadMsg(imu_link_t link)

Read one message from RX FIFO.

User need to make sure there is available message in the RX FIFO.

Parameters

- link – IMU link.

Returns

The message.

int32_t IMU_SendMsgsBlocking(imu_link_t link, const uint32_t *msgs, int32_t msgCount, bool lockSendFifo)

Blocking to send messages.

This function blocks until all messages have been filled to TX FIFO.

- If the TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED.
- If TX FIFO not locked, this function waits the available empty slot in TX FIFO, and fills the message to TX FIFO.
- To lock TX FIFO after filling all messages, set lockSendFifo to true.

If IMU_BUSY_POLL_COUNT is defined and non-zero, the function will timeout after the specified number of polling iterations and return IMU_ERR_TIMEOUT if waiting for FIFO space takes too long.

Parameters

- link – IMU link.
- msgs – The messages to send.
- msgCount – Message count, one message is a 32-bit word.
- lockSendFifo – If set to true, the TX FIFO is locked after all messages filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns IMU_ERR_TX_FIFO_LOCKED. If a timeout occurs while waiting for FIFO space, it returns IMU_ERR_TIMEOUT. Otherwise, this function returns the actual message count sent out, which

equals `msgCount` because this function is blocking until all messages have been filled into TX FIFO or a timeout occurs.

```
int32_t IMU_TrySendMsgs(imu_link_t link, const uint32_t *msgs, int32_t msgCount, bool
                      lockSendFifo)
```

Try to send messages.

This function is similar with `IMU_SendMsgsBlocking`, the difference is, this function tries to send as many as possible, if there is not enough empty slot in TX FIFO, this function fills messages to available empty slots and returns how many messages have been filled.

- If the TX FIFO is locked, this function returns `IMU_ERR_TX_FIFO_LOCKED`.
- If TX FIFO not locked, this function fills messages to TX FIFO empty slot, and returns how many messages have been filled.
- If `lockSendFifo` is set to true, TX FIFO is locked after all messages have been filled to TX FIFO. In other word, TX FIFO is locked if the function return value equals `msgCount`, when `lockSendFifo` set to true.

Parameters

- `link` – IMU link.
- `msgs` – The messages to send.
- `msgCount` – Message count, one message is a 32-bit word.
- `lockSendFifo` – If set to true, the TX FIFO is locked after all messages filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns `IMU_ERR_TX_FIFO_LOCKED`, otherwise, this function returns the actual message count sent out.

```
int32_t IMU_TryReceiveMsgs(imu_link_t link, uint32_t *msgs, int32_t desiredMsgCount, bool
                          *needAckLock)
```

Try to receive messages.

This function tries to read messages from RX FIFO. It reads the messages already exists in RX FIFO and returns the actual read count.

- If the RX FIFO has enough messages, this function reads the messages and returns.
- If the RX FIFO does not have enough messages, this function the messages in RX FIFO and returns the actual read count.
- During message reading, if RX FIFO is empty and locked, in this case peer CPU will not send message until current CPU send lock ack message. Then this function returns the message count actually received, and sets `needAckLock` to true to inform upper layer.

Parameters

- `link` – IMU link.
- `msgs` – The buffer to read messages.
- `desiredMsgCount` – Desired read count, one message is a 32-bit word.
- `needAckLock` – Upper layer should always check this value. When this is set to true by this function, upper layer should send lock ack message to peer CPU.

Returns

Count of messages actually received.

```
int32_t IMU_ReceiveMsgsBlocking(imu_link_t link, uint32_t *msgs, int32_t desiredMsgCount,  
                               bool *needAckLock)
```

Blocking to receive messages.

This function blocks until all desired messages have been received or the RX FIFO is locked.

- If the RX FIFO has enough messages, this function reads the messages and returns.
- If the RX FIFO does not have enough messages, this function waits for the new messages.
- During message reading, if RX FIFO is empty and locked, in this case peer CPU will not send message until current CPU send lock ack message. Then this function returns the message count actually received, and sets `needAckLock` to true to inform upper layer.

Parameters

- `link` – IMU link.
- `msgs` – The buffer to read messages.
- `desiredMsgCount` – Desired read count, one message is a 32-bit word.
- `needAckLock` – Upper layer should always check this value. When this is set to true by this function, upper layer should send lock ack message to peer CPU.

Returns

Count of messages actually received.

```
int32_t IMU_SendMsgPtrBlocking(imu_link_t link, uint32_t msgPtr, bool lockSendFifo)
```

Blocking to send messages pointer.

Compared with `IMU_SendMsgsBlocking`, this function fills message pointer to TX FIFO, but not the message content.

This function blocks until the message pointer is filled to TX FIFO.

- If the TX FIFO is locked, this function returns `IMU_ERR_TX_FIFO_LOCKED`.
- If TX FIFO not locked, this function waits the available empty slot in TX FIFO, and fills the message pointer to TX FIFO.
- To lock TX FIFO after filling the message pointer, set `lockSendFifo` to true.

If `IMU_BUSY_POLL_COUNT` is defined and non-zero, the function will timeout after the specified number of polling iterations and return `IMU_ERR_TIMEOUT` if waiting for FIFO space takes too long.

Parameters

- `link` – IMU link.
- `msgPtr` – The buffer pointer to message to send.
- `lockSendFifo` – If set to true, the TX FIFO is locked after message pointer filled to TX FIFO.

Returns

If TX FIFO is locked, this function returns `IMU_ERR_TX_FIFO_LOCKED`. If a timeout occurs while waiting for FIFO space, it returns `IMU_ERR_TIMEOUT`. Otherwise, this function returns 0 to indicate success.

```
static inline void IMU__LockSendFifo(imu_link_t link, bool lock)
```

Lock or unlock the TX FIFO.

Parameters

- link – IMU link.
- lock – Use true to lock the FIFO, use false to unlock.

```
void IMU__FlushSendFifo(imu_link_t link)
```

Flush the send FIFO.

Flush all messages in send FIFO.

Parameters

- link – IMU link.

```
static inline void IMU__SetSendFifoWaterMark(imu_link_t link, uint8_t waterMark)
```

Set send FIFO water mark.

The water mark must be less than IMU_MAX_MSG_FIFO_WATER_MARK, i.e. $0 < \text{waterMark} \leq \text{IMU_MAX_MSG_FIFO_WATER_MARK}$.

Parameters

- link – IMU link.
- waterMark – Send FIFO water mark.

```
static inline uint32_t IMU__GetReceivedMsgCount(imu_link_t link)
```

Get the message count in receive FIFO.

Parameters

- link – IMU link.

Returns

The message count in receive FIFO.

```
static inline uint32_t IMU__GetSendFifoEmptySpace(imu_link_t link)
```

Get the empty slot in send FIFO.

Parameters

- link – IMU link.

Returns

The empty slot count in send FIFO.

```
uint32_t IMU__GetStatusFlags(imu_link_t link)
```

Gets the IMU status flags.

Parameters

- link – IMU link.

Returns

Bit mask of the IMU status flags, see `_imu_status_flags`.

```
static inline void IMU__ClearPendingInterrupts(imu_link_t link, uint32_t mask)
```

Clear the IMU IRQ.

Parameters

- link – IMU link.
- mask – Bit mask of the interrupts to clear, see `_imu_interrupts`.

FSL_IMU_DRIVER_VERSION

IMU driver version.

enum _imu_status_flags

IMU status flags. .

Values:

enumerator kIMU_TxFifoEmpty

enumerator kIMU_TxFifoFull

enumerator kIMU_TxFifoAlmostFull

enumerator kIMU_TxFifoLocked

enumerator kIMU_RxFifoEmpty

enumerator kIMU_RxFifoFull

enumerator kIMU_RxFifoAlmostFull

enumerator kIMU_RxFifoLocked

enum _imu_interrupts

IMU interrupt. .

Values:

enumerator kIMU_RxMsgReadyInterrupt

enumerator kIMU_TxFifoSpaceAvailableInterrupt

IMU_MSG_FIFO_STATUS_MSG_FIFO_LOCKED_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_ALMOST_FULL_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_FULL_MASK

IMU_MSG_FIFO_STATUS_MSG_FIFO_EMPTY_MASK

IMU_MSG_FIFO_STATUS_MSG_COUNT_MASK

IMU_MSG_FIFO_STATUS_MSG_COUNT_SHIFT

IMU_MSG_FIFO_STATUS_WR_PTR_MASK

IMU_MSG_FIFO_STATUS_WR_PTR_SHIFT

IMU_MSG_FIFO_STATUS_RD_PTR_MASK

IMU_MSG_FIFO_STATUS_RD_PTR_SHIFT

IMU_MSG_FIFO_CNTL_MSG_RDY_INT_CLR_MASK

IMU_MSG_FIFO_CNTL_SP_AV_INT_CLR_MASK

IMU_MSG_FIFO_CNTL_FIFO_FLUSH_MASK

IMU_MSG_FIFO_CNTL_WAIT_FOR_ACK_MASK

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK_MASK

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK_SHIFT

IMU_MSG_FIFO_CNTL_FIFO_FULL_WATERMARK(x)

IMU_WR_MSG(link, msg)

IMU_RD_MSG(link)

IMU_RX_FIFO_LOCKED(link)

IMU_TX_FIFO_LOCKED(link)

IMU_TX_FIFO_ALMOST_FULL(link)

IMU_RX_FIFO_EMPTY(link)

Get Rx FIFO empty status.

IMU_LOCK_TX_FIFO(link)

IMU_UNLOCK_TX_FIFO(link)

IMU_RX_FIFO_MSG_COUNT(link)

IMU_TX_FIFO_MSG_COUNT(link)

IMU_RX_FIFO_MSG_COUNT_FROM_STATUS(rxFifoStatus)

IMU_RX_FIFO_LOCKED_FROM_STATUS(rxFifoStatus)

IMU_TX_FIFO_STATUS(link)

IMU_RX_FIFO_STATUS(link)

IMU_TX_FIFO_CNTL(link)

IMU_ERR_TX_FIFO_LOCKED

IMU driver returned error value.

IMU_ERR_TIMEOUT

IMU driver returned error value timeout.

IMU_MSG_FIFO_MAX_COUNT

Maximum message numbers in FIFO.

IMU_MAX_MSG_FIFO_WATER_MARK

Maximum message FIFO water mark.

IMU_FIFO_SW_WRAPAROUND(ptr)

IMU_WR_PTR(link)

IMU_RD_PTR(link)

IMU_BUSY_POLL_COUNT

Maximum polling iterations for IMU waiting loops.

This parameter defines the maximum number of iterations for any polling loop in the IMU driver code before timing out and returning an error.

It applies to all waiting loops in IMU driver.

This is a count of loop iterations, not a time-based value.

If defined as 0, polling loops will continue indefinitely until their exit condition is met, which could potentially cause the system to hang if sensors don't respond or if communication interfaces fail.

struct IMU_Type

#include <fsl_imu.h> IMU register structure.

2.30 INPUTMUX: Input Multiplexing Driver

FSL_INPUTMUX_DRIVER_VERSION

Group interrupt driver version for SDK.

enum _inputmux_connection_t

INPUTMUX connections type.

Values:

enumerator kINPUTMUX_Gpio3Inp0ToSct0

SCT INMUX.

enumerator kINPUTMUX_Gpio4Inp1ToSct0

enumerator kINPUTMUX_Gpio22Inp2ToSct0

enumerator kINPUTMUX_Gpio23Inp3ToSct0

enumerator kINPUTMUX_Gpio26Inp4ToSct0

enumerator kINPUTMUX_Gpio27Inp5ToSct0

enumerator kINPUTMUX_Gpio35Inp6ToSct0

enumerator kINPUTMUX_Gpio36Inp7ToSct0

enumerator kINPUTMUX_Ctimer0Mat0ToSct0

enumerator kINPUTMUX_Ctimer1Mat0ToSct0

enumerator kINPUTMUX_Ctimer2Mat0ToSct0

enumerator kINPUTMUX_Ctimer3Mat0ToSct0

enumerator kINPUTMUX_GpioIntBmatchToSct0

enumerator kINPUTMUX_SharedI2s0SclkToSct0

enumerator kINPUTMUX_SharedI2s1SclkToSct0

enumerator kINPUTMUX_SharedI2s0WsToSct0

enumerator kINPUTMUX_SharedI2s1WsToSct0

enumerator kINPUTMUX_MclkToSct0

enumerator kINPUTMUX_ArmTxevToSct0

enumerator kINPUTMUX_DebugHaltedToSct0

Pin Interrupt.

enumerator kINPUTMUX_GpioPort0Pin0ToPintsel

enumerator kINPUTMUX_GpioPort0Pin1ToPintsel

enumerator kINPUTMUX_GpioPort0Pin2ToPintsel

enumerator kINPUTMUX_GpioPort0Pin3ToPintsel

enumerator kINPUTMUX_GpioPort0Pin4ToPintsel

enumerator kINPUTMUX_GpioPort0Pin5ToPintsel

enumerator kINPUTMUX_GpioPort0Pin6ToPintsel
enumerator kINPUTMUX_GpioPort0Pin7ToPintsel
enumerator kINPUTMUX_GpioPort0Pin8ToPintsel
enumerator kINPUTMUX_GpioPort0Pin9ToPintsel
enumerator kINPUTMUX_GpioPort0Pin10ToPintsel
enumerator kINPUTMUX_GpioPort0Pin11ToPintsel
enumerator kINPUTMUX_GpioPort0Pin12ToPintsel
enumerator kINPUTMUX_GpioPort0Pin13ToPintsel
enumerator kINPUTMUX_GpioPort0Pin14ToPintsel
enumerator kINPUTMUX_GpioPort0Pin15ToPintsel
enumerator kINPUTMUX_GpioPort0Pin16ToPintsel
enumerator kINPUTMUX_GpioPort0Pin17ToPintsel
enumerator kINPUTMUX_GpioPort0Pin18ToPintsel
enumerator kINPUTMUX_GpioPort0Pin19ToPintsel
enumerator kINPUTMUX_GpioPort0Pin20ToPintsel
enumerator kINPUTMUX_GpioPort0Pin21ToPintsel
enumerator kINPUTMUX_GpioPort0Pin22ToPintsel
enumerator kINPUTMUX_GpioPort0Pin23ToPintsel
enumerator kINPUTMUX_GpioPort0Pin24ToPintsel
enumerator kINPUTMUX_GpioPort0Pin25ToPintsel
enumerator kINPUTMUX_GpioPort0Pin26ToPintsel
enumerator kINPUTMUX_GpioPort0Pin27ToPintsel
enumerator kINPUTMUX_GpioPort0Pin28ToPintsel
enumerator kINPUTMUX_GpioPort0Pin29ToPintsel
enumerator kINPUTMUX_GpioPort0Pin30ToPintsel
enumerator kINPUTMUX_GpioPort0Pin31ToPintsel
enumerator kINPUTMUX_GpioPort1Pin0ToPintsel
enumerator kINPUTMUX_GpioPort1Pin1ToPintsel
enumerator kINPUTMUX_GpioPort1Pin2ToPintsel
enumerator kINPUTMUX_GpioPort1Pin3ToPintsel
enumerator kINPUTMUX_GpioPort1Pin4ToPintsel
enumerator kINPUTMUX_GpioPort1Pin5ToPintsel
enumerator kINPUTMUX_GpioPort1Pin6ToPintsel

enumerator kINPUTMUX_GpioPort1Pin7ToPintsel

enumerator kINPUTMUX_GpioPort1Pin8ToPintsel

enumerator kINPUTMUX_GpioPort1Pin9ToPintsel

enumerator kINPUTMUX_GpioPort1Pin10ToPintsel

enumerator kINPUTMUX_GpioPort1Pin11ToPintsel

enumerator kINPUTMUX_GpioPort1Pin12ToPintsel

enumerator kINPUTMUX_GpioPort1Pin13ToPintsel

enumerator kINPUTMUX_GpioPort1Pin14ToPintsel

enumerator kINPUTMUX_GpioPort1Pin15ToPintsel

enumerator kINPUTMUX_GpioPort1Pin16ToPintsel

enumerator kINPUTMUX_GpioPort1Pin17ToPintsel

enumerator kINPUTMUX_GpioPort1Pin18ToPintsel

enumerator kINPUTMUX_GpioPort1Pin19ToPintsel

enumerator kINPUTMUX_GpioPort1Pin20ToPintsel

enumerator kINPUTMUX_GpioPort1Pin21ToPintsel

enumerator kINPUTMUX_GpioPort1Pin22ToPintsel

enumerator kINPUTMUX_GpioPort1Pin23ToPintsel

enumerator kINPUTMUX_GpioPort1Pin24ToPintsel

enumerator kINPUTMUX_GpioPort1Pin25ToPintsel

enumerator kINPUTMUX_GpioPort1Pin26ToPintsel

enumerator kINPUTMUX_GpioPort1Pin27ToPintsel

enumerator kINPUTMUX_GpioPort1Pin28ToPintsel

enumerator kINPUTMUX_GpioPort1Pin29ToPintsel

enumerator kINPUTMUX_GpioPort1Pin30ToPintsel

enumerator kINPUTMUX_GpioPort1Pin31ToPintsel

Frequency measure.

enumerator kINPUTMUX_SysoscToFreqmeas

enumerator kINPUTMUX_SfroToFreqmeas

enumerator kINPUTMUX_FfroToFreqmeas

enumerator kINPUTMUX_LposcToFreqmeas

enumerator kINPUTMUX_Xtal32kToFreqmeas

enumerator kINPUTMUX_C0FrHclkToFreqmeas

enumerator kINPUTMUX_FreqmeGpioClkInToFreqmeas

enumerator kINPUTMUX__T3pllMcuFlexspiClkToFreqmeas

enumerator kINPUTMUX__TddrMcuFlexspiClkToFreqmeas

enumerator kINPUTMUX__TddrMcuEnetClkToFreqmeas

enumerator kINPUTMUX__TcpuMcuFlexspiClkToFreqmeas

enumerator kINPUTMUX__Nco32kToFreqmeas

enumerator kINPUTMUX__PmuFclkToFreqmeas

enumerator kINPUTMUX__Osc32kClk1hzToFreqmeas

enumerator kINPUTMUX__Osc32kClk1khzToFreqmeas

enumerator kINPUTMUX__LcdFclkToFreqmeas

enumerator kINPUTMUX__Flexcomm0FclkToFreqmeas

enumerator kINPUTMUX__DmicFclkToFreqmeas

enumerator kINPUTMUX__Flexspi0FclkToFreqmeas

enumerator kINPUTMUX__TcpuMcuClkToFreqmeas

enumerator kINPUTMUX__AvpllCh2ClkoutToFreqmeas

CTmier0 capture input mux.

enumerator kINPUTMUX__Gpio0Inp0ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio1Inp1ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio12Inp2ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio13Inp3ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio14Inp4ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio21Inp5ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio24Inp6ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio25Inp7ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio37Inp8ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio38Inp9ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio39Inp10ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio51Inp11ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio52Inp12ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio53Inp13ToTimer0CaptureChannels

enumerator kINPUTMUX__Gpio54Inp14ToTimer0CaptureChannels

enumerator kINPUTMUX__SharedI2s0WsToTimer0CaptureChannels

enumerator kINPUTMUX__SharedI2s1WsToTimer0CaptureChannels

enumerator kINPUTMUX__BtuHostTrigger0ToTimer0CaptureChannels

enumerator kINPUTMUX__BtuHostTrigger1ToTimer0CaptureChannels

enumerator kINPUTMUX__BtuHostTrigger2ToTimer0CaptureChannels

enumerator kINPUTMUX__FlexcommDmaDone0ToTimer0CaptureChannels

enumerator kINPUTMUX__FlexcommDmaDone1ToTimer0CaptureChannels

enumerator kINPUTMUX__FlexcommDmaCmpltdone0ToTimer0CaptureChannels

enumerator kINPUTMUX__FlexcommDmaCmpltdone1ToTimer0CaptureChannels

CTmrier1 capture input mux.

enumerator kINPUTMUX__Gpio0Inp0ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio1Inp1ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio12Inp2ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio13Inp3ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio14Inp4ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio21Inp5ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio24Inp6ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio25Inp7ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio37Inp8ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio38Inp9ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio39Inp10ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio51Inp11ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio52Inp12ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio53Inp13ToTimer1CaptureChannels

enumerator kINPUTMUX__Gpio54Inp14ToTimer1CaptureChannels

enumerator kINPUTMUX__SharedI2s0WsToTimer1CaptureChannels

enumerator kINPUTMUX__SharedI2s1WsToTimer1CaptureChannels

enumerator kINPUTMUX__EnetToTimer1CaptureChannels

enumerator kINPUTMUX__BtuHostTrigger0ToTimer1CaptureChannels

enumerator kINPUTMUX__BtuHostTrigger1ToTimer1CaptureChannels

enumerator kINPUTMUX__BtuHostTrigger2ToTimer1CaptureChannels

CTmrier2 capture input mux.

enumerator kINPUTMUX__Gpio0Inp0ToTimer2CaptureChannels

enumerator kINPUTMUX__Gpio1Inp1ToTimer2CaptureChannels

enumerator kINPUTMUX__Gpio12Inp2ToTimer2CaptureChannels

enumerator kINPUTMUX__Gpio13Inp3ToTimer2CaptureChannels

enumerator kINPUTMUX__Gpio14Inp4ToTimer2CaptureChannels

enumerator kINPUTMUX_Gpio21Inp5ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio24Inp6ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio25Inp7ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio37Inp8ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio38Inp9ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio39Inp10ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio51Inp11ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio52Inp12ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio53Inp13ToTimer2CaptureChannels
enumerator kINPUTMUX_Gpio54Inp14ToTimer2CaptureChannels
enumerator kINPUTMUX_SharedI2s0WsToTimer2CaptureChannels
enumerator kINPUTMUX_SharedI2s1WsToTimer2CaptureChannels
enumerator kINPUTMUX_BtuHostTrigger0ToTimer2CaptureChannels
enumerator kINPUTMUX_BtuHostTrigger1ToTimer2CaptureChannels
enumerator kINPUTMUX_BtuHostTrigger2ToTimer2CaptureChannels
enumerator kINPUTMUX_FlexcommDmaDone0ToTimer2CaptureChannels
enumerator kINPUTMUX_FlexcommDmaDone1ToTimer2CaptureChannels
enumerator kINPUTMUX_FlexcommDmaCmpltdDone0ToTimer2CaptureChannels
enumerator kINPUTMUX_FlexcommDmaCmpltdDone1ToTimer2CaptureChannels
CTmier3 capture input mux.

enumerator kINPUTMUX_Gpio0Inp0ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio1Inp1ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio12Inp2ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio13Inp3ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio14Inp4ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio21Inp5ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio24Inp6ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio25Inp7ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio37Inp8ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio38Inp9ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio39Inp10ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio51Inp11ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio52Inp12ToTimer3CaptureChannels

enumerator kINPUTMUX_Gpio53Inp13ToTimer3CaptureChannels
enumerator kINPUTMUX_Gpio54Inp14ToTimer3CaptureChannels
enumerator kINPUTMUX_SharedI2s0WsToTimer3CaptureChannels
enumerator kINPUTMUX_SharedI2s1WsToTimer3CaptureChannels
enumerator kINPUTMUX_BtuHostTrigger0ToTimer3CaptureChannels
enumerator kINPUTMUX_BtuHostTrigger1ToTimer3CaptureChannels
enumerator kINPUTMUX_BtuHostTrigger2ToTimer3CaptureChannels
DMA0 ITRIG.
enumerator kINPUTMUX_NsGpioPint0ToDma0
enumerator kINPUTMUX_NsGpioPint1ToDma0
enumerator kINPUTMUX_NsGpioPint2ToDma0
enumerator kINPUTMUX_NsGpioPint3ToDma0
enumerator kINPUTMUX_Ctimer0M0ToDma0
enumerator kINPUTMUX_Ctimer0M1ToDma0
enumerator kINPUTMUX_Ctimer1M0ToDma0
enumerator kINPUTMUX_Ctimer1M1ToDma0
enumerator kINPUTMUX_Ctimer2M0ToDma0
enumerator kINPUTMUX_Ctimer2M1ToDma0
enumerator kINPUTMUX_Ctimer3M0ToDma0
enumerator kINPUTMUX_Ctimer3M1ToDma0
enumerator kINPUTMUX_Dma0TrigOutAToDma0
enumerator kINPUTMUX_Dma0TrigOutBToDma0
enumerator kINPUTMUX_Dma0TrigOutCToDma0
enumerator kINPUTMUX_Dma0TrigOutDToDma0
enumerator kINPUTMUX_Sct0Dmac0ToDma0
enumerator kINPUTMUX_Sct0Dmac1ToDma0
enumerator kINPUTMUX_EnetMac0DmaReq0ToDma0
enumerator kINPUTMUX_EnetMac0DmaReq1ToDma0
enumerator kINPUTMUX_UsimDmaRxSingleToDma0
enumerator kINPUTMUX_UsimDmaTxSingleToDma0
enumerator kINPUTMUX_GauGpadc0DmaSingleToDma0
enumerator kINPUTMUX_GauGpadc1DmaSingleToDma0
enumerator kINPUTMUX_GauGpdacaDmaSingleToDma0

enumerator kINPUTMUX__GauGpdacbDmaSingleToDma0
enumerator kINPUTMUX__FlexspiRxToDma0
enumerator kINPUTMUX__FlexspiTxToDma0
enumerator kINPUTMUX__LcdRxRegToDmaSingleToDma0
enumerator kINPUTMUX__LcdTxRegToDmaSingleToDma0
DMA1 ITRIG.
enumerator kINPUTMUX__NsGpioPint0ToDma1
enumerator kINPUTMUX__NsGpioPint1ToDma1
enumerator kINPUTMUX__NsGpioPint2ToDma1
enumerator kINPUTMUX__NsGpioPint3ToDma1
enumerator kINPUTMUX__Ctimer0M0ToDma1
enumerator kINPUTMUX__Ctimer0M1ToDma1
enumerator kINPUTMUX__Ctimer1M0ToDma1
enumerator kINPUTMUX__Ctimer1M1ToDma1
enumerator kINPUTMUX__Ctimer2M0ToDma1
enumerator kINPUTMUX__Ctimer2M1ToDma1
enumerator kINPUTMUX__Ctimer3M0ToDma1
enumerator kINPUTMUX__Ctimer3M1ToDma1
enumerator kINPUTMUX__Dma1TrigOutAToDma1
enumerator kINPUTMUX__Dma1TrigOutBToDma1
enumerator kINPUTMUX__Dma1TrigOutCToDma1
enumerator kINPUTMUX__Dma1TrigOutDToDma1
enumerator kINPUTMUX__Sct0Dmac0ToDma1
enumerator kINPUTMUX__Sct0Dmac1ToDma1
enumerator kINPUTMUX__EnetMac0DmaReq0ToDma1
enumerator kINPUTMUX__EnetMac0DmaReq1ToDma1
enumerator kINPUTMUX__UsimDmaRxSingleToDma1
enumerator kINPUTMUX__UsimDmaTxSingleToDma1
enumerator kINPUTMUX__GauGpadc0DmaSingleToDma1
enumerator kINPUTMUX__GauGpadc1DmaSingleToDma1
enumerator kINPUTMUX__GauGpdacaDmaSingleToDma1
enumerator kINPUTMUX__GauGpdacbDmaSingleToDma1
enumerator kINPUTMUX__FlexspiRxToDma1

enumerator kINPUTMUX_FlexspiTxToDma1
enumerator kINPUTMUX_LcdRxRegToDmaSingleToDma1
enumerator kINPUTMUX_LcdTxRegToDmaSingleToDma1
DMA0 OTRIG.
enumerator kINPUTMUX_Dma0OtrigChannel0ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel1ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel2ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel3ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel4ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel5ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel6ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel7ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel8ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel9ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel10ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel11ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel12ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel13ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel14ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel15ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel16ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel17ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel18ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel19ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel20ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel21ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel22ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel23ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel24ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel25ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel26ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel27ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel28ToTriginChannels

enumerator kINPUTMUX_Dma0OtrigChannel29ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel30ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel31ToTriginChannels
enumerator kINPUTMUX_Dma0OtrigChannel32ToTriginChannels
DMA1 OTRIG.
enumerator kINPUTMUX_Dma1OtrigChannel0ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel1ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel2ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel3ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel4ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel5ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel6ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel7ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel8ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel9ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel10ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel11ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel12ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel13ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel14ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel15ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel16ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel17ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel18ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel19ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel20ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel21ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel22ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel23ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel24ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel25ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel26ToTriginChannels
enumerator kINPUTMUX_Dma1OtrigChannel27ToTriginChannels

enumerator kINPUTMUX__Dma1OtrigChannel28ToTriginChannels
 enumerator kINPUTMUX__Dma1OtrigChannel29ToTriginChannels
 enumerator kINPUTMUX__Dma1OtrigChannel30ToTriginChannels
 enumerator kINPUTMUX__Dma1OtrigChannel31ToTriginChannels
 enumerator kINPUTMUX__Dma1OtrigChannel32ToTriginChannels

enum __inputmux_signal__t

INPUTMUX signal enable/disable type.

Values:

enumerator kINPUTMUX__Dmac0InputTriggerPint0Ena
 DMA0 input trigger source enable.
 enumerator kINPUTMUX__Dmac0InputTriggerPint1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerPint2Ena
 enumerator kINPUTMUX__Dmac0InputTriggerPint3Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer0M0Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer0M1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer1M0Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer1M1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer2M0Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer2M1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer3M0Ena
 enumerator kINPUTMUX__Dmac0InputTriggerCtimer3M1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerDma0OutAEna
 enumerator kINPUTMUX__Dmac0InputTriggerDma0OutBEna
 enumerator kINPUTMUX__Dmac0InputTriggerDma0OutCEna
 enumerator kINPUTMUX__Dmac0InputTriggerDma0OutDEna
 enumerator kINPUTMUX__Dmac0InputTriggerSct0Dmac0Ena
 enumerator kINPUTMUX__Dmac0InputTriggerSct0Dmac1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerEnetMac0DmaReq0Ena
 enumerator kINPUTMUX__Dmac0InputTriggerEnetMac0DmaReq1Ena
 enumerator kINPUTMUX__Dmac0InputTriggerUsimDmaRxSingleEna
 enumerator kINPUTMUX__Dmac0InputTriggerUsimDmaTxSingleEna
 enumerator kINPUTMUX__Dmac0InputTriggerGauGpadc0DmaSingleEna
 enumerator kINPUTMUX__Dmac0InputTriggerGauGpadc1DmaSingleEna
 enumerator kINPUTMUX__Dmac0InputTriggerGauGpdacaDmaSingleEna

enumerator kINPUTMUX__Dmac0InputTriggerGauGpdacbDmaSingleEna
enumerator kINPUTMUX__Dmac0InputTriggerFlexspiRxEna
enumerator kINPUTMUX__Dmac0InputTriggerFlexspiTxEna
enumerator kINPUTMUX__Dmac0InputTriggerLcdRxRegToDmaSingleEna
enumerator kINPUTMUX__Dmac0InputTriggerLcdTxRegToDmaSingleEna
DMA1 input trigger source enable.
enumerator kINPUTMUX__Dmac1InputTriggerPint0Ena
enumerator kINPUTMUX__Dmac1InputTriggerPint1Ena
enumerator kINPUTMUX__Dmac1InputTriggerPint2Ena
enumerator kINPUTMUX__Dmac1InputTriggerPint3Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer0M0Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer0M1Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer1M0Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer1M1Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer2M0Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer2M1Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer3M0Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer3M1Ena
enumerator kINPUTMUX__Dmac1InputTriggerDma1OutAEna
enumerator kINPUTMUX__Dmac1InputTriggerDma1OutBEna
enumerator kINPUTMUX__Dmac1InputTriggerDma1OutCEna
enumerator kINPUTMUX__Dmac1InputTriggerDma1OutDEna
enumerator kINPUTMUX__Dmac1InputTriggerSct0Dmac0Ena
enumerator kINPUTMUX__Dmac1InputTriggerSct0Dmac1Ena
enumerator kINPUTMUX__Dmac1InputTriggerEnetMac0DmaReq0Ena
enumerator kINPUTMUX__Dmac1InputTriggerEnetMac0DmaReq1Ena
enumerator kINPUTMUX__Dmac1InputTriggerUsimDmaRxSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerUsimDmaTxSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerGauGpadc0DmaSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerGauGpadc1DmaSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerGauGpdacaDmaSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerGauGpdacbDmaSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerFlexspiRxEna

```

enumerator kINPUTMUX__Dmac1InputTriggerFlexspiTxEna
enumerator kINPUTMUX__Dmac1InputTriggerLcdRxRegToDmaSingleEna
enumerator kINPUTMUX__Dmac1InputTriggerLcdTxRegToDmaSingleEna
DMA0 REQ signal.

```

```

enumerator kINPUTMUX__Flexcomm0RxToDmac0Ch0RequestEna
enumerator kINPUTMUX__Flexcomm0TxToDmac0Ch1RequestEna
enumerator kINPUTMUX__Flexcomm1RxToDmac0Ch2RequestEna
enumerator kINPUTMUX__Flexcomm1TxToDmac0Ch3RequestEna
enumerator kINPUTMUX__Flexcomm2RxToDmac0Ch4RequestEna
enumerator kINPUTMUX__Flexcomm2TxToDmac0Ch5RequestEna
enumerator kINPUTMUX__Flexcomm3RxToDmac0Ch6RequestEna
enumerator kINPUTMUX__Flexcomm3TxToDmac0Ch7RequestEna
enumerator kINPUTMUX__Dmic0Ch0ToDmac0Ch16RequestEna
enumerator kINPUTMUX__Dmic0Ch1ToDmac0Ch17RequestEna
enumerator kINPUTMUX__Dmic0Ch2ToDmac0Ch18RequestEna
enumerator kINPUTMUX__Dmic0Ch3ToDmac0Ch19RequestEna
enumerator kINPUTMUX__Flexcomm14RxToDmac0Ch26RequestEna
enumerator kINPUTMUX__Flexcomm14TxToDmac0Ch27RequestEna
DMA1 REQ signal.

```

```

enumerator kINPUTMUX__Flexcomm0RxToDmac1Ch0RequestEna
enumerator kINPUTMUX__Flexcomm0TxToDmac1Ch1RequestEna
enumerator kINPUTMUX__Flexcomm1RxToDmac1Ch2RequestEna
enumerator kINPUTMUX__Flexcomm1TxToDmac1Ch3RequestEna
enumerator kINPUTMUX__Flexcomm2RxToDmac1Ch4RequestEna
enumerator kINPUTMUX__Flexcomm2TxToDmac1Ch5RequestEna
enumerator kINPUTMUX__Flexcomm3RxToDmac1Ch6RequestEna
enumerator kINPUTMUX__Flexcomm3TxToDmac1Ch7RequestEna
enumerator kINPUTMUX__Dmic0Ch0ToDmac1Ch16RequestEna
enumerator kINPUTMUX__Dmic0Ch1ToDmac1Ch17RequestEna
enumerator kINPUTMUX__Dmic0Ch2ToDmac1Ch18RequestEna
enumerator kINPUTMUX__Dmic0Ch3ToDmac1Ch19RequestEna
enumerator kINPUTMUX__Flexcomm14RxToDmac1Ch26RequestEna
enumerator kINPUTMUX__Flexcomm14TxToDmac1Ch27RequestEna

```

```

typedef enum _inputmux_connection_t inputmux_connection_t
INPUTMUX connections type.

```

```
typedef enum _inputmux_signal_t inputmux_signal_t
```

INPUTMUX signal enable/disable type.

```
void INPUTMUX__Init(void *base)
```

Initialize INPUTMUX peripheral.

This function enables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

```
void INPUTMUX__AttachSignal(void *base, uint32_t index, inputmux_connection_t connection)
```

Attaches a signal.

This function attaches multiplexed signals from INPUTMUX to target signals. For example, to attach GPIO PORT0 Pin 5 to PINT peripheral, do the following:

```
INPUTMUX__AttachSignal(INPUTMUX, 2, kINPUTMUX_GpioPort0Pin5ToPintsel);
```

In this example, INTMUX has 8 registers for PINT, PINT_SEL0~PINT_SEL7. With parameter index specified as 2, this function configures register PINT_SEL2.

Parameters

- base – Base address of the INPUTMUX peripheral.
- index – The serial number of destination register in the group of INPUTMUX registers with same name.
- connection – Applies signal from source signals collection to target signal.

Return values

None. –

```
void INPUTMUX__EnableSignal(void *base, inputmux_signal_t signal, bool enable)
```

Enable/disable a signal.

This function gates the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.
- signal – Enable signal register id and bit offset.
- enable – Selects enable or disable.

Return values

None. –

```
void INPUTMUX__Deinit(void *base)
```

Deinitialize INPUTMUX peripheral.

This function disables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

```
SCT0_PMUX_ID
```

Periphinmux IDs.

SHSGPIO_PMUX_ID
PINTSEL_PMUX_ID
DMA0_ITRIG_PMUX_ID
DMA0_OTRIG_PMUX_ID
DMA1_ITRIG_PMUX_ID
DMA1_OTRIG_PMUX_ID
CT32BIT0_CAP_PMUX_ID
CT32BIT1_CAP_PMUX_ID
CT32BIT2_CAP_PMUX_ID
CT32BIT3_CAP_PMUX_ID
FREQMEAS_PMUX_ID
DMA0_REQ_ENA0_ID
DMA1_REQ_ENA0_ID
DMA0_ITRIG_EN0_ID
DMA0_ITRIG_EN1_ID
DMA1_ITRIG_EN0_ID
DMA1_ITRIG_EN1_ID
ENA_SHIFT
PMUX_SHIFT

2.31 IO_MUX Driver

enum io_mux_pin_config_t

IO MUX pin configuration. Bit [1:0] for pull configuration Bit [3:2] for drive strength configuration.

Values:

enumerator IO_MUX_PinConfigNoPullDriveWeakest

enumerator IO_MUX_PinConfigNoPullDriveWeak

enumerator IO_MUX_PinConfigNoPullDriveStrong

enumerator IO_MUX_PinConfigNoPullDriveStrongest

enumerator IO_MUX_PinConfigPullUpDriveWeakest

enumerator IO_MUX_PinConfigPullUpDriveWeak

enumerator IO_MUX_PinConfigPullUpDriveStrong

enumerator IO_MUX_PinConfigPullUpDriveStrongest

enumerator IO_MUX_PinConfigPullDownDriveWeakest

enumerator IO_MUX_PinConfigPullDownDriveWeak

enumerator IO_MUX_PinConfigPullDownDriveStrong

enumerator IO_MUX_PinConfigPullDownDriveStrongest

enumerator IO_MUX_PinConfigNoPull

enumerator IO_MUX_PinConfigPullUp

enumerator IO_MUX_PinConfigPullDown

enum io_mux_sleep_pin_level_t

IO MUX sleep pin level.

Values:

enumerator IO_MUX_SleepPinLevelLow

enumerator IO_MUX_SleepPinLevelHigh

enumerator IO_MUX_SleepPinLevelUnchanged

IO_MUX_GPIO_FC_MASK(gpio, fcIdx, fcMsk)

IO_MUX_SGPIO_FLAG(mask)

IO_MUX_GPIO_FLAG(mask)

IO_MUX_FC_OFFSET(mask)

IO_MUX_FC_MASK(mask)

IO_MUX_TIMER_MASK(inMsk, outMsk)

IO_MUX_TIMER_IN_MASK(mask)

IO_MUX_TIMER_OUT_MASK(mask)

IO_MUX_SCTIMER_MASK(inMsk, outMsk)

IO_MUX_FC0_USART_SCK

IO_MUX_FC0_USART_DATA

IO_MUX_FC0_USART_CMD

IO_MUX_FC0_I2C_2_3

IO_MUX_FC0_I2S

IO_MUX_FC0_I2S_DATA

IO_MUX_FC0_SPI_SS0

IO_MUX_FC1_USART_SCK

IO_MUX_FC1_USART_DATA

IO_MUX_FC1_USART_CMD

IO_MUX_FC1_I2C_8_9

IO_MUX_FC1_I2S

IO_MUX_FC1_I2S_DATA
IO_MUX_FC1_SPI_SS0
IO_MUX_FC2_USART_SCK
IO_MUX_FC2_USART_DATA
IO_MUX_FC2_USART_CMD
IO_MUX_FC2_I2C_13_14
IO_MUX_FC2_I2C_16_17
IO_MUX_FC2_I2S
IO_MUX_FC2_I2S_DATA
IO_MUX_FC2_SPI_SS0
IO_MUX_FC3_USART_SCK
IO_MUX_FC3_USART_DATA
IO_MUX_FC3_USART_CMD
IO_MUX_FC3_I2C_24_26
IO_MUX_FC3_I2C_19_20
IO_MUX_FC3_I2S
IO_MUX_FC3_I2S_DATA
IO_MUX_FC3_SPI_SS0
IO_MUX_FC14_USART_SCK
IO_MUX_FC14_USART_DATA
IO_MUX_FC14_USART_CMD
IO_MUX_FC14_I2C_56_57
IO_MUX_FC14_I2S
IO_MUX_FC14_I2S_DATA
IO_MUX_FC14_SPI_SS0
IO_MUX_QUAD_SPI_FLASH
IO_MUX_QUAD_SPI_PSRAM
IO_MUX_PDM
IO_MUX_USB
IO_MUX_SCT_OUT_0
IO_MUX_SCT_OUT_1
IO_MUX_SCT_OUT_8
IO_MUX_SCT_OUT_4

IO_MUX_SCT_OUT_5
IO_MUX_SCT_OUT_6
IO_MUX_SCT_OUT_7
IO_MUX_SCT_OUT_9
IO_MUX_SCT_IN_0
IO_MUX_SCT_IN_1
IO_MUX_SCT_IN_2
IO_MUX_SCT_IN_3
IO_MUX_SCT_IN_4
IO_MUX_SCT_IN_5
IO_MUX_SCT_IN_6
IO_MUX_SCT_IN_7
IO_MUX_CT0_MAT0_OUT
IO_MUX_CT0_MAT1_OUT
IO_MUX_CT0_MAT2_OUT
IO_MUX_CT0_MAT3_OUT
IO_MUX_CT1_MAT0_OUT
IO_MUX_CT1_MAT1_OUT
IO_MUX_CT1_MAT2_OUT
IO_MUX_CT1_MAT3_OUT
IO_MUX_CT2_MAT0_OUT
IO_MUX_CT2_MAT1_OUT
IO_MUX_CT2_MAT2_OUT
IO_MUX_CT2_MAT3_OUT
IO_MUX_CT3_MAT0_OUT
IO_MUX_CT3_MAT1_OUT
IO_MUX_CT3_MAT2_OUT
IO_MUX_CT_INP0
IO_MUX_CT_INP1
IO_MUX_CT_INP2
IO_MUX_CT_INP3
IO_MUX_CT_INP4
IO_MUX_CT_INP5

IO_MUX_CT_INP6
IO_MUX_CT_INP7
IO_MUX_CT_INP8
IO_MUX_CT_INP9
IO_MUX_CT_INP10
IO_MUX_CT_INP11
IO_MUX_CT_INP12
IO_MUX_CT_INP13
IO_MUX_CT_INP14
IO_MUX_MCLK
IO_MUX_UTICK
IO_MUX_USIM
IO_MUX_LCD_8080
IO_MUX_LCD_SPI
IO_MUX_FREQ_GPIO_CLK
IO_MUX_GPIO_INT_BMATCH
IO_MUX_GAU_TRIGGER0
IO_MUX_ACOMP0_GPIO_OUT
IO_MUX_ACOMP0_EDGE_PULSE
IO_MUX_ACOMP1_GPIO_OUT
IO_MUX_ACOMP1_EDGE_PULSE
IO_MUX_GAU_TRIGGER1
IO_MUX_SDIO
IO_MUX_ENET_CLK
IO_MUX_ENET_RX
IO_MUX_ENET_TX
IO_MUX_ENET_MDIO
IO_MUX_ENET_TIMER0
IO_MUX_ENET_TIMER1
IO_MUX_ENET_TIMER2
IO_MUX_ENET_TIMER3
IO_MUX_CLKIN_FRM_PD
IO_MUX_GPIO0

IO_MUX_GPIO1
IO_MUX_GPIO2
IO_MUX_GPIO3
IO_MUX_GPIO4
IO_MUX_GPIO5
IO_MUX_GPIO6
IO_MUX_GPIO7
IO_MUX_GPIO8
IO_MUX_GPIO9
IO_MUX_GPIO10
IO_MUX_GPIO11
IO_MUX_GPIO12
IO_MUX_GPIO13
IO_MUX_GPIO14
IO_MUX_GPIO15
IO_MUX_GPIO16
IO_MUX_GPIO17
IO_MUX_GPIO18
IO_MUX_GPIO19
IO_MUX_GPIO20
IO_MUX_GPIO21
IO_MUX_GPIO22
IO_MUX_GPIO23
IO_MUX_GPIO24
IO_MUX_GPIO25
IO_MUX_GPIO26
IO_MUX_GPIO27
IO_MUX_GPIO28
IO_MUX_GPIO29
IO_MUX_GPIO30
IO_MUX_GPIO31
IO_MUX_GPIO32
IO_MUX_GPIO33

IO_MUX_GPIO34
IO_MUX_GPIO35
IO_MUX_GPIO36
IO_MUX_GPIO37
IO_MUX_GPIO38
IO_MUX_GPIO39
IO_MUX_GPIO40
IO_MUX_GPIO41
IO_MUX_GPIO42
IO_MUX_GPIO43
IO_MUX_GPIO44
IO_MUX_GPIO45
IO_MUX_GPIO46
IO_MUX_GPIO47
IO_MUX_GPIO48
IO_MUX_GPIO49
IO_MUX_GPIO50
IO_MUX_GPIO51
IO_MUX_GPIO52
IO_MUX_GPIO53
IO_MUX_GPIO54
IO_MUX_GPIO55
IO_MUX_GPIO56
IO_MUX_GPIO57
IO_MUX_GPIO58
IO_MUX_GPIO59
IO_MUX_GPIO60
IO_MUX_GPIO61
IO_MUX_GPIO62
IO_MUX_GPIO63
IO_MUX_SGPIO0
IO_MUX_SGPIO1
IO_MUX_SGPIO2

IO_MUX_SGPIO3
IO_MUX_SGPIO4
IO_MUX_SGPIO5
IO_MUX_SGPIO6
IO_MUX_SGPIO7
IO_MUX_SGPIO8
IO_MUX_SGPIO9
IO_MUX_SGPIO10
IO_MUX_SGPIO11
IO_MUX_SGPIO12
IO_MUX_SGPIO13
IO_MUX_SGPIO14
IO_MUX_SGPIO15
IO_MUX_SGPIO16
IO_MUX_SGPIO17
IO_MUX_SGPIO18
IO_MUX_SGPIO19
IO_MUX_SGPIO20
IO_MUX_SGPIO21
IO_MUX_SGPIO22
IO_MUX_SGPIO23
IO_MUX_SGPIO24
IO_MUX_SGPIO25
IO_MUX_SGPIO26
IO_MUX_SGPIO27
IO_MUX_SGPIO28
IO_MUX_SGPIO29
IO_MUX_SGPIO30
IO_MUX_SGPIO31
IO_MUX_AON_CAPTURE

```
static inline void IO_MUX_SetPinMux(uint32_t pinLowMask, uint32_t pinHighMask, uint32_t
                                   gpioFcSetMask, uint32_t gpioFcClrMask, uint32_t
                                   fselSetMask, uint32_t fselClrMask, uint32_t
                                   ctimerSetMask, uint32_t ctimerClrMask, uint32_t
                                   sctimerSetMask, uint32_t sctimerClrMask)
```

Sets the IO_MUX pin mux mode.

This is an example to set the GPIO2/GPIO3 as the Flexcomm0 UART RX/TX:

```
IO_MUX_SetPinMux(IO_MUX_FC0_USART_DATA);
```

This is an example to set the GPIO6/GPIO10 as Flexcomm1 I2C SDA/SCL:

```
IO_MUX_SetPinMux(IO_MUX_FC1_I2C_6_10);
```

Note: The parameters can be filled with the pin function ID macros.

Parameters

- pinLowMask – The GPIO0-31 pins mask.
- pinHighMask – The GPIO32-63 pins mask.
- gpioFcSetMask – The GPIO and Flexcomm registers mask to set, defined by IO_MUX_GPIO_FC_MASK()
- gpioFcClrMask – The GPIO and Flexcomm registers mask to clear, defined by IO_MUX_GPIO_FC_MASK()
- fselSetMask – The FSEL register mask to set
- fselClrMask – The FSEL register mask to clear
- ctimerSetMask – The C_TIMER_IN/C_TIMER_OUT register mask to set, defined by IO_MUX_CTIMER_MASK()
- ctimerClrMask – The C_TIMER_IN/C_TIMER_OUT register mask to clear, defined by IO_MUX_CTIMER_MASK()
- sctimerSetMask – The SC_TIMER register mask to set
- sctimerClrMask – The SC_TIMER register mask to clear

```
static inline void IO_MUX_SetPinConfig(uint32_t pin, io_mux_pin_config_t config)
```

Sets the IO_MUX pin mux pull up/down configuration.

This is an example to set the GPIO2 pin to pull down:

```
IO_MUX_SetPinConfig(2U, IO_MUX_PinConfigPullDown);
```

Parameters

- pin – The GPIO pin index to config.
- config – The pull up/down setting for the pin.

```
static inline void IO_MUX_SetPinOutLevelInSleep(uint32_t pin, io_mux_sleep_pin_level_t level)
```

Sets IO output level in sleep mode. If level set to IO_MUX_SleepPinLevelUnchanged, the IO configuration is same as the active mode.

This is an example to set the GPIO2 pin to output high during sleep:

```
IO_MUX_SetPinOutLevelInSleep(2U, IO_MUX_SleepPinLevelHigh);
```

Parameters

- pin – The GPIO pin index to config.
- level – Output level in sleep.

```
static inline void IO_MUX_SetRfPinOutLevelInSleep(uint32_t pin, io_mux_sleep_pin_level_t level)
```

Sets RF Switch Pin 0-3 output level in sleep mode. If level set to `IO_MUX_SleepPinLevelUnchanged`, the IO configuration is same as the active mode.

This is an example to set the RF_CNTL0 pin to output low during sleep:

```
IO_MUX_SetRfPinOutLevelInSleep(0U, IO_MUX_SleepPinLevelLow);
```

Parameters

- pin – The RF Switch pin index to config.
- level – Output level in sleep.

FSL_IO_MUX_DRIVER_VERSION

IO_MUX driver version 2.2.2.

FSL_COMPONENT_ID

2.32 IPED Driver

enum `_iped_status`

Values:

enumerator `kStatus_IPED_RegionIsLocked`

typedef `uint32_t iped_region_t`

typedef `uint32_t iped_prince_rounds_t`

```
static inline void IPED_EncryptEnable(FLEXSPI_Type *base)
```

Enable data encryption.

This function enables IPED on-the-fly data encryption.

Parameters

- base – IPED peripheral address.

```
static inline void IPED_EncryptDisable(FLEXSPI_Type *base)
```

Disable data encryption.

This function disables IPED on-the-fly data encryption.

Parameters

- base – IPED peripheral address.

```
static inline void IPED_SetLock(FLEXSPI_Type *base, iped_region_t region)
```

Locks access for specified region registers or data mask register.

This function sets lock on specified region.

Parameters

- base – IPED peripheral address.
- region – number to lock

static inline bool IPED_IsRegionLocked(FLEXSPI_Type *base, *iped_region_t* region)

Gets info whether IPED region is locked.

Parameters

- base – IPED peripheral address.
- region – Selection of the IPED region to be queried.

status_t IPED_SetRegionEnable(FLEXSPI_Type *base, *iped_region_t* region, bool enable)

Enable encryption for a specific IPED region encryption.

Parameters

- base – IPED peripheral address.
- region – Selection of the IPED region to be enabled.

static inline bool IPED_IsRegionEnabled(FLEXSPI_Type *base, *iped_region_t* region)

Gets info whether IPED region encryption is enabled.

Parameters

- base – IPED peripheral address.
- region – Selection of the IPED region to be queried.

status_t IPED_SetRegionAddressRange(FLEXSPI_Type *base, *iped_region_t* region, uint32_t start_address, uint32_t end_address)

Sets IPED region address range.

This function configures IPED region address range.

Parameters

- base – IPED peripheral address.
- region – Selection of the IPED region to be configured.
- start_address – Start address for region.
- end_address – End address for region.

void IPED_GetRegionAddressRange(FLEXSPI_Type *base, *iped_region_t* region, uint32_t *start_address, uint32_t *end_address)

Gets IPED region base address.

This function reads current start and end address settings for selected region.

Parameters

- base – IPED peripheral address.
- region – Selection of the IPED region to be configured.
- start_address – Start address for region.
- end_address – End address for region.

void IPED_SetRegionIV(FLEXSPI_Type *base, *iped_region_t* region, const uint8_t iv[8])

Sets the IPED region IV.

This function sets specified AES IV for the given region.

Parameters

- base – IPED peripheral address.
- region – Selection of the IPED region to be configured.

- *iv* – 64-bit AES IV in little-endian byte order.

void IPED_GetRegionIV(FLEXSPI_Type *base, *iped_region_t* region, uint8_t iv[8])

Gets the IPED region IV.

This function gets specified AES IV for the given region.

Parameters

- *base* – IPED peripheral address.
- *region* – Selection of the IPED region to be configured.
- *iv* – 64-bit AES IV in little-endian byte order.

void IPED_SetRegionAAD(FLEXSPI_Type *base, *iped_region_t* region, const uint8_t aad[8])

Sets the IPED region AAD.

This function sets specified AES AAD for the given region.

Parameters

- *base* – IPED peripheral address.
- *region* – Selection of the IPED region to be configured.
- *iv* – 64-bit AES AAD in little-endian byte order.

void IPED_GetRegionAAD(FLEXSPI_Type *base, *iped_region_t* region, uint8_t aad[8])

Gets the IPED region AAD.

This function gets specified AES AAD for the given region.

Parameters

- *base* – IPED peripheral address.
- *region* – Selection of the IPED region to be configured.
- *iv* – 64-bit AES AAD in little-endian byte order.

static inline void IPED_SetPrinceRounds(FLEXSPI_Type *base, *iped_prince_rounds_t* rounds)

Sets the number of rounds used for PRINCE.

Parameters

- *base* – IPED peripheral address.
- *rounds* – Number of PRINCE rounds used during encryption/decryption

static inline *iped_prince_rounds_t* IPED_GetPrinceRounds(FLEXSPI_Type *base)

Gets the number of rounds used for PRINCE.

Parameters

- *base* – IPED peripheral address.
- *rounds* – Number of PRINCE rounds used during encryption/decryption

FSL_IPED_DRIVER_VERSION

IPED driver version for RW61x. Version 1.0.0.

- Version 1.0.1
 - Initial version

kIPED_Region0

IPED region 0

kIPED_Region1
 IPED region 1

kIPED_Region2
 IPED region 2

kIPED_Region3
 IPED region 3

kIPED_Region4
 IPED region 4

kIPED_Region5
 IPED region 5

kIPED_Region6
 IPED region 6

kIPED_Region7
 IPED region 7

kIPED_Region8
 IPED region 8

kIPED_Region9
 IPED region 9

kIPED_Region10
 IPED region 10

kIPED_Region11
 IPED region 11

kIPED_Region12
 IPED region 12

kIPED_Region13
 IPED region 13

kIPED_Region14
 IPED region 14

kIPED_Region15
 IPED region 15

kIPED_PrinceRounds12

kIPED_PrinceRounds22

IPED_REGION_COUNT
 IPED region count.

IPED_RW_ENABLE_VAL

IPED_RW_DISABLE_VAL

IPED_CTX_REG_OFFSET

2.33 Intrusion and Tamper Response Controller

2.34 ITRC

status_t ITRC_SetActionToEvent(ITRC_Type *base, *itrc_out_signals_t* out, *itrc_in_signals_t* in, *itrc_lock_t* lock, *itrc_enable_t* enable)

Set ITRC Action to Event.

This function sets input Event signal to corresponding output Action response signal.

Parameters

- base – ITRC peripheral base address
- out – ITRC OUT signal action
- in – ITRC IN signal event
- lock – if set locks INx_SEL configuration. This can be cleared only by PMC Core reset.
- enable – if set input Event will be selected for output Action, otherwise disable (if not already locked).

Returns

kStatus_Success if success, kStatus_InvalidArgument otherwise

void ITRC_SetSWEvent0(ITRC_Type *base)

Trigger ITRC SW Event 0.

This function set SW_EVENT0 register with value !=0 which triggers ITRC SW Event 0.

Parameters

- base – ITRC peripheral base address

void ITRC_SetSWEvent1(ITRC_Type *base)

Trigger ITRC SW Event 1.

This function set SW_EVENT1 register with value !=0 which triggers ITRC SW Event 1.

Parameters

- base – ITRC peripheral base address

bool ITRC_GetInEventStatus(ITRC_Type *base, *itrc_in_signals_t* event)

Get ITRC input event status.

This function returns ITRC status corresponding to provided input event.

Parameters

- base – ITRC peripheral base address
- event – represents input event to get from STATUS register (see ITRC_STATUS_INx)

Returns

boolean TRUE if corresponding event occurred FALSE otherwise

bool ITRC_GetOutActionStatus(ITRC_Type *base, *itrc_out_signals_t* action)

Get ITRC output action status.

This function returns ITRC register output status.

Parameters

- base – ITRC peripheral base address

- *action* – represents output action to get from STATUS register (see ITRC_STATUS_OUTx)

Returns

boolean TRUE if corresponding action occurred FALSE otherwise

status_t ITRC_ClearInEventStatus(ITRC_Type *base, *itrc_in_signals_t* event)

Clear In ITRC status.

This function clears corresponding ITRC event or action in input STATUS register.

Parameters

- *base* – ITRC peripheral base address
- *event* – represents input event in STATUS register to be cleared (see ITRC_STATUS_INx)

Returns

kStatus_Success if success, kStatus_InvalidArgument otherwise

status_t ITRC_ClearOutActionStatus(ITRC_Type *base, *itrc_out_signals_t* action)

Clear Out ITRC status.

This function clears corresponding ITRC event or action in output STATUS register.

Parameters

- *base* – ITRC peripheral base address
- *action* – represents output action in STATUS register to be cleared (see OUTx_STATUS)

Returns

kStatus_Success if success, kStatus_InvalidArgument otherwise

status_t ITRC_ClearAllStatus(ITRC_Type *base)

Clear All ITRC status.

This function clears all event and action status.

Parameters

- *base* – ITRC peripheral base address

Returns

kStatus_Success if success

status_t ITRC_Init(ITRC_Type *base)

Initialize ITRC.

This function initializes ITRC by enabling IRQ.

Parameters

- *base* – ITRC peripheral base address
- *conf* – ITRC configuration structure

Returns

Status of the init operation

void ITRC_Deinit(ITRC_Type *base)

Deinitialize ITRC.

This function deinitializes ITRC by disabling IRQ.

Parameters

- *base* – ITRC peripheral base address

FSL_ITRC_DRIVER_VERSION

Defines ITRC driver version 2.0.0.

Change log:

- Version 2.0.0
 - Initial version.

typedef uint32_t itrc_in_signals_t

ITRC input events.

typedef uint32_t itrc_out_signals_t

ITRC output actions.

typedef uint32_t itrc_lock_t

ITRC lock/unlock.

typedef uint32_t itrc_enable_t

ITRC enable/disable.

void ITRC_DriverIRQHandler(void)

ITRC_INPUT_SIGNALS_NUM

kITRC_CauTempeprature

kITRC_PmipTemperature

kITRC_VddCore

kITRC_Vdd18

kITRC_Vdd33

kITRC_VddCoreGlitch

kITRC_AnalogSensor

kITRC_Ahb

kITRC_Cwd

kITRC_Css

kITRC_Pkc

kITRC_Otp

kITRC_Prince

kITRC_ClockGlitch

kITRC_SecurityIP

kITRC_Trng

kITRC_PmipGlitch

kITRC_PmipVddCoreGlitch

kITRC_TcpuPll

kITRC_T3Pll

kITRC_SwEvent0

kITRC_SwEvent1

kITRC_Irq

kITRC_ChipReset

kITRC_Unlock

kITRC_Lock

kITRC_Enable

kITRC_Disable

IN_STATUS0_EVENTS_MASK

IN_STATUS1_EVENTS_MASK

OUT_ACTIONS_MASK

ITRC

2.35 Common Driver

FSL_COMMON_DRIVER_VERSION

common driver version.

DEBUG_CONSOLE_DEVICE_TYPE_NONE

No debug console.

DEBUG_CONSOLE_DEVICE_TYPE_UART

Debug console based on UART.

DEBUG_CONSOLE_DEVICE_TYPE_LPUART

Debug console based on LPUART.

DEBUG_CONSOLE_DEVICE_TYPE_LPSCI

Debug console based on LPSCI.

DEBUG_CONSOLE_DEVICE_TYPE_USBCDC

Debug console based on USBCDC.

DEBUG_CONSOLE_DEVICE_TYPE_FLEXCOMM

Debug console based on FLEXCOMM.

DEBUG_CONSOLE_DEVICE_TYPE_IUART

Debug console based on i.MX UART.

DEBUG_CONSOLE_DEVICE_TYPE_VUSART

Debug console based on LPC_VUSART.

DEBUG_CONSOLE_DEVICE_TYPE_MINI_USART

Debug console based on LPC_USART.

DEBUG_CONSOLE_DEVICE_TYPE_SWO

Debug console based on SWO.

DEBUG_CONSOLE_DEVICE_TYPE_QSCI

Debug console based on QSCI.

MIN(*a*, *b*)

Computes the minimum of *a* and *b*.

MAX(*a*, *b*)

Computes the maximum of *a* and *b*.

UINT16_MAX

Max value of uint16_t type.

UINT32_MAX

Max value of uint32_t type.

SDK_ATOMIC_LOCAL_ADD(*addr*, *val*)

Add value *val* from the variable at address *address*.

SDK_ATOMIC_LOCAL_SUB(*addr*, *val*)

Subtract value *val* to the variable at address *address*.

SDK_ATOMIC_LOCAL_SET(*addr*, *bits*)

Set the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR(*addr*, *bits*)

Clear the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_TOGGLE(*addr*, *bits*)

Toggle the bits specified by *bits* to the variable at address *address*.

SDK_ATOMIC_LOCAL_CLEAR_AND_SET(*addr*, *clearBits*, *setBits*)

For the variable at address *address*, clear the bits specified by *clearBits* and set the bits specified by *setBits*.

SDK_ATOMIC_LOCAL_COMPARE_AND_SET(*addr*, *expected*, *newValue*)

For the variable at address *address*, check whether the value equal to *expected*. If value same as *expected* then update *newValue* to address and return **true**, else return **false**.

SDK_ATOMIC_LOCAL_TEST_AND_SET(*addr*, *newValue*)

For the variable at address *address*, set as *newValue* value and return old value.

USEC_TO_COUNT(*us*, *clockFreqInHz*)

Macro to convert a microsecond period to raw count value

COUNT_TO_USEC(*count*, *clockFreqInHz*)

Macro to convert a raw count value to microsecond

MSEC_TO_COUNT(*ms*, *clockFreqInHz*)

Macro to convert a millisecond period to raw count value

COUNT_TO_MSEC(*count*, *clockFreqInHz*)

Macro to convert a raw count value to millisecond

SDK_ISR_EXIT_BARRIER

SDK_SIZEALIGN(*var*, *alignbytes*)

Macro to define a variable with L1 d-cache line size alignment

Macro to define a variable with L2 cache line size alignment

Macro to change a value to a given size aligned value

AT_NONCACHEABLE_SECTION(*var*)

Define a variable *var*, and place it in non-cacheable section.

`AT_NONCACHEABLE_SECTION_ALIGN(var, alignbytes)`

Define a variable *var*, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

`AT_NONCACHEABLE_SECTION_INIT(var)`

Define a variable *var* with initial value, and place it in non-cacheable section.

`AT_NONCACHEABLE_SECTION_ALIGN_INIT(var, alignbytes)`

Define a variable *var* with initial value, and place it in non-cacheable section, the start address of the variable is aligned to *alignbytes*.

`enum __status_groups`

Status group numbers.

Values:

enumerator `kStatusGroup_Generic`

Group number for generic status codes.

enumerator `kStatusGroup_FLASH`

Group number for FLASH status codes.

enumerator `kStatusGroup_LPSPI`

Group number for LPSPI status codes.

enumerator `kStatusGroup_FLEXIO_SPI`

Group number for FLEXIO SPI status codes.

enumerator `kStatusGroup_DSPI`

Group number for DSPI status codes.

enumerator `kStatusGroup_FLEXIO_UART`

Group number for FLEXIO UART status codes.

enumerator `kStatusGroup_FLEXIO_I2C`

Group number for FLEXIO I2C status codes.

enumerator `kStatusGroup_LPI2C`

Group number for LPI2C status codes.

enumerator `kStatusGroup_UART`

Group number for UART status codes.

enumerator `kStatusGroup_I2C`

Group number for I2C status codes.

enumerator `kStatusGroup_LPSCI`

Group number for LPSCI status codes.

enumerator `kStatusGroup_LPUART`

Group number for LPUART status codes.

enumerator `kStatusGroup_SPI`

Group number for SPI status code.

enumerator `kStatusGroup_XRDC`

Group number for XRDC status code.

enumerator `kStatusGroup_SEMA42`

Group number for SEMA42 status code.

enumerator `kStatusGroup_SDHC`

Group number for SDHC status code

enumerator kStatusGroup_SDMMC
Group number for SDMMC status code

enumerator kStatusGroup_SAI
Group number for SAI status code

enumerator kStatusGroup_MCG
Group number for MCG status codes.

enumerator kStatusGroup_SCG
Group number for SCG status codes.

enumerator kStatusGroup_SDSPI
Group number for SDSPI status codes.

enumerator kStatusGroup_FLEXIO_I2S
Group number for FLEXIO I2S status codes

enumerator kStatusGroup_FLEXIO_MCULCD
Group number for FLEXIO LCD status codes

enumerator kStatusGroup_FLASHIAP
Group number for FLASHIAP status codes

enumerator kStatusGroup_FLEXCOMM_I2C
Group number for FLEXCOMM I2C status codes

enumerator kStatusGroup_I2S
Group number for I2S status codes

enumerator kStatusGroup_IUART
Group number for IUART status codes

enumerator kStatusGroup_CSI
Group number for CSI status codes

enumerator kStatusGroup_MIPI_DSI
Group number for MIPI DSI status codes

enumerator kStatusGroup_SDRAMC
Group number for SDRAMC status codes.

enumerator kStatusGroup_POWER
Group number for POWER status codes.

enumerator kStatusGroup_ENET
Group number for ENET status codes.

enumerator kStatusGroup_PHY
Group number for PHY status codes.

enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.

enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.

enumerator kStatusGroup_LMEM
Group number for LMEM status codes.

enumerator kStatusGroup_QSPI
Group number for QSPI status codes.

enumerator kStatusGroup_DMA
Group number for DMA status codes.

enumerator kStatusGroup_EDMA
Group number for EDMA status codes.

enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.

enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.

enumerator kStatusGroup_LTC
Group number for LTC status codes.

enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.

enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.

enumerator kStatusGroup_DMIC
Group number for DMIC status codes.

enumerator kStatusGroup_SDIF
Group number for SDIF status codes.

enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.

enumerator kStatusGroup_OTP
Group number for OTP status codes.

enumerator kStatusGroup_MCAN
Group number for MCAN status codes.

enumerator kStatusGroup_CAAM
Group number for CAAM status codes.

enumerator kStatusGroup_ECSPI
Group number for ECSPI status codes.

enumerator kStatusGroup_USDHC
Group number for USDHC status codes.

enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

enumerator kStatusGroup_DCP
Group number for DCP status codes.

enumerator kStatusGroup_MSCAN
Group number for MSCAN status codes.

enumerator kStatusGroup_ESAI
Group number for ESAI status codes.

enumerator kStatusGroup_FLEXSPI
Group number for FLEXSPI status codes.

enumerator `kStatusGroup_MMDC`
Group number for MMDC status codes.

enumerator `kStatusGroup_PDM`
Group number for MIC status codes.

enumerator `kStatusGroup_SDMA`
Group number for SDMA status codes.

enumerator `kStatusGroup_ICS`
Group number for ICS status codes.

enumerator `kStatusGroup_SPDIF`
Group number for SPDIF status codes.

enumerator `kStatusGroup_LPC_MINISPI`
Group number for LPC_MINISPI status codes.

enumerator `kStatusGroup_HASHCRYPT`
Group number for Hashcrypt status codes

enumerator `kStatusGroup_LPC_SPI_SSP`
Group number for LPC_SPI_SSP status codes.

enumerator `kStatusGroup_I3C`
Group number for I3C status codes

enumerator `kStatusGroup_LPC_I2C_1`
Group number for LPC_I2C_1 status codes.

enumerator `kStatusGroup_NOTIFIER`
Group number for NOTIFIER status codes.

enumerator `kStatusGroup_DebugConsole`
Group number for debug console status codes.

enumerator `kStatusGroup_SEMC`
Group number for SEMC status codes.

enumerator `kStatusGroup_ApplicationRangeStart`
Starting number for application groups.

enumerator `kStatusGroup_IAP`
Group number for IAP status codes

enumerator `kStatusGroup_SFA`
Group number for SFA status codes

enumerator `kStatusGroup_SPC`
Group number for SPC status codes.

enumerator `kStatusGroup_PUF`
Group number for PUF status codes.

enumerator `kStatusGroup_TOUCH_PANEL`
Group number for touch panel status codes

enumerator `kStatusGroup_VBAT`
Group number for VBAT status codes

enumerator `kStatusGroup_XSPI`
Group number for XSPI status codes

enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes

enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes

enumerator kStatusGroup_HAL_GPIO
Group number for HAL GPIO status codes.

enumerator kStatusGroup_HAL_UART
Group number for HAL UART status codes.

enumerator kStatusGroup_HAL_TIMER
Group number for HAL TIMER status codes.

enumerator kStatusGroup_HAL_SPI
Group number for HAL SPI status codes.

enumerator kStatusGroup_HAL_I2C
Group number for HAL I2C status codes.

enumerator kStatusGroup_HAL_FLASH
Group number for HAL FLASH status codes.

enumerator kStatusGroup_HAL_PWM
Group number for HAL PWM status codes.

enumerator kStatusGroup_HAL_RNG
Group number for HAL RNG status codes.

enumerator kStatusGroup_HAL_I2S
Group number for HAL I2S status codes.

enumerator kStatusGroup_HAL_ADC_SENSOR
Group number for HAL ADC SENSOR status codes.

enumerator kStatusGroup_TIMERMANAGER
Group number for TiMER MANAGER status codes.

enumerator kStatusGroup_SERIALIZEDMANAGER
Group number for SERIAL MANAGER status codes.

enumerator kStatusGroup_LED
Group number for LED status codes.

enumerator kStatusGroup_BUTTON
Group number for BUTTON status codes.

enumerator kStatusGroup_EXTERN_EEPROM
Group number for EXTERN EEPROM status codes.

enumerator kStatusGroup_SHELL
Group number for SHELL status codes.

enumerator kStatusGroup_MEM_MANAGER
Group number for MEM MANAGER status codes.

enumerator kStatusGroup_LIST
Group number for List status codes.

enumerator kStatusGroup_OSA
Group number for OSA status codes.

enumerator `kStatusGroup_COMMON_TASK`
Group number for Common task status codes.

enumerator `kStatusGroup_MSG`
Group number for messaging status codes.

enumerator `kStatusGroup_SDK_OCOTP`
Group number for OCOTP status codes.

enumerator `kStatusGroup_SDK_FLEXSPINOR`
Group number for FLEXSPINOR status codes.

enumerator `kStatusGroup_CODEC`
Group number for codec status codes.

enumerator `kStatusGroup_ASRC`
Group number for codec status ASRC.

enumerator `kStatusGroup_OTFAD`
Group number for codec status codes.

enumerator `kStatusGroup_SDIOSLV`
Group number for SDIOSLV status codes.

enumerator `kStatusGroup_MECC`
Group number for MECC status codes.

enumerator `kStatusGroup_ENET_QOS`
Group number for ENET_QOS status codes.

enumerator `kStatusGroup_LOG`
Group number for LOG status codes.

enumerator `kStatusGroup_I3CBUS`
Group number for I3CBUS status codes.

enumerator `kStatusGroup_QSCI`
Group number for QSCI status codes.

enumerator `kStatusGroup_ELEMU`
Group number for ELEMU status codes.

enumerator `kStatusGroup_QUEUEDSPI`
Group number for QSPI status codes.

enumerator `kStatusGroup_POWER_MANAGER`
Group number for POWER_MANAGER status codes.

enumerator `kStatusGroup_IPED`
Group number for IPED status codes.

enumerator `kStatusGroup_ELS_PKC`
Group number for ELS PKC status codes.

enumerator `kStatusGroup_CSS_PKC`
Group number for CSS PKC status codes.

enumerator `kStatusGroup_HOSTIF`
Group number for HOSTIF status codes.

enumerator `kStatusGroup_CLIF`
Group number for CLIF status codes.

enumerator kStatusGroup_BMA
Group number for BMA status codes.

enumerator kStatusGroup_NETC
Group number for NETC status codes.

enumerator kStatusGroup_ELE
Group number for ELE status codes.

enumerator kStatusGroup_GLIKEY
Group number for GLIKEY status codes.

enumerator kStatusGroup_AON_POWER
Group number for AON_POWER status codes.

enumerator kStatusGroup_AON_COMMON
Group number for AON_COMMON status codes.

enumerator kStatusGroup_ENDAT3
Group number for ENDAT3 status codes.

enumerator kStatusGroup_HIPERFACE
Group number for HIPERFACE status codes.

enumerator kStatusGroup_NPX
Group number for NPX status codes.

enumerator kStatusGroup_ELA_CSEC
Group number for ELA_CSEC status codes.

enumerator kStatusGroup_FLEXIO_T_FORMAT
Group number for T-format status codes.

enumerator kStatusGroup_FLEXIO_A_FORMAT
Group number for A-format status codes.

Generic status return codes.

Values:

enumerator kStatus_Success
Generic status for Success.

enumerator kStatus_Fail
Generic status for Fail.

enumerator kStatus_ReadOnly
Generic status for read only failure.

enumerator kStatus_OutOfRange
Generic status for out of range access.

enumerator kStatus_InvalidArgument
Generic status for invalid argument check.

enumerator kStatus_Timeout
Generic status for timeout.

enumerator kStatus_NoTransferInProgress
Generic status for no transfer in progress.

enumerator `kStatus_Busy`

Generic status for module is busy.

enumerator `kStatus_NoData`

Generic status for no data is found for the operation.

typedef `int32_t status_t`

Type used for all status and error return values.

void `*SDK_Malloc(size_t size, size_t alignbytes)`

Allocate memory with given alignment and aligned size.

This is provided to support the dynamically allocated memory used in cache-able region.

Parameters

- `size` – The length required to malloc.
- `alignbytes` – The alignment size.

Return values

The – allocated memory.

void `SDK_Free(void *ptr)`

Free memory.

Parameters

- `ptr` – The memory to be release.

void `SDK_DelayAtLeastUs(uint32_t delayTime_us, uint32_t coreClock_Hz)`

Delay at least for some time. Please note that, this API uses while loop for delay, different run-time environments make the time not precise, if precise delay count was needed, please implement a new delay function with hardware timer.

Parameters

- `delayTime_us` – Delay time in unit of microsecond.
- `coreClock_Hz` – Core clock frequency with Hz.

static inline `status_t EnableIRQ(IRQn_Type interrupt)`

Enable specific interrupt.

Enable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only enables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ number.

Return values

- `kStatus_Success` – Interrupt enabled successfully
- `kStatus_Fail` – Failed to enable the interrupt

static inline `status_t DisableIRQ(IRQn_Type interrupt)`

Disable specific interrupt.

Disable LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only disables the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ number.

Return values

- `kStatus_Success` – Interrupt disabled successfully
- `kStatus_Fail` – Failed to disable the interrupt

`static inline status_t EnableIRQWithPriority(IRQn_Type interrupt, uint8_t priNum)`

Enable the IRQ, and also set the interrupt priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ to Enable.
- `priNum` – Priority number set to interrupt controller register.

Return values

- `kStatus_Success` – Interrupt priority set successfully
- `kStatus_Fail` – Failed to set the interrupt priority.

`static inline status_t IRQ_SetPriority(IRQn_Type interrupt, uint8_t priNum)`

Set the IRQ priority.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- `interrupt` – The IRQ to set.
- `priNum` – Priority number set to interrupt controller register.

Return values

- `kStatus_Success` – Interrupt priority set successfully
- `kStatus_Fail` – Failed to set the interrupt priority.

`static inline status_t IRQ_ClearPendingIRQ(IRQn_Type interrupt)`

Clear the pending IRQ flag.

Only handle LEVEL1 interrupt. For some devices, there might be multiple interrupt levels. For example, there are NVIC and intmux. Here the interrupts connected to NVIC are the LEVEL1 interrupts, because they are routed to the core directly. The interrupts connected to intmux are the LEVEL2 interrupts, they are routed to NVIC first then routed to core.

This function only handles the LEVEL1 interrupts. The number of LEVEL1 interrupts is indicated by the feature macro `FSL_FEATURE_NUMBER_OF_LEVEL1_INT_VECTORS`.

Parameters

- interrupt – The flag which IRQ to clear.

Return values

- kStatus_Success – Interrupt priority set successfully
- kStatus_Fail – Failed to set the interrupt priority.

static inline uint32_t DisableGlobalIRQ(void)

Disable the global IRQ.

Disable the global interrupt and return the current primask register. User is required to provided the primask register for the EnableGlobalIRQ().

Returns

Current primask value.

static inline void EnableGlobalIRQ(uint32_t primask)

Enable the global IRQ.

Set the primask register with the provided primask value but not just enable the primask. The idea is for the convenience of integration of RTOS. some RTOS get its own management mechanism of primask. User is required to use the EnableGlobalIRQ() and DisableGlobalIRQ() in pair.

Parameters

- primask – value of primask register to be restored. The primask value is supposed to be provided by the DisableGlobalIRQ().

static inline bool __SDK_AtomicLocalCompareAndSet(uint32_t *addr, uint32_t expected, uint32_t newValue)

static inline uint32_t __SDK_AtomicTestAndSet(uint32_t *addr, uint32_t newValue)

FSL_DRIVER_TRANSFER_DOUBLE_WEAK_IRQ

Macro to use the default weak IRQ handler in drivers.

MAKE_STATUS(group, code)

Construct a status code value from a group and code number.

MAKE_VERSION(major, minor, bugfix)

Construct the version number for drivers.

The driver version is a 32-bit number, for both 32-bit platforms(such as Cortex M) and 16-bit platforms(such as DSC).

Unused	Major Version	Minor Version	Bug Fix
31 25 24	17 16	9 8	0

ARRAY_SIZE(x)

Computes the number of elements in an array.

UINT64_H(X)

Macro to get upper 32 bits of a 64-bit value

UINT64_L(X)

Macro to get lower 32 bits of a 64-bit value

SUPPRESS_FALL_THROUGH_WARNING()

For switch case code block, if case section ends without “break;” statement, there will be fallthrough warning with compiler flag -Wextra or -Wimplicit-fallthrough=n when using armgcc. To suppress this warning, “SUPPRESS_FALL_THROUGH_WARNING;” need to be added at the end of each case section which misses “break;”statement.

MSDK_REG_SECURE_ADDR(x)

Convert the register address to the one used in secure mode.

MSDK_REG_NONSECURE_ADDR(x)

Convert the register address to the one used in non-secure mode.

MSDK_INVALID_IRQ_HANDLER

Invalid IRQ handler address.

2.36 LCDIC Driver

status_t LCDIC_Init(LCDIC_Type *base, const *lcdic_config_t* *config)

Initialize the LCDIC.

This function initializes the LCDIC to work, it configures the LCDIC according to the configure structure and enables the module. After calling this function, the peripheral is ready to work.

Parameters

- base – LCDIC peripheral base address.

Return values

kStatus_Success – Initialize successfully.

void LCDIC_Deinit(LCDIC_Type *base)

De-initialize the LCDIC.

This function disables the LCDIC, and disables peripheral clock if necessary.

Parameters

- base – LCDIC peripheral base address.

void LCDIC_GetDefaultConfig(*lcdic_config_t* *config)

Get the default configuration for to initialize the LCDIC.

The default configuration value is:

```
config->mode      = kLCDIC_3WireSPI;
config->endian     = kLCDIC_BigEndian;
config->rxThreshold = kLCDIC_RxThreshold0Word;
config->txThreshold = kLCDIC_TxThreshold3Word;

config->timerRatio0 = 8;
config->timerRatio1 = 9;

config->resetPulseWidth_Timer0 = 20;
config->resetSequence          = 0;
config->resetSequencePulseNum  = 1;
config->resetPolarity          = kLCDIC_ResetActiveLow;

config->i8080CtrlFlags = kLCDIC_I8080_CsActiveLow | kLCDIC_I8080_DcCmdLow | kLCDIC_
↪I8080_RdActiveLow |
                      kLCDIC_I8080_WrActiveLow | kLCDIC_I8080_CsEnableIdleOff;

config->csWaitTime      = 2;
config->csSetupTime     = 2;
config->csHoldTime      = 2;
config->dcSetupTime     = 2;
config->dcHoldTime      = 2;
config->writeDataSetupTime = 2;
```

(continues on next page)

(continued from previous page)

```
config->writeDataHoldTime      = 2;
config->writeEnableActiveWidth = 6;
config->writeEnableInactiveWidth = 6;
config->readEnableActiveWidth  = 15;
config->readEnableInactiveWidth = 15;

config->spiCtrlFlags =
    kLCDIC_SPI_MsbFirst | kLCDIC_SPI_ClkActiveHigh | kLCDIC_SPI_ClkPhaseFirstEdge |
    ↪kLCDIC_SPI_DcCmdLow;

config->teTimeoutTime_Timer1 = 16;
config->teSyncWaitTime_Timer1 = 0;

config->cmdShortTimeout_Timer0 = 1;
config->cmdLongTimeout_Timer1 = 16;
```

Parameters

- config – Pointer to the LCDIC configuration.

void LCDIC__ResetState(LCDIC_Type *base)

Reset the LCDIC.

This function resets the LCDIC state. After calling this function, all data in TX_FIFO and RX_FIFO will be cleared and all transactions on LCD interface will restart despite of formal status.

The configurations will not be reset.

Parameters

- base – LCDIC peripheral base address.

static inline void LCDIC__EnableInterrupts(LCDIC_Type *base, uint32_t interrupts)

Enables LCDIC interrupts.

Parameters

- base – LCDIC peripheral base address.
- interrupts – The interrupts to enable, pass in as OR'ed value of _ldic_interrupt.

static inline void LCDIC__DisableInterrupts(LCDIC_Type *base, uint32_t interrupts)

Disable LCDIC interrupts.

Parameters

- base – LCDIC peripheral base address.
- interrupts – The interrupts to disable, pass in as OR'ed value of _ldic_interrupt.

static inline uint32_t LCDIC__GetInterruptStatus(LCDIC_Type *base)

Get LCDIC interrupt pending status.

Note: The interrupt must be enabled, otherwise the interrupt flags will not assert.

Parameters

- base – LCDIC peripheral base address.

Returns

The interrupt pending status.

```
static inline uint32_t LCDIC_GetInterruptRawStatus(LCDIC_Type *base)
```

Get LCDIC raw interrupt status.

This function gets the raw interrupt pending flags, it is not affected by interrupt enabled status.

Parameters

- base – LCDIC peripheral base address.

Returns

The raw interrupt status.

```
static inline void LCDIC_ClearInterruptStatus(LCDIC_Type *base, uint32_t interrupts)
```

Clear LCDIC interrupt status.

Parameters

- base – LCDIC peripheral base address.
- interrupts – The interrupt status to clear , pass in as OR'ed value of `_ldic_interrupt`.

```
static inline uint32_t LCDIC_GetStatusFlags(LCDIC_Type *base)
```

Get LCDIC status flags.

Note: The interval between two times calling this function shall be larger than one LCDIC function clock.

Parameters

- base – LCDIC peripheral base address.

Returns

The status flags, it is OR'ed value of `_ldic_flags`.

```
static inline uint32_t LCDIC_GetProcessingTrxCmd(LCDIC_Type *base)
```

Get current on-going LCDIC TRX-CMD.

Note: The interval between two times calling this function shall be larger than one LCDIC function clock.

Parameters

- base – LCDIC peripheral base address.

Returns

The TRX-CMD on-going.

```
static inline void LCDIC_SetTxThreshold(LCDIC_Type *base, ldic_tx_threshold_t threshold)
```

Set TX FIFO threshold.

Parameters

- base – LCDIC peripheral base address.
- threshold – TX threshold.

```
static inline void LCDIC_SetRxThreshold(LCDIC_Type *base, ldic_rx_threshold_t threshold)
```

Set RX FIFO threshold.

Parameters

- base – LCDIC peripheral base address.

- threshold – RX threshold.

status_t LCDIC_WriteTxFifoBlocking(LCDIC_Type *base, const uint32_t *data, uint32_t dataLen_Word)

Write the TX FIFO using blocking way.

This function waits for empty slot in TX FIFO and fill the data to TX FIFO.

Parameters

- base – LCDIC peripheral base address.
- data – Data to send, the data length must be dividable by 4.
- dataLen_Word – Data length in word.

Return values

- kStatus_Success – Write successfully.
- kStatus_Timeout – Timeout happened.

status_t LCDIC_ReadRxFifoBlocking(LCDIC_Type *base, uint32_t *data, uint32_t dataLen_Word)

Read the RX FIFO using blocking way.

This function waits for valid data in RX FIFO and read them.

Parameters

- base – LCDIC peripheral base address.
- data – Array for received data, the data length must be dividable by 4.
- dataLen_Word – Data length in word.

Return values

- kStatus_Success – Read successfully.
- kStatus_Timeout – Timeout happened.

static inline void LCDIC_SendResetSequence(LCDIC_Type *base)

Send reset sequence to the reset pin.

The function sends reset to reset pin, to reset the external panel. The reset sequence parameters are configured by *lcdic_config_t*.

Parameters

- base – LCDIC peripheral base address.

void LCDIC_SetResetSequenceDoneCallback(*lcdic_reset_done_callback_t* callback)

Set the callback called when reset sequence sent done.

Parameters

- callback – The callback to set.

static inline void LCDIC_EnableDMA(LCDIC_Type *base, bool enable)

Enable or disable to trigger DMA.

Parameters

- base – LCDIC peripheral base address.
- enable – Use true to enable, false to disable.

status_t LCDIC_SendCommandBlocking(LCDIC_Type *base, uint8_t cmd)

Send command using blocking way.

This function sends out command and waits until send finished.

Parameters

- `base` – LCDIC peripheral base address.
- `cmd` – Command to send.

Return values

`kStatus__Success` – Command sent successfully.

`status_t` LCDIC_SendRepeatDataBlocking(LCDIC_Type *base, const *lcdic_repeat_tx_xfer_t* *xfer)

Send repeat data using blocking way.

This function sends out command and the repeat data, then waits until send finished or timeout happened.

Parameters

- `base` – LCDIC peripheral base address.
- `xfer` – Pointer to the transfer configuration.

Return values

- `kStatus__Success` – Sent successfully.
- `kStatus__Timeout` – Timeout happened.
- `kStatus__InvalidArgument` – Invalid argument.

`status_t` LCDIC_SendDataArrayBlocking(LCDIC_Type *base, const *lcdic_tx_xfer_t* *xfer)

Send data array using blocking way.

This function sends out command and the data array, then waits until send finished or timeout happened.

Parameters

- `base` – LCDIC peripheral base address.
- `xfer` – Pointer to the transfer configuration.

Return values

- `kStatus__Success` – Sent successfully.
- `kStatus__Timeout` – Timeout happened.
- `kStatus__InvalidArgument` – Invalid argument.

`status_t` LCDIC_ReadDataArrayBlocking(LCDIC_Type *base, const *lcdic_rx_xfer_t* *xfer)

Read data array using blocking way.

This function sends out command and read the data array, then waits until send finished or timeout happened.

Parameters

- `base` – LCDIC peripheral base address.
- `xfer` – Pointer to the transfer configuration.

Return values

- `kStatus__Success` – Sent successfully.
- `kStatus__Timeout` – Timeout happened.
- `kStatus__InvalidArgument` – Invalid argument.

`status_t` LCDIC_TransferBlocking(LCDIC_Type *base, const *lcdic_xfer_t* *xfer)

LCDIC data transfer using blocking way.

This function sends command only, or sends repeat data, or sends data array, or reads data array based on the transfer structure. It uses blocking way, only returns when transfer succeeded or failed.

Parameters

- base – LCDIC peripheral base address.
- xfer – Pointer to the transfer configuration.

Return values

- kStatus_Success – Sent successfully.
- kStatus_Timeout – Timeout happened.
- kStatus_InvalidArgument – Invalid argument.

status_t LCDIC_TransferCreateHandle(LCDIC_Type *base, *lcdic_handle_t* *handle, *lcdic_transfer_callback_t* callback, void *userData)

Initializes the LCDIC driver handle, which is used in transactional functions.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- callback – The callback function.
- userData – The parameter of the callback function.

Return values

kStatus_Success – Successfully created the handle.

status_t LCDIC_TransferNonBlocking(LCDIC_Type *base, *lcdic_handle_t* *handle, *lcdic_xfer_t* *xfer)

Transfer data using IRQ.

This function transfer data using IRQ. This is a non-blocking function, which returns right away. When all data is sent out/received, or timeout happened, the callback function is called.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- xfer – LCDIC transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_Busy – LCDIC driver is busy with another transfer.

status_t LCDIC_SendCommandNonBlocking(LCDIC_Type *base, *lcdic_handle_t* *handle, uint8_t cmd)

Send command using interrupt-driven way.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- cmd – Command to send.

Return values

- kStatus_Success – Command sent successfully.
- kStatus_Busy – LCDIC driver is busy with another transfer.

```
status_t LCDIC_SendRepeatDataNonBlocking(LCDIC_Type *base, lcdic_handle_t *handle, const  
                                         lcdic_repeat_tx_xfer_t *xfer)
```

Send repeat data using interrupt-driven way.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- xfer – LCDIC transfer structure.

Return values

- *kStatus_Success* – Successfully start a transfer.
- *kStatus_InvalidArgument* – Input argument is invalid.
- *kStatus_Busy* – LCDIC driver is busy with another transfer.

```
status_t LCDIC_SendDataArrayNonBlocking(LCDIC_Type *base, lcdic_handle_t *handle, const  
                                         lcdic_tx_xfer_t *xfer)
```

Send data array using interrupt-driven way.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- xfer – LCDIC transfer structure.

Return values

- *kStatus_Success* – Successfully start a transfer.
- *kStatus_InvalidArgument* – Input argument is invalid.
- *kStatus_Busy* – LCDIC driver is busy with another transfer.

```
status_t LCDIC_ReadDataArrayNonBlocking(LCDIC_Type *base, lcdic_handle_t *handle, const  
                                         lcdic_rx_xfer_t *xfer)
```

Read data array using interrupt-driven way.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- xfer – LCDIC transfer structure.

Return values

- *kStatus_Success* – Successfully start a transfer.
- *kStatus_InvalidArgument* – Input argument is invalid.
- *kStatus_Busy* – LCDIC driver is busy with another transfer.

```
void LCDIC_TransferHandleIRQ(LCDIC_Type *base, void *handle)
```

LCDIC IRQ handler function.

IRQ handler to work with LCDIC_TransferNonBlocking.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.

```
void LCDIC__TransferInstallIRQHandler(uint32_t instance, void *handle,  
                                     lcdic_transfer_irq_handler_t handler)
```

Install the IRQ handler.

Install IRQ handler for specific instance.

Parameters

- instance – LCDIC instance.
- handle – Driver handle, it will be used as IRQ handler parameter.
- handler – IRQ handler to instance.

```
uint32_t LCDIC__GetInstance(LCDIC_Type *base)
```

Get the instance from the base address.

Parameters

- base – LCDIC peripheral base address

Returns

The LCDIC module instance

```
IRQn_Type LCDIC__GetIRQn(uint32_t instance)
```

Get IRQn for specific instance.

Parameters

- instance – LCDIC instance.

Returns

The LCDIC IRQn.

```
uint32_t LCDIC__FillByteToWord(const uint8_t *bytes, uint8_t len)
```

Get data from byte array, and fill to 4-byte word.

LCDIC data registers only accept 4-byte data, but the user passed data might be not 4-byte size aligned. This function is used to construct the unaligned part to a word, to write to LCDIC register.

Parameters

- bytes – The byte array.
- len – Length of the byte array.

Returns

The construct word.

```
void LCDIC__ExtractByteFromWord(uint32_t word, uint8_t *bytes, uint8_t len)
```

Get data from 4-byte, and fill to byte array.

LCDIC data registers only accept 4-byte data, but the user passed data might be not 4-byte size aligned. This function is used to get desired bytes from the word read from LCDIC register, and save to the user data array.

Parameters

- word – Word data read from LCDIC register.
- bytes – The byte array.
- len – Length of the byte array.

```
status_t LCDIC__PrepareSendCommand(LCDIC_Type *base, uint8_t cmd)
```

Prepare the command sending.

Fill the TRX command and command to TX FIFO, after calling this function, user should wait for transfer done by checking status or IRQ.

Parameters

- base – LCDIC peripheral base address.
- cmd – Command to send.

Return values

kStatus_Success – Operation succeeded.

status_t LCDIC_PrepareSendRepeatData(LCDIC_Type *base, const *lcdic_repeat_tx_xfer_t* *xfer)

Prepare the repeat data sending.

Fill the required data to TX FIFO, after calling this function, user should wait for transfer done by checking status or IRQ.

Parameters

- base – LCDIC peripheral base address.
- xfer – Transfer structure.

Return values

- kStatus_Success – Operation succeeded.
- kStatus_InvalidArgument – Invalid argument.

status_t LCDIC_PrepareSendDataArray(LCDIC_Type *base, const *lcdic_tx_xfer_t* *xfer, uint32_t *xferSizeWordAligned, uint8_t *xferSizeWordUnaligned, uint32_t *wordUnalignedData)

Prepare sending data array.

Fill the required command data to TX FIFO, after calling this function, user should fill the xfer->txData to TX FIFO based on FIFO status.

Parameters

- base – LCDIC peripheral base address.
- xfer – Transfer structure.
- xferSizeWordAligned – The word size aligned part of the transfer data.
- xferSizeWordUnaligned – The word size unaligned part of the transfer data.
- wordUnalignedData – Word to save the word size unaligned data, it should be sent after all word size aligned data write finished.

Return values

- kStatus_Success – Operation succeeded.
- kStatus_InvalidArgument – Invalid argument.

status_t LCDIC_PrepareReadDataArray(LCDIC_Type *base, const *lcdic_rx_xfer_t* *xfer, uint32_t *xferSizeWordAligned, uint8_t *xferSizeWordUnaligned)

Prepare reading data array.

Fill the required command data to TX FIFO, after calling this function, user should read RX FIFO to xfer->rxData based on FIFO status.

Parameters

- base – LCDIC peripheral base address.
- xfer – Transfer structure.
- xferSizeWordAligned – The word size aligned part of the transfer data.
- xferSizeWordUnaligned – The word size unaligned part of the transfer data.

Return values

- kStatus_Success – Operation succeeded.
- kStatus_InvalidArgument – Invalid argument.

FSL_LCDIC_DRIVER_VERSION

enum _ldic_mode

LCDIC mode.

Values:

enumerator kLCDIC_3WireSPI

3-wire SPI mode.

enumerator kLCDIC_4WireSPI

4-wire SPI mode.

enumerator kLCDIC_I8080

I8080 mode.

enum _ldic_endian

LCDIC byte order data endian.

Values:

enumerator kLCDIC_BigEndian

Big endian.

enumerator kLCDIC_LittleEndian

Little endian.

enum _ldic_rx_threshold

LCDIC RX FIFO threshold.

RX threshold interrupt happens if the occupied word number in RX FIFO is bigger than the threshold value.

Values:

enumerator kLCDIC_RxThreshold0Word

0 word.

enumerator kLCDIC_RxThreshold1Word

1 word.

enum _ldic_tx_threshold

LCDIC TX FIFO threshold.

TX threshold interrupt happens if the empty word number in TX FIFO is bigger than the threshold value.

Values:

enumerator kLCDIC_TxThreshold0Word

0 word.

enumerator kLCDIC_TxThreshold1Word

1 word.

enumerator kLCDIC_TxThreshold2Word

2 word.

enumerator kLCDIC_TxThreshold3Word

3 word.

enumerator kLCDIC__TxThreshold4Word
4 word.

enumerator kLCDIC__TxThreshold5Word
5 word.

enumerator kLCDIC__TxThreshold6Word
6 word.

enumerator kLCDIC__TxThreshold7Word
7 word.

enum _ldic_reset_polarity
LCDIC reset signal polarity.

Values:

enumerator kLCDIC__ResetActiveLow
Active low.

enumerator kLCDIC__ResetActiveHigh
Active high.

LCDIC I8080 control flags. .

Values:

enumerator kLCDIC__I8080__CsActiveLow
CS active low.

enumerator kLCDIC__I8080__CsActiveHigh
CS active high.

enumerator kLCDIC__I8080__DcCmdLow
DC 0 means command, while 1 means data.

enumerator kLCDIC__I8080__DcCmdHigh
DC 1 means command, while 0 means data.

enumerator kLCDIC__I8080__RdActiveLow
RD active low.

enumerator kLCDIC__I8080__RdActiveHigh
RD active high.

enumerator kLCDIC__I8080__WrActiveLow
WR active low.

enumerator kLCDIC__I8080__WrActiveHigh
WR active high.

enumerator kLCDIC__I8080__CsEnableIdleOff
CS off while no transmission.

enumerator kLCDIC__I8080__CsEnableDcSwitchOff
CS off while DC switches.

LCDIC SPI mode control flags. .

Values:

enumerator kLCDIC_SPI_MsbFirst

MSB(bit 7) sent and received first.

enumerator kLCDIC_SPI_LsbFirst

LSB(bit 0) sent and received first.

enumerator kLCDIC_SPI_ClkActiveHigh

CPOL=0. Clock active-high (idle low)

enumerator kLCDIC_SPI_ClkActiveLow

CPOL=1. Clock active-low (idle high)

enumerator kLCDIC_SPI_ClkPhaseFirstEdge

CPHA=0. Data sample at first clock edge.

enumerator kLCDIC_SPI_ClkPhaseSecondEdge

CPHA=1. Data sample at second clock edge.

enumerator kLCDIC_SPI_DcCmdLow

DC 0 means command, while 1 means data.

enumerator kLCDIC_SPI_DcCmdHigh

DC 1 means command, while 0 means data.

enum _ldic_interrupt

LCDIC interrupts.

Values:

enumerator kLCDIC_ResetDoneInterrupt

enumerator kLCDIC_CmdDoneInterrupt

enumerator kLCDIC_CmdTimeoutInterrupt

enumerator kLCDIC_TeTimeoutInterrupt

enumerator kLCDIC_TxOverflowInterrupt

enumerator kLCDIC_TxThresholdInterrupt

enumerator kLCDIC_RxUnderflowInterrupt

enumerator kLCDIC_RxThresholdInterrupt

enumerator kLCDIC_AllInterrupt

All interrupts.

LCDIC status flags. .

Values:

enumerator kLCDIC_IdleFlag

enumerator kLCDIC_TxThresholdFlag

enumerator kLCDIC_TxFullFlag

enumerator kLCDIC_RxThresholdFlag

enumerator kLCDIC_RxEmptyFlag

enumerator kLCDIC_AllFlag

All flags.

LCDIC TE sync mode .

Values:

enumerator kLCDIC__TeNoSync

Don't need to sync.

enumerator kLCDIC__TeRisingEdgeSync

Sync to TE rising edge.

enumerator kLCDIC__TeFallingEdgeSync

Sync to TE falling edge.

LCDIC TRX command timeout mode .

Values:

enumerator kLCDIC__ShortTimeout

Using short timeout.

enumerator kLCDIC__LongTimeout

Using long timeout.

LCDIC data format .

Values:

enumerator kLCDIC__DataFormatByte

Byte.

enumerator kLCDIC__DataFormatHalfWord

Half word (2-byte).

enumerator kLCDIC__DataFormatWord

Word (4-byte).

LCDIC data or command .

Values:

enumerator kLCDIC__Command

Command.

enumerator kLCDIC__Data

Data.

LCDIC TX or RX .

Values:

enumerator kLCDIC__RX

RX

enumerator kLCDIC__TX

TX.

enum lcdic_xfer_mode_t

LCDIC transfer mode.

Values:

enumerator `kLCDIC__XferCmdOnly`

Only send command.

enumerator `kLCDIC__XferSendRepeatData`

Send repeat data.

enumerator `kLCDIC__XferSendDataArray`

Send data array.

enumerator `kLCDIC__XferReceiveDataArray`

Receive data array.

typedef enum `_lcdic_mode` `lcdic_mode_t`

LCDIC mode.

typedef enum `_lcdic_endian` `lcdic_endian_t`

LCDIC byte order data endian.

typedef enum `_lcdic_rx_threshold` `lcdic_rx_threshold_t`

LCDIC RX FIFO threshold.

RX threshold interrupt happens if the occupied word number in RX FIFO is bigger than the threshold value.

typedef enum `_lcdic_tx_threshold` `lcdic_tx_threshold_t`

LCDIC TX FIFO threshold.

TX threshold interrupt happens if the empty word number in TX FIFO is bigger than the threshold value.

typedef enum `_lcdic_reset_polarity` `lcdic_reset_polarity_t`

LCDIC reset signal polarity.

typedef struct `_lcdic_config` `lcdic_config_t`

LCDIC configuration.

typedef union `_lcdic_trx_cmd` `lcdic_trx_cmd_t`

LCDIC TRX command.

typedef struct `_lcdic_repeat_tx_xfer` `lcdic_repeat_tx_xfer_t`

LCDIC repeat data TX transfer structure.

typedef struct `_lcdic_tx_xfer` `lcdic_tx_xfer_t`

LCDIC data array TX transfer structure.

typedef struct `_lcdic_rx_xfer` `lcdic_rx_xfer_t`

LCDIC data array RX transfer structure.

typedef struct `_lcdic_xfer` `lcdic_xfer_t`

LCDIC transfer structure.

typedef struct `_lcdic_handle` `lcdic_handle_t`

typedef void (*`lcdic_transfer_callback_t`)(`LCDIC_Type` *`base`, `lcdic_handle_t` *`handle`, `status_t` `status`, void *`userData`)

LCDIC transfer callback function.

The status is `kStatus_Success` when transfer finished successfully, it is `kStatus_Timeout` when timeout happened.

typedef void (*`lcdic_transfer_irq_handler_t`)(`LCDIC_Type` *`base`, void *`handle`)

Typedef for transactional APIs IRQ handler.

```
typedef void (*lcdc_reset_done_callback_t)(LCDIC_Type *base)
```

Typedef for reset sequence sent done callback.

```
LCDIC_RESET_STATE_DELAY
```

Delay used in LCDIC_ResetState.

This should be larger than 5 * core clock / LCDIC function clock.

```
LCDIC_WAIT_CMD_DONE_TIMEOUT
```

How many loops the driver will wait when waiting for command execution done.

LCDIC hardware provides command timeout (kLCDIC_CmdTimeoutInterrupt) and TE timeout feature (kLCDIC_TeTimeoutInterrupt). When LCDIC driver waits for a command finish, it checks the hardware timeout flags, and returns directly when timeout occurs.

Besides the hardware timeout, LCDIC driver provides a software timeout feature, macro LCDIC_WAIT_CMD_DONE_TIMEOUT defines how many times will LCDIC driver check the hardware flags while waiting for command execution done. If kLCDIC_CmdDoneInterrupt, kLCDIC_CmdTimeoutInterrupt, and kLCDIC_TeTimeoutInterrupt flags are not set after LCDIC_WAIT_CMD_DONE_TIMEOUT times check, LCDIC driver will return kStatus_Timeout.

Define LCDIC_WAIT_CMD_DONE_TIMEOUT as 0 will disable this software timeout feature.

```
LCDIC_MAX_BYTE_PER_TRX
```

```
struct _lcdic_config
```

#include <fsl_lcdc.h> LCDIC configuration.

Public Members

```
lcdic_mode_t mode
```

LCDIC work mode.

```
lcdic_endian_t endian
```

Data endian.

```
lcdic_rx_threshold_t rxThreshold
```

RX FIFO threshold.

```
lcdic_tx_threshold_t txThreshold
```

TX FIFO threshold.

```
uint8_t timerRatio0
```

Valid range: 0~15. $\text{freq}(\text{timer0}) = \text{freq}(\text{lcdic_clk}) / (2^{\text{timerRatio0}})$.

```
uint8_t timerRatio1
```

Valid range: 0~15. $\text{freq}(\text{timer1}) = \text{freq}(\text{timer0}) / (2^{\text{timerRatio1}})$.

```
uint8_t resetPulseWidth_Timer0
```

Reset pulse width, in the unit of timer0 period. Valid range 1 ~ 64.

```
uint8_t resetSequence
```

Reset sequence, it is a 8-bit value sent to reset pin from LSB.

```
uint8_t resetSequencePulseNum
```

Reset sequence pulse number, valid range is 1 ~ 8.

```
lcdic_reset_polarity_t resetPolarity
```

Reset signal polarity.

uint8_t i8080CtrlFlags

I8080 control flags, it is OR'ed value of _lddic_i8080_ctrl_flags.

uint8_t csWaitTime

Minimum CS inactive pulse width. $T(csw)=T(lcdic_clk)*csWaitTime$, valid range 0-7.

uint8_t csSetupTime

Minimum CS setup time before WR/RD. $T(css)=T(lcdic_clk)*csSetupTime$, valid range 0-255.

uint8_t csHoldTime

Minimum CS hold time after WR/RD. $T(csh)=T(lcdic_clk)*csHoldTime$, valid range 0-7.

uint8_t dcSetupTime

Minimum DC setup time before WR/RD/CS. $T(dcs)=T(lcdic_clk)*dsSetupTime$, valid range 0-7.

uint8_t dcHoldTime

Minimum DC hold time after WR/RD/CS. $T(dch)=T(lcdic_clk)*dsHoldTime$, valid range 0-7.

uint8_t writeDataSetupTime

Minimum write data setup time after WR active. $T(wdh)=T(lcdic_clk)*writeDataSetupTime$, valid range 0-7.

uint8_t writeDataHoldTime

Minimum write data setup time before WR active. $T(wds)=T(lcdic_clk)*writeDataHoldTime$, valid range 0-7.

uint8_t writeEnableActiveWidth

Minmum write enable active pulse width. $T(waw)=T(lcdic_clk)*writeEnableActiveWidth$, valid range 0-63.

uint8_t writeEnableInactiveWidth

Minmum write enable inactive pulse width. $T(wiw)=T(lcdic_clk)*writeEnableInactiveWidth$, valid range 0-63.

uint8_t readEnableActiveWidth

Minmum read enable active pulse width. $T(raw)=T(lcdic_clk)*readEnableActiveWidth$, valid range 0-255.

uint8_t readEnableInactiveWidth

Minmum read enable inactive pulse width. $T(riw)=T(lcdic_clk)*readEnableInactiveWidth$, valid range 0-255.

uint8_t spiCtrlFlags

SPI control flags, it is OR'ed value of _lddic_spi_ctrl_flags.

uint8_t teTimeoutTime_Timer1

Tearing effect timeout time. $T(te_to)=T(timer1)*teTimeoutTime_Timer1$.

uint8_t teSyncWaitTime_Timer1

Tearing effect signal synchronization wait time. $T(tew)=T(timer1)*teSyncWaitTime_Timer1$.

uint8_t cmdShortTimeout_Timer0

Command short timeout. $T(cmd_short_to)=T(timer0)*cmdShortTimeout_Timer0$.

uint8_t cmdLongTimeout_Timer1

Command long timeout. $T(cmd_long_to)=T(timer1)*cmdLongTimeout_Timer1$.

union _lddic_trx_cmd

#include <fsl_lddic.h> LCDIC TRX command.

Public Members

struct *_lcdc_trx_cmd* bits

uint32_t u32

struct *_lcdc_repeat_tx_xfer*

#include <fsl_lcdc.h> LCDIC repeat data TX transfer structure.

Public Members

uint8_t cmd

Command.

uint8_t teSyncMode

TE sync mode, see *_lcdc_te_sync_mode*.

uint8_t trxTimeoutMode

TRX command timeout mode, see *_lcdc_trx_timeout_mode*.

uint8_t dataFormat

Data format, see *_lcdc_data_format*.

uint32_t dataLen

Data length.

uint32_t txRepeatData

The repeat data.

struct *_lcdc_tx_xfer*

#include <fsl_lcdc.h> LCDIC data array TX transfer structure.

Public Members

uint8_t cmd

Command.

uint8_t teSyncMode

TE sync mode, see *_lcdc_te_sync_mode*.

uint8_t trxTimeoutMode

TRX command timeout mode, see *_lcdc_trx_timeout_mode*.

uint8_t dataFormat

Data format, see *_lcdc_data_format*.

uint32_t dataLen

Data length.

const uint8_t *txData

The data to send.

struct *_lcdc_rx_xfer*

#include <fsl_lcdc.h> LCDIC data array RX transfer structure.

Public Members

uint8_t cmd

Command.

uint8_t dummyCount

Dummy cycle between TX and RX, only used for SPI mode.

uint8_t trxTimeoutMode

TRX command timeout mode, see `_lcdc_trx_timeout_mode`.

uint8_t dataFormat

Data format, see `_lcdc_data_format`.

uint32_t dataLen

Data length.

uint8_t *rxData

Pointer to the data receive array.

struct _lcdc_xfer

#include <fsl_lcdc.h> LCDIC transfer structure.

Public Members

lcdc_xfer_mode_t mode

Transfer mode.

struct _lcdc_handle

#include <fsl_lcdc.h> LCDIC handle structure.

Public Members

volatile bool xferInProgress

Transfer in progress.

lcdc_xfer_mode_t xferMode

On-going transfer mode.

lcdc_transfer_callback_t callback

Callback function.

void *userData

LCDIC callback function parameter.

uint32_t xferSizeWordAligned

4-byte aligned part of the transfer size.

uint8_t xferSizeWordUnaligned

4-byte unaligned part of the transfer size.

uint32_t tmpData

Temp data for driver internal use.

struct bits

Public Members

uint32_t dataLen

Data length in bytes, transfered byte is dataLen + 1.

uint32_t dummyCount

Dummy cycle count between TX and RX (for SPI only).

uint32_t useAutoRepeat

Use auto repeat mode or not.

uint32_t teSyncMode

TE sync mode, see `_lcdcic_te_sync_mode`.

uint32_t trxTimeoutMode

TRX command timeout mode, see `_lcdcic_trx_timeout_mode`.

uint32_t dataFormat

Data format, see `_lcdcic_data_format`.

uint32_t enableCmdDoneInt

Enable command done interrupt or not.

uint32_t cmdOrData

Command or data, see `_lcdcic_dc`.

uint32_t trx

TX or RX, see `_lcdcic_trx`.

union `__unnamed36__`

Public Members

uint8_t cmdToSendOnly

Command to send in mode `kLCDIC_XferCmdOnly`.

lcdcic_repeat_tx_xfer_t repeatTxXfer

For mode `kLCDIC_XferSendRepeatData`.

lcdcic_tx_xfer_t txXfer

For mode `kLCDIC_XferSendDataArray`.

lcdcic_rx_xfer_t rxXfer

For mode `kLCDIC_XferReceiveDataArray`.

union `__unnamed38__`

Public Members

const uint8_t *txData

Data array to send.

uint8_t *rxData

RX data array.

2.37 LCDIC DMA Driver

```
status_t LCDIC_TransferCreateHandleDMA(LCDIC_Type *base, lcdic_dma_handle_t *handle,  
                                       lcdic_dma_callback_t callback, void *userData,  
                                       dma_handle_t *txDmaHandle, dma_handle_t  
                                       *rxDmaHandle, dma_descriptor_t dmaDesc[2])
```

Initialize the LCDIC DMA handle.

This function initializes the LCDIC DMA handle which can be used for other LCDIC transactional APIs. Usually, for a specified LCDIC instance, user need only call this API once to get the initialized handle.

Parameters

- base – LCDIC peripheral base address.
- handle – LCDIC handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txDmaHandle – DMA handle pointer for LCDIC Tx, the handle shall be static allocated by users.
- rxDmaHandle – DMA handle pointer for LCDIC Rx, the handle shall be static allocated by users.
- dmaDesc – User allocated dma descriptor, it should be in non-cacheable region and 16-byte aligned.

```
status_t LCDIC_TransferDMA(LCDIC_Type *base, lcdic_dma_handle_t *handle, const lcdic_xfer_t  
                           *xfer)
```

Perform a non-blocking LCDIC transfer using DMA.

This function returned immediately after transfer initiates, monitor the transfer done by callback.

Parameters

- base – LCDIC peripheral base address.
- handle – LCDIC DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_Busy – LCDIC is not idle, is running another transfer.

```
status_t LCDIC_SendDataArrayDMA(LCDIC_Type *base, lcdic_dma_handle_t *handle, const  
                                lcdic_tx_xfer_t *xfer)
```

Send data array using DMA.

Parameters

- base – LCDIC peripheral base address.
- handle – Pointer to the *lcdic_handle_t* structure to store the transfer state.
- xfer – LCDIC transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.

- `kStatus_InvalidArgument` – Input argument is invalid.
- `kStatus_Busy` – LCDIC driver is busy with another transfer.

```
status_t LCDIC_ReadDataArrayDMA(LCDIC_Type *base, lcdic_dma_handle_t *handle, const
                                lcdic_rx_xfer_t *xfer)
```

Read data array using DMA.

Parameters

- `base` – LCDIC peripheral base address.
- `handle` – Pointer to the `lcdic_dma_handle_t` structure to store the transfer state.
- `xfer` – LCDIC transfer structure.

Return values

- `kStatus_Success` – Successfully start a transfer.
- `kStatus_InvalidArgument` – Input argument is invalid.
- `kStatus_Busy` – LCDIC driver is busy with another transfer.

```
void LCDIC_TransferHandleIRQDMA(LCDIC_Type *base, void *handle)
```

LCDIC IRQ handler function work with DMA transactional APIs.

IRQ handler to work with `LCDIC_TransferDMA`.

Parameters

- `base` – LCDIC peripheral base address.
- `handle` – Pointer to the `lcdic_dma_handle_t` structure to store the transfer state.

```
FSL_LCDIC_DMA_DRIVER_VERSION
```

```
typedef struct lcdic_dma_handle lcdic_dma_handle_t
```

```
typedef void (*lcdic_dma_callback_t)(LCDIC_Type *base, lcdic_dma_handle_t *handle, status_t
status, void *userData)
```

LCDIC DMA callback called at the end of transfer.

```
struct lcdic_dma_handle
```

`#include <fsl_lcdic_dma.h>` LCDIC DMA transfer handle, users should not touch the content of the handle.

Public Members

```
volatile bool xferInProgress
```

Transfer in progress

```
lcdic_xfer_mode_t xferMode
```

On-going transfer mode.

```
dma_handle_t *txDmaHandle
```

DMA handler for send

```
dma_handle_t *rxDmaHandle
```

DMA handler for receive

```
lcdic_dma_callback_t callback
```

Callback when transfer finished.

`void *userData`
User Data for callback

`uint32_t xferSizeWordAligned`
4-byte size aligned part or the transfer data size.

`uint8_t xferSizeWordUnaligned`
4-byte size unaligned part of the transfer data size.

`uint8_t rxSizeWordUnaligned`
Same as `xferSizeWordUnaligned`, it is only used for RX.

`uint32_t tmpData`
To save temporary data during transfer.

`dma_descriptor_t *dmaDesc`
Pointer to two DMA descriptor, should be 16-byte aligned.

`union __unnamed25__`

Public Members

`const uint8_t *txData`
Pointer to the TX data.

`uint8_t *rxData`
Pointer to the RX data.

2.38 LCDIC: LCD Interface Controller

2.39 GPIO: General Purpose I/O

`void GPIO_PortInit(GPIO_Type *base, uint32_t port)`
Initializes the GPIO peripheral.
This function ungates the GPIO clock.

Parameters

- `base` – GPIO peripheral base pointer.
- `port` – GPIO port number.

`void GPIO_PinInit(GPIO_Type *base, uint32_t port, uint32_t pin, const gpio_pin_config_t *config)`

Initializes a GPIO pin used by the board.

To initialize the GPIO, define a pin configuration, either input or output, in the user file. Then, call the `GPIO_PinInit()` function.

This is an example to define an input pin or output pin configuration:

```
Define a digital input pin configuration,
gpio_pin_config_t config =
{
    kGPIO_DigitalInput,
    0,
}
Define a digital output pin configuration,
```

(continues on next page)

(continued from previous page)

```
gpio_pin_config_t config =
{
    kGPIO_DigitalOutput,
    0,
}
```

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number
- config – GPIO pin configuration pointer

static inline void GPIO_PinWrite(GPIO_Type *base, uint32_t port, uint32_t pin, uint8_t output)

Sets the output level of the one GPIO pin to the logic 1 or 0.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number
- output – GPIO pin output logic level.
 - 0: corresponding pin output low-logic level.
 - 1: corresponding pin output high-logic level.

static inline uint32_t GPIO_PinRead(GPIO_Type *base, uint32_t port, uint32_t pin)

Reads the current input value of the GPIO PIN.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- pin – GPIO pin number

Return values

GPIO – port input value

- 0: corresponding pin input low-logic level.
- 1: corresponding pin input high-logic level.

FSL_GPIO_DRIVER_VERSION

LPC GPIO driver version.

enum _gpio_pin_direction

LPC GPIO direction definition.

Values:

enumerator kGPIO_DigitalInput

Set current pin as digital input

enumerator kGPIO_DigitalOutput

Set current pin as digital output

enum `_gpio_pin_enable_mode`

GPIO Pin Interrupt enable mode.

Values:

enumerator `kGPIO_PinIntEnableLevel`

Generate Pin Interrupt on level mode

enumerator `kGPIO_PinIntEnableEdge`

Generate Pin Interrupt on edge mode

enum `_gpio_pin_enable_polarity`

GPIO Pin Interrupt enable polarity.

Values:

enumerator `kGPIO_PinIntEnableHighOrRise`

Generate Pin Interrupt on high level or rising edge

enumerator `kGPIO_PinIntEnableLowOrFall`

Generate Pin Interrupt on low level or falling edge

enum `_gpio_interrupt_index`

LPC GPIO interrupt index definition.

Values:

enumerator `kGPIO_InterruptA`

Set current pin as interrupt A

enumerator `kGPIO_InterruptB`

Set current pin as interrupt B

typedef enum `_gpio_pin_direction` `gpio_pin_direction_t`

LPC GPIO direction definition.

typedef struct `_gpio_pin_config` `gpio_pin_config_t`

The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

typedef enum `_gpio_pin_enable_mode` `gpio_pin_enable_mode_t`

GPIO Pin Interrupt enable mode.

typedef enum `_gpio_pin_enable_polarity` `gpio_pin_enable_polarity_t`

GPIO Pin Interrupt enable polarity.

typedef enum `_gpio_interrupt_index` `gpio_interrupt_index_t`

LPC GPIO interrupt index definition.

typedef struct `_gpio_interrupt_config` `gpio_interrupt_config_t`

Configures the interrupt generation condition.

static inline void `GPIO_PortSet(GPIO_Type *base, uint32_t port, uint32_t mask)`

Sets the output level of the multiple GPIO pins to the logic 1.

Parameters

- `base` – GPIO peripheral base pointer(Typically GPIO)
- `port` – GPIO port number
- `mask` – GPIO pin number macro

```
static inline void GPIO_PortClear(GPIO_Type *base, uint32_t port, uint32_t mask)
```

Sets the output level of the multiple GPIO pins to the logic 0.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- mask – GPIO pin number macro

```
static inline void GPIO_PortToggle(GPIO_Type *base, uint32_t port, uint32_t mask)
```

Reverses current output logic of the multiple GPIO pins.

Parameters

- base – GPIO peripheral base pointer(Typically GPIO)
- port – GPIO port number
- mask – GPIO pin number macro

GPIO_PIN_INT_LEVEL

GPIO_PIN_INT_EDGE

PINT_PIN_INT_HIGH_OR_RISE_TRIGGER

PINT_PIN_INT_LOW_OR_FALL_TRIGGER

```
struct _gpio_pin_config
```

#include <fsl_gpio.h> The GPIO pin configuration structure.

Every pin can only be configured as either output pin or input pin at a time. If configured as a input pin, then leave the outputConfig unused.

Public Members

gpio_pin_direction_t pinDirection

GPIO direction, input or output

uint8_t outputLogic

Set default output logic, no use in input

```
struct _gpio_interrupt_config
```

#include <fsl_gpio.h> Configures the interrupt generation condition.

2.40 MRT: Multi-Rate Timer

```
void MRT_Init(MRT_Type *base, const mrt_config_t *config)
```

Ungates the MRT clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application using the MRT driver.

Parameters

- base – Multi-Rate timer peripheral base address
- config – Pointer to user's MRT config structure. If MRT has MULTITASK bit field in MODCFG register, param config is useless.

```
void MRT_Deinit(MRT_Type *base)
```

Gate the MRT clock.

Parameters

- base – Multi-Rate timer peripheral base address

```
static inline void MRT_GetDefaultConfig(mrt_config_t *config)
```

Fill in the MRT config struct with the default settings.

The default values are:

```
config->enableMultiTask = false;
```

Parameters

- config – Pointer to user's MRT config structure.

```
static inline void MRT_SetupChannelMode(MRT_Type *base, mrt_chnl_t channel, const  
                                         mrt_timer_mode_t mode)
```

Sets up an MRT channel mode.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Channel that is being configured.
- mode – Timer mode to use for the channel.

```
static inline void MRT_EnableInterrupts(MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
```

Enables the MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `mrt_interrupt_enable_t`

```
static inline void MRT_DisableInterrupts(MRT_Type *base, mrt_chnl_t channel, uint32_t mask)
```

Disables the selected MRT interrupt.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The interrupts to disable. This is a logical OR of members of the enumeration `mrt_interrupt_enable_t`

```
static inline uint32_t MRT_GetEnabledInterrupts(MRT_Type *base, mrt_chnl_t channel)
```

Gets the enabled MRT interrupts.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `mrt_interrupt_enable_t`

static inline uint32_t MRT_GetStatusFlags(MRT_Type *base, *mrt_chnl_t* channel)

Gets the MRT status flags.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

The status flags. This is the logical OR of members of the enumeration *mrt_status_flags_t*

static inline void MRT_ClearStatusFlags(MRT_Type *base, *mrt_chnl_t* channel, uint32_t mask)

Clears the MRT status flags.

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- mask – The status flags to clear. This is a logical OR of members of the enumeration *mrt_status_flags_t*

void MRT_UpdateTimerPeriod(MRT_Type *base, *mrt_chnl_t* channel, uint32_t count, bool immediateLoad)

Used to update the timer period in units of count.

The new value will be immediately loaded or will be loaded at the end of the current time interval. For one-shot interrupt mode the new value will be immediately loaded.

Note: User can call the utility macros provided in *fsl_common.h* to convert to ticks

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number
- count – Timer period in units of ticks
- immediateLoad – true: Load the new value immediately into the TIMER register; false: Load the new value at the end of current timer interval

static inline uint32_t MRT_GetCurrentTimerCount(MRT_Type *base, *mrt_chnl_t* channel)

Reads the current timer counting value.

This function returns the real-time timer counting value, in a range from 0 to a timer period.

Note: User can call the utility macros provided in *fsl_common.h* to convert ticks to usec or msec

Parameters

- base – Multi-Rate timer peripheral base address
- channel – Timer channel number

Returns

Current timer counting value in ticks

```
static inline void MRT_StartTimer(MRT_Type *base, mrt_chnl_t channel, uint32_t count)
```

Starts the timer counting.

After calling this function, timers load period value, counts down to 0 and depending on the timer mode it will either load the respective start value again or stop.

Note: User can call the utility macros provided in `fsl_common.h` to convert to ticks

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.
- `count` – Timer period in units of ticks. Count can contain the LOAD bit, which control the force load feature.

```
static inline void MRT_StopTimer(MRT_Type *base, mrt_chnl_t channel)
```

Stops the timer counting.

This function stops the timer from counting.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

```
static inline uint32_t MRT_GetIdleChannel(MRT_Type *base)
```

Find the available channel.

This function returns the lowest available channel number.

Parameters

- `base` – Multi-Rate timer peripheral base address

```
static inline void MRT_ReleaseChannel(MRT_Type *base, mrt_chnl_t channel)
```

Release the channel when the timer is using the multi-task mode.

In multi-task mode, the INUSE flags allow more control over when MRT channels are released for further use. The user can hold on to a channel acquired by calling `MRT_GetIdleChannel()` for as long as it is needed and release it by calling this function. This removes the need to ask for an available channel for every use.

Parameters

- `base` – Multi-Rate timer peripheral base address
- `channel` – Timer channel number.

```
FSL_MRT_DRIVER_VERSION
```

```
enum _mrt_chnl
```

List of MRT channels.

Values:

```
enumerator kMRT_Channel_0
```

MRT channel number 0

```
enumerator kMRT_Channel_1
```

MRT channel number 1

```
enumerator kMRT_Channel_2
```

MRT channel number 2

enumerator kMRT_Channel_3
MRT channel number 3

enum _mrt_timer_mode
List of MRT timer modes.

Values:

enumerator kMRT_RepeatMode
Repeat Interrupt mode

enumerator kMRT_OneShotMode
One-shot Interrupt mode

enumerator kMRT_OneShotStallMode
One-shot stall mode

enum _mrt_interrupt_enable
List of MRT interrupts.

Values:

enumerator kMRT_TimerInterruptEnable
Timer interrupt enable

enum _mrt_status_flags
List of MRT status flags.

Values:

enumerator kMRT_TimerInterruptFlag
Timer interrupt flag

enumerator kMRT_TimerRunFlag
Indicates state of the timer

typedef enum _mrt_chnl mrt_chnl_t
List of MRT channels.

typedef enum _mrt_timer_mode mrt_timer_mode_t
List of MRT timer modes.

typedef enum _mrt_interrupt_enable mrt_interrupt_enable_t
List of MRT interrupts.

typedef enum _mrt_status_flags mrt_status_flags_t
List of MRT status flags.

typedef struct _mrt_config mrt_config_t
MRT configuration structure.

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the MRT_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

struct _mrt_config
#include <fsl_mrt.h> MRT configuration structure.

This structure holds the configuration settings for the MRT peripheral. To initialize this structure to reasonable defaults, call the MRT_GetDefaultConfig() function and pass a pointer to your config structure instance.

The config struct can be made const so it resides in flash

Public Members

`bool enableMultiTask`

true: Timers run in multi-task mode; false: Timers run in hardware status mode

2.41 This type defines status return values used by NBOOT functions that are not easily disturbed by Fault Attacks

`kStatus_NBOOT_Success`

Operation completed successfully.

`kStatus_NBOOT_Fail`

Operation failed.

`kStatus_NBOOT_InvalidArgument`

Invalid argument passed to the function.

`kStatus_NBOOT_RequestTimeout`

Operation timed out.

`kStatus_NBOOT_KeyNotLoaded`

The requested key is not loaded.

`kStatus_NBOOT_AuthFail`

Authentication failed.

`kStatus_NBOOT_OperationNotAvailable`

Operation not available on this HW.

`kStatus_NBOOT_KeyNotAvailable`

Key is not available.

`kStatus_NBOOT_IvCounterOverflow`

Overflow of IV counter (PRINCE/IPED).

`kStatus_NBOOT_SelftestFail`

FIPS self-test failure.

`kStatus_NBOOT_InvalidDataFormat`

Invalid data format for example antipole

`kStatus_NBOOT_IskCertUserDataTooBig`

Size of User data in ISK certificate is greater than 96 bytes

`kStatus_NBOOT_IskCertSignatureOffsetTooSmall`

Signature offset in ISK certificate is smaller than expected

`kStatus_NBOOT_MemcpyFail`

Unexpected error detected during `nboot_memcpy()`

`NXPCLCSS_HASH_RTF_OUTPUT_SIZE`

Size of run-time fingerprint appended to the hash in `pDigest` in bytes, if `#NXPCLCSS_HASH_RTF_OUTPUT_ENABLE` was specified.

`NXPCLHASH_WA_SIZE_MAX`

2.42 OCOTP Driver

OTP Status Group.

Values:

enumerator kStatusGroup__OtpGroup

OTP Error Status definitions.

Values:

enumerator kStatus_OTP_InvalidAddress

Invalid OTP address

enumerator kStatus_OTP_Timeout

OTP operation time out

status_t OCOTP__OtpInit(void)

Initialize OTP controller.

This function enables OTP Controller clock.

Returns

kStatus_Success

status_t OCOTP__OtpDeinit(void)

De-Initialize OTP controller.

This function disables OTP Controller Clock.

Returns

kStatus_Success

status_t OCOTP__OtpFuseRead(uint32_t addr, uint32_t *data)

Read Fuse value from OTP Fuse Block.

This function read fuse data from OTP Fuse block to specified data buffer.

Parameters

- addr – Fuse address
- data – Buffer to hold the data read from OTP Fuse block

Returns

kStatus_Success - Data read from OTP Fuse block successfully
 kStatus_OTP_Timeout - OTP read timeout
 kStatus_InvalidArgument - data pointer is invalid

status_t OCOTP__ReadSocOtp(uint64_t *data, uint32_t tag)

Read Fuse line with specific tag value from SoC OTP.

This function read Fuse line with specific tag value from SoC OTP to specified data buffer.

Parameters

- data – Buffer to hold the data read from SoC OTP
- tag – Tag value to match

Returns

kStatus_Success - Data read from SoC OTP successfully
 kStatus_Fail - Data read from SoC OTP failed, or cannot find the tag
 kStatus_InvalidArgument - data pointer is invalid

status_t OCOTP_ReadUniqueID(uint8_t *uid, uint32_t *idLen)

Read unique ID from OTP Fuse Block.

This function read unique ID from OTP Fuse block to specified data buffer.

Parameters

- uid – The buffer to store unique ID, buffer byte length is FSL_OCOTP_UID_LENGTH.
- idLen[in/out] – The unique ID byte length. Set the length to read, return the length read out.

Returns

kStatus_Success - Data read from OTP Fuse block successfully
kStatus_OTP_Timeout - OTP read timeout
kStatus_InvalidArgument - data pointer is invalid

status_t OCOTP_ReadSVC(uint64_t *svc)

Read Static Voltage Compansation from SOC OTP.

This function read SVC from OTP Fuse block to specified data buffer.

Parameters

- svc – The buffer to store SVC.

Returns

kStatus_Success - Data read from SOC OTP successfully
kStatus_Fail - SOC OTP read failure

status_t OCOTP_ReadPackage(uint32_t *pack)

Read package type from SOC OTP.

Parameters

- pack – The buffer to store package type.

Returns

kStatus_Success - Data read from SOC OTP successfully
kStatus_Fail - SOC OTP read failure

FSL_OCOTP_DRIVER_VERSION

OCOTP driver version 2.2.3.

FSL_OCOTP_UID_LENGTH

OCOTP unique ID length.

2.43 OSTIMER: OS Event Timer Driver

void OSTIMER_Init(OSTIMER_Type *base)

Initializes an OSTIMER by turning its bus clock on.

void OSTIMER_Deinit(OSTIMER_Type *base)

Deinitializes a OSTIMER instance.

This function shuts down OSTIMER bus clock

Parameters

- base – OSTIMER peripheral base address.

uint64_t OSTIMER_GrayToDecimal(uint64_t gray)

Translate the value from gray-code to decimal.

Parameters

- gray – The gray value input.

Returns

The decimal value.

static inline uint64_t OSTIMER_DecimalToGray(uint64_t dec)

Translate the value from decimal to gray-code.

Parameters

- dec – The decimal value.

Returns

The gray code of the input value.

uint32_t OSTIMER_GetStatusFlags(OSTIMER_Type *base)

Get OSTIMER status Flags.

This returns the status flag. Currently, only match interrupt flag can be got.

Parameters

- base – OSTIMER peripheral base address.

Returns

status register value

void OSTIMER_ClearStatusFlags(OSTIMER_Type *base, uint32_t mask)

Clear Status Interrupt Flags.

This clears intrrupt status flag. Currently, only match interrupt flag can be cleared.

Parameters

- base – OSTIMER peripheral base address.
- mask – Clear bit mask.

Returns

none

status_t OSTIMER_SetMatchRawValue(OSTIMER_Type *base, uint64_t count, *ostimer_callback_t* cb)

Set the match raw value for OSTIMER.

This function will set a match value for OSTIMER with an optional callback. And this callback will be called while the data in dedicated pair match register is equals to the value of central EVTIMER. Please note that, the data format is gray-code, if decimal data was desired, please using OSTIMER_SetMatchValue().

Parameters

- base – OSTIMER peripheral base address.
- count – OSTIMER timer match value.(Value is gray-code format)
- cb – OSTIMER callback (can be left as NULL if none, otherwise should be a void func(void)).

Return values

- kStatus_Success -- Set match raw value and enable interrupt Successfully.
- kStatus_Fail -- Set match raw value fail.

status_t OSTIMER_SetMatchValue(OSTIMER_Type *base, uint64_t count, *ostimer_callback_t* cb)

Set the match value for OSTIMER.

This function will set a match value for OSTIMER with an optional callback. And this callback will be called while the data in dedicated pair match register is equals to the value of central OS TIMER.

Parameters

- *base* – OSTIMER peripheral base address.
- *count* – OSTIMER timer match value.(Value is decimal format, and this value will be translate to Gray code internally.)
- *cb* – OSTIMER callback (can be left as NULL if none, otherwise should be a void func(void)).

Return values

- *kStatus_Success* – - Set match value and enable interrupt Successfully.
- *kStatus_Fail* – - Set match value fail.

static inline void OSTIMER_SetMatchRegister(OSTIMER_Type *base, uint64_t value)

Set value to OSTIMER MATCH register directly.

This function writes the input value to OSTIMER MATCH register directly, it does not touch any other registers. Note that, the data format is gray-code. The function OSTIMER_DecimalToGray could convert decimal value to gray code.

Parameters

- *base* – OSTIMER peripheral base address.
- *value* – OSTIMER timer match value (Value is gray-code format).

static inline void OSTIMER_EnableMatchInterrupt(OSTIMER_Type *base)

Enable the OSTIMER counter match interrupt.

Enable the timer counter match interrupt. The interrupt happens when OSTIMER counter matches the value in MATCH registers.

Parameters

- *base* – OSTIMER peripheral base address.

static inline void OSTIMER_DisableMatchInterrupt(OSTIMER_Type *base)

Disable the OSTIMER counter match interrupt.

Disable the timer counter match interrupt. The interrupt happens when OSTIMER counter matches the value in MATCH registers.

Parameters

- *base* – OSTIMER peripheral base address.

static inline uint64_t OSTIMER_GetCurrentTimerRawValue(OSTIMER_Type *base)

Get current timer raw count value from OSTIMER.

This function will get a gray code type timer count value from OS timer register. The raw value of timer count is gray code format.

Parameters

- *base* – OSTIMER peripheral base address.

Returns

Raw value of OSTIMER, gray code format.

```
uint64_t OSTIMER_GetCurrentTimerValue(OSTIMER_Type *base)
```

Get current timer count value from OSTIMER.

This function will get a decimal timer count value. The RAW value of timer count is gray code format, will be translated to decimal data internally.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of OSTIMER which will be formatted to decimal value.

```
static inline uint64_t OSTIMER_GetCaptureRawValue(OSTIMER_Type *base)
```

Get the capture value from OSTIMER.

This function will get a captured gray-code value from OSTIMER. The Raw value of timer capture is gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of capture register, data format is gray code.

```
uint64_t OSTIMER_GetCaptureValue(OSTIMER_Type *base)
```

Get the capture value from OSTIMER.

This function will get a capture decimal-value from OSTIMER. The RAW value of timer capture is gray code format, will be translated to decimal data internally.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of capture register, data format is decimal.

```
void OSTIMER_HandleIRQ(OSTIMER_Type *base, ostimer_callback_t cb)
```

OS timer interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in OSTIMER_SetMatchValue()). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- base – OS timer peripheral base address.
- cb – callback scheduled for this instance of OS timer

Returns

none

```
FSL_OSTIMER_DRIVER_VERSION
```

OSTIMER driver version.

```
enum __ostimer_flags
```

OSTIMER status flags.

Values:

```
enumerator kOSTIMER_MatchInterruptFlag
```

Match interrupt flag bit, sets if the match value was reached.

```
typedef void (*ostimer_callback_t)(void)
```

ostimer callback function.

2.44 PINT: Pin Interrupt and Pattern Match Driver

FSL_PINT_DRIVER_VERSION

enum __pint_pin_enable

PINT Pin Interrupt enable type.

Values:

enumerator kPINT_PinIntEnableNone

Do not generate Pin Interrupt

enumerator kPINT_PinIntEnableRiseEdge

Generate Pin Interrupt on rising edge

enumerator kPINT_PinIntEnableFallEdge

Generate Pin Interrupt on falling edge

enumerator kPINT_PinIntEnableBothEdges

Generate Pin Interrupt on both edges

enumerator kPINT_PinIntEnableLowLevel

Generate Pin Interrupt on low level

enumerator kPINT_PinIntEnableHighLevel

Generate Pin Interrupt on high level

enum __pint_int

PINT Pin Interrupt type.

Values:

enumerator kPINT_PinInt0

Pin Interrupt 0

enum __pint_pmatch_input_src

PINT Pattern Match bit slice input source type.

Values:

enumerator kPINT_PatternMatchInp0Src

Input source 0

enumerator kPINT_PatternMatchInp1Src

Input source 1

enumerator kPINT_PatternMatchInp2Src

Input source 2

enumerator kPINT_PatternMatchInp3Src

Input source 3

enumerator kPINT_PatternMatchInp4Src

Input source 4

enumerator kPINT_PatternMatchInp5Src

Input source 5

enumerator kPINT_PatternMatchInp6Src

Input source 6

enumerator kPINT_PatternMatchInp7Src

Input source 7

enumerator kPINT_SecPatternMatchInp0Src
Input source 0

enumerator kPINT_SecPatternMatchInp1Src
Input source 1

enum __pint_pmatch_bslice
PINT Pattern Match bit slice type.

Values:

enumerator kPINT_PatternMatchBSlice0
Bit slice 0

enum __pint_pmatch_bslice_cfg
PINT Pattern Match configuration type.

Values:

enumerator kPINT_PatternMatchAlways
Always Contributes to product term match

enumerator kPINT_PatternMatchStickyRise
Sticky Rising edge

enumerator kPINT_PatternMatchStickyFall
Sticky Falling edge

enumerator kPINT_PatternMatchStickyBothEdges
Sticky Rising or Falling edge

enumerator kPINT_PatternMatchHigh
High level

enumerator kPINT_PatternMatchLow
Low level

enumerator kPINT_PatternMatchNever
Never contributes to product term match

enumerator kPINT_PatternMatchBothEdges
Either rising or falling edge

typedef enum __pint_pin_enable pint_pin_enable_t
PINT Pin Interrupt enable type.

typedef enum __pint_int pint_pin_int_t
PINT Pin Interrupt type.

typedef enum __pint_pmatch_input_src pint_pmatch_input_src_t
PINT Pattern Match bit slice input source type.

typedef enum __pint_pmatch_bslice pint_pmatch_bslice_t
PINT Pattern Match bit slice type.

typedef enum __pint_pmatch_bslice_cfg pint_pmatch_bslice_cfg_t
PINT Pattern Match configuration type.

typedef struct __pint_status pint_status_t
PINT event status.

typedef void (*pint_cb_t)(pint_pin_int_t pintr, pint_status_t *status)
PINT Callback function.

```
typedef struct pint_pmatch_cfg pint_pmatch_cfg_t
```

```
void PINT_Init(PINT_Type *base)
```

Initialize PINT peripheral.

This function initializes the PINT peripheral and enables the clock.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

```
void PINT_SetCallback(PINT_Type *base, pint_cb_t callback)
```

Set PINT callback.

This function set the callback for PINT interrupt handler.

Parameters

- base – Base address of the PINT peripheral.
- callback – Callback.

Return values

None. –

```
void PINT_PinInterruptConfig(PINT_Type *base, pint_pin_int_t intr, pint_pin_enable_t enable)
```

Configure PINT peripheral pin interrupt.

This function configures a given pin interrupt.

Parameters

- base – Base address of the PINT peripheral.
- intr – Pin interrupt.
- enable – Selects detection logic.

Return values

None. –

```
void PINT_PinInterruptGetConfig(PINT_Type *base, pint_pin_int_t pintr, pint_pin_enable_t *enable)
```

Get PINT peripheral pin interrupt configuration.

This function returns the configuration of a given pin interrupt.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.
- enable – Pointer to store the detection logic.

Return values

None. –

```
void PINT_PinInterruptClrStatus(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt status only when the pin was triggered by edge-sensitive.

This function clears the selected pin interrupt status.

Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetStatus(PINT_Type *base, pintr)
```

Get Selected pin interrupt status.

This function returns the selected pin interrupt status.

Parameters

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

Return values

status – = 0 No pin interrupt request. = 1 Selected Pin interrupt request active.

```
void PINT_PinInterruptClrStatusAll(PINT_Type *base)
```

Clear all pin interrupts status only when pins were triggered by edge-sensitive.

This function clears the status of all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetStatusAll(PINT_Type *base)
```

Get all pin interrupts status.

This function returns the status of all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

status – Each bit position indicates the status of corresponding pin interrupt.
= 0 No pin interrupt request. = 1 Pin interrupt request active.

```
static inline void PINT_PinInterruptClrFallFlag(PINT_Type *base, pintr)
```

Clear Selected pin interrupt fall flag.

This function clears the selected pin interrupt fall flag.

Parameters

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlag(PINT_Type *base, pintr)
```

Get selected pin interrupt fall flag.

This function returns the selected pin interrupt fall flag.

Parameters

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

Return values

flag – = 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrFallFlagAll(PINT_Type *base)
```

Clear all pin interrupt fall flags.

This function clears the fall flag for all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlagAll(PINT_Type *base)
```

Get all pin interrupt fall flags.

This function returns the fall flag of all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

flags – Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlag(PINT_Type *base, pintr)
```

Clear Selected pin interrupt rise flag.

This function clears the selected pin interrupt rise flag.

Parameters

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlag(PINT_Type *base, pintr)
```

Get selected pin interrupt rise flag.

This function returns the selected pin interrupt rise flag.

Parameters

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

Return values

flag – = 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlagAll(PINT_Type *base)
```

Clear all pin interrupt rise flags.

This function clears the rise flag for all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlagAll(PINT_Type *base)
```

Get all pin interrupt rise flags.

This function returns the rise flag of all pin interrupts.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

flags – Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
void PINT_PatternMatchConfig(PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)
```

Configure PINT pattern match.

This function configures a given pattern match bit slice.

Parameters

- *base* – Base address of the PINT peripheral.
- *bslice* – Pattern match bit slice number.
- *cfg* – Pointer to bit slice configuration.

Return values

None. –

```
void PINT_PatternMatchGetConfig(PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)
```

Get PINT pattern match configuration.

This function returns the configuration of a given pattern match bit slice.

Parameters

- *base* – Base address of the PINT peripheral.
- *bslice* – Pattern match bit slice number.
- *cfg* – Pointer to bit slice configuration.

Return values

None. –

```
static inline uint32_t PINT_PatternMatchGetStatus(PINT_Type *base, pint_pmatch_bslice_t bslice)
```

Get pattern match bit slice status.

This function returns the status of selected bit slice.

Parameters

- *base* – Base address of the PINT peripheral.
- *bslice* – Pattern match bit slice number.

Return values

status – = 0 Match has not been detected. = 1 Match has been detected.

```
static inline uint32_t PINT_PatternMatchGetStatusAll(PINT_Type *base)
```

Get status of all pattern match bit slices.

This function returns the status of all bit slices.

Parameters

- *base* – Base address of the PINT peripheral.

Return values

status – Each bit position indicates the match status of corresponding bit slice. = 0 Match has not been detected. = 1 Match has been detected.

uint32_t PINT_PatternMatchResetDetectLogic(PINT_Type *base)

Reset pattern match detection logic.

This function resets the pattern match detection logic if any of the product term is matching.

Parameters

- base – Base address of the PINT peripheral.

Return values

pmstatus – Each bit position indicates the match status of corresponding bit slice. = 0 Match was detected. = 1 Match was not detected.

static inline void PINT_PatternMatchEnable(PINT_Type *base)

Enable pattern match function.

This function enables the pattern match function.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchDisable(PINT_Type *base)

Disable pattern match function.

This function disables the pattern match function.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchEnableRXEV(PINT_Type *base)

Enable RXEV output.

This function enables the pattern match RXEV output.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

static inline void PINT_PatternMatchDisableRXEV(PINT_Type *base)

Disable RXEV output.

This function disables the pattern match RXEV output.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_EnableCallback(PINT_Type *base)

Enable callback.

This function enables the interrupt for the selected PINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_DisableCallback(PINT_Type *base)

Disable callback.

This function disables the interrupt for the selected PINT peripheral. Although the pins are still being monitored but the callback function is not called.

Parameters

- base – Base address of the peripheral.

Return values

None. –

void PINT_Deinit(PINT_Type *base)

Deinitialize PINT peripheral.

This function disables the PINT clock.

Parameters

- base – Base address of the PINT peripheral.

Return values

None. –

void PINT_EnableCallbackByIndex(PINT_Type *base, *pint_pin_int_t* pintIdx)

enable callback by pin index.

This function enables callback by pin index instead of enabling all pins.

Parameters

- base – Base address of the peripheral.
- pintIdx – pin index.

Return values

None. –

void PINT_DisableCallbackByIndex(PINT_Type *base, *pint_pin_int_t* pintIdx)

disable callback by pin index.

This function disables callback by pin index instead of disabling all pins.

Parameters

- base – Base address of the peripheral.
- pintIdx – pin index.

Return values

None. –

PINT_USE_LEGACY_CALLBACK

PININT_BITSLICE_SRC_START

PININT_BITSLICE_SRC_MASK

PININT_BITSLICE_CFG_START

PININT_BITSLICE_CFG_MASK

PININT_BITSLICE_ENDP_MASK

PINT_PIN_INT_LEVEL

PINT_PIN_INT_EDGE

PINT_PIN_INT_FALL_OR_HIGH_LEVEL

PINT_PIN_INT_RISE

PINT_PIN_RISE_EDGE

PINT_PIN_FALL_EDGE

PINT_PIN_BOTH_EDGE

PINT_PIN_LOW_LEVEL

PINT_PIN_HIGH_LEVEL

struct __pint_status

#include <fsl_pint.h> PINT event status.

struct __pint_pmatch_cfg

#include <fsl_pint.h>

2.45 Power Driver

enum __power_wakeup_edge

Pin edge for wakeup.

Values:

enumerator kPOWER_WakeupEdgeLow

Wakeup on pin low level.

enumerator kPOWER_WakeupEdgeHigh

Wakeup on pin high level.

enum __power_wakeup_pin

Wakeup pin.

Values:

enumerator kPOWER_WakeupPin0

Wakeup0 pin.

enumerator kPOWER_WakeupPin1

Wakeup1 pin.

enum __power_reset_cause

Reset cause.

Values:

enumerator kPOWER_ResetCauseSysResetReq

CM33 system soft reset request.

enumerator kPOWER_ResetCauseLockup

CM33 locked up.

enumerator kPOWER_ResetCauseWdt

Watchdog timer.

enumerator kPOWER_ResetCauseApResetReq

Debug mailbox reset.

enumerator kPOWER__ResetCauseCodeWdt

Code watchdog timer.

enumerator kPOWER__ResetCauseItrc

ITRC_CHIP reset.

enumerator kPOWER__ResetCauseResetB

sw_resetb_scantest reset.

enumerator kPOWER__ResetCauseAll

All reset causes. Used in POWER_ClearResetCause().

enum __power__reset__source

Reset source.

Values:

enumerator kPOWER__ResetSourceSysResetReq

CM33 system soft reset request.

enumerator kPOWER__ResetSourceLockup

CM33 locked up.

enumerator kPOWER__ResetSourceWdt

Watchdog timer.

enumerator kPOWER__ResetSourceApResetReq

Debug mailbox reset.

enumerator kPOWER__ResetSourceCodeWdt

Code watchdog timer.

enumerator kPOWER__ResetSourceItrc

ITRC_CHIP reset.

enumerator kPOWER__ResetSourceAll

All reset sources.

enum __pm2__mem__pu__bits

PM2 mem power up bits definition.

Values:

enumerator kPOWER__Pm2MemPuEnet

enumerator kPOWER__Pm2MemPuSdio

enumerator kPOWER__Pm2MemPuOtp

enumerator kPOWER__Pm2MemPuRom

enumerator kPOWER__Pm2MemPuFlexspi

enumerator kPOWER__Pm2MemPuPq

enumerator kPOWER__Pm2MemPuPkc

enumerator kPOWER__Pm2MemPuEls

enumerator kPOWER__Pm2MemPuAon1

enumerator kPOWER__Pm2MemPuAon0

enumerator kPOWER__Pm2MemPuSram18

enumerator kPOWER_Pm2MemPuSram17
enumerator kPOWER_Pm2MemPuSram16
enumerator kPOWER_Pm2MemPuSram15
enumerator kPOWER_Pm2MemPuSram14
enumerator kPOWER_Pm2MemPuSram13
enumerator kPOWER_Pm2MemPuSram12
enumerator kPOWER_Pm2MemPuSram11
enumerator kPOWER_Pm2MemPuSram10
enumerator kPOWER_Pm2MemPuSram9
enumerator kPOWER_Pm2MemPuSram8
enumerator kPOWER_Pm2MemPuSram7
enumerator kPOWER_Pm2MemPuSram6
enumerator kPOWER_Pm2MemPuSram5
enumerator kPOWER_Pm2MemPuSram4
enumerator kPOWER_Pm2MemPuSram3
enumerator kPOWER_Pm2MemPuSram2
enumerator kPOWER_Pm2MemPuSram1
enumerator kPOWER_Pm2MemPuSram0
enumerator kPOWER_Pm2MemPuAll

enum _pm2_ana_pu_bits

PM2 ana power up bits definition.

Values:

enumerator kPOWER_Pm2AnaPuT3
enumerator kPOWER_Pm2AnaPuTcpuTop
enumerator kPOWER_Pm2AnaPuTddrTop
enumerator kPOWER_Pm2AnaPuAnaTop
enumerator kPOWER_Pm2AnaPuGau
enumerator kPOWER_Pm2AnaPuUsb
enumerator kPOWER_Pm2AnaPuAvpll
enumerator kPOWER_Pm2AnaPuAll

enum _clk_gate_bits

clock gate bits definition

Values:

enumerator kPOWER_ClkGateTddrMciEnet

enumerator kPOWER_ClkGateAll

enum _clk_pm3_buck_bits

PM3 buck control bits definition.

Values:

enumerator kPOWER_Pm3Buck18

1: Use normal buck18 level in PM3. 0: Use sleep buck18 level in PM3

enumerator kPOWER_Pm3Buck11

1: Use normal buck11 level in PM3. 0: Use sleep buck11 level in PM3

enumerator kPOWER_Pm3BuckAll

enum _capt_slow_pulse_width

Capture slow pulse width.

Values:

enumerator kPOWER_CaptSlowPulseWidth1

enumerator kPOWER_CaptSlowPulseWidth2

enumerator kPOWER_CaptSlowPulseWidth3

enumerator kPOWER_CaptSlowPulseWidth4

enumerator kPOWER_CaptSlowPulseWidth5

enumerator kPOWER_CaptSlowPulseWidth6

enumerator kPOWER_CaptSlowPulseWidth7

enum _capt_slow_pulse_edge

Capture slow pulse edge.

Values:

enumerator kPOWER_CaptSlowPulseEdgeRising

enumerator kPOWER_CaptSlowPulseEdgeFalling

enumerator kPOWER_CaptSlowPulseEdgeAny

typedef enum _power_wakeup_edge power_wakeup_edge_t

Pin edge for wakeup.

typedef enum _power_wakeup_pin power_wakeup_pin_t

Wakeup pin.

typedef enum _power_reset_cause power_reset_cause_t

Reset cause.

typedef enum _power_reset_source power_reset_source_t

Reset source.

typedef enum _capt_slow_pulse_width capt_slow_pulse_width_t

Capture slow pulse width.

typedef enum _capt_slow_pulse_edge capt_slow_pulse_edge_t

Capture slow pulse edge.

```
typedef void (*capt_pulse_timer_callback_t)(void *param)
```

Capture timer callback function.

Param param

: User parameter for callback.

```
typedef void (*power_switch_callback_t)(uint32_t mode, void *param)
```

Power mode switch callback function.

Param mode

: Power mode to switch.

Param param

: User parameter for callback.

```
typedef struct _power_init_config power_init_config_t
```

Init configuration.

```
typedef struct _power_sleep_config power_sleep_config_t
```

Sleep configuration.

```
typedef struct _power_gdet_data power_gdet_data_t
```

Glitch detector configuration.

```
typedef bool (*power_load_gdet_cfg)(power_gdet_data_t *data)
```

Glitch detector configuration load function.

```
__STATIC_INLINE void POWER_EnableResetSource (uint32_t source)
```

Enable system reset source.

Parameters

- source – : A bitmask of of power_reset_source_t

```
__STATIC_INLINE void POWER_DisableResetSource (uint32_t source)
```

Disable system reset source.

Parameters

- source – : A bitmask of of power_reset_source_t

```
__STATIC_INLINE uint32_t POWER_GetResetCause (void)
```

Get last reset cause.

Returns

Or'ed cause of power_reset_cause_t

```
__STATIC_INLINE void POWER_ClearResetCause (uint32_t cause)
```

Clear last reset cause.

Parameters

- cause – : A bitmask of of power_reset_cause_t

```
__STATIC_INLINE void POWER_ConfigWakeupPin (power_wakeup_pin_t pin,  
power_wakeup_edge_t edge)
```

Configure pin edge for wakeup.

Parameters

- pin – : Wakeup pin
- edge – : Pin level for wakeup

bool POWER_GetWakeupStatus(IRQn_Type irq)

Check if IRQ is the wakeup source.

Parameters

- irq – : IRQ number

Returns

true if IRQ is the wakeup source, false otherwise.

void POWER_ClearWakeupStatus(IRQn_Type irq)

Clear wakeup status.

Parameters

- irq – : IRQ number

void POWER_EnableWakeup(IRQn_Type irq)

Enable the Wakeup interrupt.

Parameters

- irq – : IRQ number

void POWER_DisableWakeup(IRQn_Type irq)

Disable the Wakeup interrupts.

Parameters

- irq – : IRQ number

AT_QUICKACCESS_SECTION_CODE (void POWER_SetSleepMode(uint32_t mode))

Set power mode on idle.

Parameters

- mode – : 0 ~ 4 stands for PM0 ~ PM4.

__STATIC_INLINE uint32_t POWER_GetWakenMode (void)

Get power mode waken up from.

Returns

Power mode.

void POWER_GetCurrentSleepConfig(*power_sleep_config_t* *config)

Get current sleep configuration.

Parameters

- config – : Pointer to config structure to save current config.

void POWER_InitPowerConfig(const *power_init_config_t* *config)

Initialize power configuration.

Parameters

- config – : Pointer to init config structure.

void POWER_ConfigCauInSleep(bool pdCau)

Configure CAU_SOC_SLP_REF_GEN_CLK on/off status in SoC sleep mode.

Parameters

- pdCau – : true for clock off; false for clock on.

void POWER_SetPowerSwitchCallback(*power_switch_callback_t* pre, void *preParam,
power_switch_callback_t post, void *postParam)

Set power mode switch callback. The callbacks are called with interrupt disabled.

Parameters

- pre – : Function called before power mode switch
- preParam – : User parameter for pre callback
- post – : Function called after power mode switch
- postParam – : User parameter for post callback

AT_QUICKACCESS_SECTION_CODE (bool POWER_EnterPowerMode(uint32_t mode, const power_sleep_config_t *config))

Switch system into certain power mode.

Parameters

- mode – : 0 ~ 4 stands for PM0 ~ PM4.
- config – : Sleep configuration on PM2-PM4.

Returns

True for success, else failure.

void POWER_PowerOnWlan(void)

Power on WLAN.

void POWER_PowerOffWlan(void)

Power off WLAN.

__STATIC_INLINE void PMU_EnableWlanWakeup (uint8_t wlWakeup)

Enable MCI wakeup WLAN.

Parameters

- wlWakeup – : 8 bits wakeup mask

__STATIC_INLINE void PMU_DisableWlanWakeup (uint8_t wlWakeup)

Disable MCI wakeup WLAN.

Parameters

- wlWakeup – : 8 bits wakeup mask

void POWER_PowerOnBle(void)

Power on BLE.

void POWER_PowerOffBle(void)

Power off BLE.

__STATIC_INLINE void PMU_EnableBleWakeup (uint8_t bleWakeup)

Enable MCI wakeup BLE.

Parameters

- bleWakeup – : 8 bits wakeup mask

__STATIC_INLINE void PMU_DisableBleWakeup (uint8_t bleWakeup)

Disable MCI wakeup BLE.

Parameters

- bleWakeup – : 8 bits wakeup mask

void POWER_PowerOnGau(void)

Power on GAU.

void POWER_PowerOffGau(void)

Power off GAU.

```
void POWER_EnableCaptSlowPulseTimer(capt_slow_pulse_width_t width,  
                                     capt_slow_pulse_edge_t edge, uint32_t timeout,  
                                     capt_pulse_timer_callback_t cb, void *param)
```

Enable capture slow pulse timer with 32768Hz clock source.

Parameters

- width – : input capture filter width in cycles
- edge – : trigger condition of counter
- timeout – : timer expire counter which will trigger callback
- callback – : callback function on timer expire
- param – : callback parameter

```
void POWER_EnableCaptFastPulseTimer(uint32_t timeout, capt_pulse_timer_callback_t cb, void  
                                     *param)
```

Enable capture fast pulse timer with 3.84/4MHz clock source.

Parameters

- timeout – : timer expire counter which will trigger callback
- callback – : callback function on timer expire
- param – : callback parameter

```
void POWER_DisableCaptPulseTimer(void)
```

Disable capture pulse timer.

```
void POWER_InitVoltage(uint32_t dro, uint32_t pack)
```

Configure power rail voltage and LVD/HVD threshold.

Parameters

- dro – : trim value from fuse.
- pack – : Device package type: 0 - QFN, 1 - CSP, 2 - BGA

```
void Power_InitLoadGdetCfg(power_load_gdet_cfg loadFunc, const power_gdet_data_t *data,  
                           uint32_t pack)
```

Initialize glitch detector configuration.

Parameters

- loadFunc – : function pointer to the GDET load configuration.
- data – : GDET config data loaded from fuse.
- pack – : Device package type: 0 - QFN, 1 - CSP, 2 - BGA

```
AT_QUICKACCESS_SECTION_CODE (void POWER_DisableGDetVSensors(void))
```

Disable GDET and VSensors.

```
AT_QUICKACCESS_SECTION_CODE (bool POWER_EnableGDetVSensors(void))
```

Enable GDET and VSensors.

Returns

True for success, else failure.

```
uint32_t POWER_TrimSvc(uint32_t gdetTrim, uint32_t pack)
```

Apply SVC GDC equation and get the SVC trim configuration.

Parameters

- gdetTrim – : GDET trim value from fuse.
- pack – : Device package type: 0 - QFN, 1 - CSP, 2 - BGA

FSL_POWER_DRIVER_VERSION

POWER driver version 2.5.3.

bool iBuck

true: VCORE and AVDD18 supplied from iBuck; false: supplied from external DCDC.

bool gateCauRefClk

true: CAU_SOC_SLP_REF_GEN_CLK gated; false: CAU_SOC_SLP_REF_GEN_CLK on.

uint32_t pm2MemPuCfg

Modules to keep powered on in PM2 mode. Logical OR of the enums in `_pm2_mem_pu_bits`.

uint32_t pm2AnaPuCfg

Ana to keep powered on in PM2 mode. Logical OR of the enums in `_pm2_ana_pu_bits`.

uint32_t clkGate

Source clock gate control. Logical OR of the enums in `_clk_gate_bits`.

uint32_t memPdCfg

PMU MEM_CFG: Power Down memory configuration. Bit0-5 for PM3, bit8 for PM4. bit0: ram0-5 384KB bit1: ram6 64KB bit2: ram7 64KB bit3: ram8-9 128KB bit4: ram10-13 256KB bit5: ram14-18 320KB. bit8: aon mem higher 8KB

uint32_t pm3BuckCfg

PMIP BUCK control in PM3 mode. Logical OR of the enums in `_clk_pm3_buck_bits`.

uint32_t CFG[6]

uint32_t TRIM0

struct __power_init_config

#include <fsl_power.h> Init configuration.

struct __power_sleep_config

#include <fsl_power.h> Sleep configuration.

struct __power_gdet_data

#include <fsl_power.h> Glitch detector configuration.

2.46 POWERQUAD: PowerQuad hardware accelerator

void PQ_GetDefaultConfig(*pq_config_t* *config)

Get default configuration.

This function initializes the POWERQUAD configuration structure to a default value. FORMAT register field definitions Bits[15:8] scaler (for scaled 'q31' formats) Bits[5:4] external format. 00b=q15, 01b=q31, 10b=float Bits[1:0] internal format. 00b=q15, 01b=q31, 10b=float POWERQUAD->INAFORMAT = (config->inputAPrescale « 8U) | (config->inputAFormat « 4U) | config->machineFormat

For all Powerquad operations internal format must be float (with the only exception being the FFT related functions, ie FFT/IFFT/DCT/IDCT which must be set to q31). The default values are: config->inputAFormat = kPQ_Float; config->inputAPrescale = 0; config->inputBFormat = kPQ_Float; config->inputBPrescale = 0; config->outputFormat = kPQ_Float; config->outputPrescale = 0; config->tmpFormat = kPQ_Float; config->tmpPrescale = 0; config->machineFormat = kPQ_Float; config->tmpBase = 0xE0000000;

Parameters

- config – Pointer to “pq_config_t” structure.

void PQ_SetConfig(POWERQUAD_Type *base, const *pq_config_t* *config)

Set configuration with format/prescale.

Parameters

- base – POWERQUAD peripheral base address
- config – Pointer to “pq_config_t” structure.

static inline void PQ_SetCoproprocessorScaler(POWERQUAD_Type *base, const *pq_prescale_t* *prescale)

set coprocessor scaler for coprocessor instructions, this function is used to set output saturation and scaling for input/output.

Parameters

- base – POWERQUAD peripheral base address
- prescale – Pointer to “pq_prescale_t” structure.

void PQ_Init(POWERQUAD_Type *base)

Initializes the POWERQUAD module.

Parameters

- base – POWERQUAD peripheral base address.

void PQ_Deinit(POWERQUAD_Type *base)

De-initializes the POWERQUAD module.

Parameters

- base – POWERQUAD peripheral base address.

void PQ_SetFormat(POWERQUAD_Type *base, *pq_computationengine_t* engine, *pq_format_t* format)

Set format for non-coprocessor instructions.

Parameters

- base – POWERQUAD peripheral base address
- engine – Computation engine
- format – Data format

static inline void PQ_WaitDone(POWERQUAD_Type *base)

Wait for the completion.

Parameters

- base – POWERQUAD peripheral base address

static inline void PQ_LnF32(float *pSrc, float *pDst)

Processing function for the floating-point natural log.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (0 +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_InvF32(float *pSrc, float *pDst)

Processing function for the floating-point reciprocal.

Parameters

- *pSrc – points to the block of input data. The range of the input value is non-zero.

- *pDst – points to the block of output data

static inline void PQ_SqrtF32(float *pSrc, float *pDst)

Processing function for the floating-point square-root.

Parameters

- *pSrc – points to the block of input data. The range of the input value is [0 +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_InvSqrtF32(float *pSrc, float *pDst)

Processing function for the floating-point inverse square-root.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (0 +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_EtoxF32(float *pSrc, float *pDst)

Processing function for the floating-point natural exponent.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_EtonxF32(float *pSrc, float *pDst)

Processing function for the floating-point natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data. The range of the input value is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_SinF32(float *pSrc, float *pDst)

Processing function for the floating-point sine.

Parameters

- *pSrc – points to the block of input data. The input value is in radians, the range is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_CosF32(float *pSrc, float *pDst)

Processing function for the floating-point cosine.

Parameters

- *pSrc – points to the block of input data. The input value is in radians, the range is (-INFINITY +INFINITY).
- *pDst – points to the block of output data

static inline void PQ_BiquadF32(float *pSrc, float *pDst)

Processing function for the floating-point biquad.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data

static inline void PQ_DivF32(float *x1, float *x2, float *pDst)

Processing function for the floating-point division.

Get $x1 / x2$.

Parameters

- $x1$ – $x1$
- $x2$ – $x2$
- $*pDst$ – points to the block of output data

static inline void PQ_Biquad1F32(float *pSrc, float *pDst)

Processing function for the floating-point biquad.

Parameters

- $*pSrc$ – points to the block of input data
- $*pDst$ – points to the block of output data

static inline int32_t PQ_LnFixed(int32_t val)

Processing function for the fixed natural log.

Parameters

- val – value to be calculated. The range of the input value is $(0 + INFINITY)$.

Returns

returns $\ln(val)$.

static inline int32_t PQ_InvFixed(int32_t val)

Processing function for the fixed reciprocal.

Parameters

- val – value to be calculated. The range of the input value is non-zero.

Returns

returns $\text{inv}(val)$.

static inline uint32_t PQ_SqrtFixed(uint32_t val)

Processing function for the fixed square-root.

Parameters

- val – value to be calculated. The range of the input value is $[0 + INFINITY)$.

Returns

returns $\text{sqrt}(val)$.

static inline int32_t PQ_InvSqrtFixed(int32_t val)

Processing function for the fixed inverse square-root.

Parameters

- val – value to be calculated. The range of the input value is $(0 + INFINITY)$.

Returns

returns $1/\text{sqrt}(val)$.

static inline int32_t PQ_EtoxFixed(int32_t val)

Processing function for the Fixed natural exponent.

Parameters

- val – value to be calculated. The range of the input value is $(-INFINITY + INFINITY)$.

Returns

returns etox^{val} .

static inline int32_t PQ_EtonxFixed(int32_t val)

Processing function for the fixed natural exponent with negative parameter.

Parameters

- val – value to be calculated. The range of the input value is (-INFINITY +INFINITY).

Returns

returns etox^{val} .

static inline int32_t PQ_SinQ31(int32_t val)

Processing function for the fixed sine.

Parameters

- val – value to be calculated. The input value is [-1, 1] in Q31 format, which means [-pi, pi].

Returns

returns $\sin(\text{val})$.

static inline int16_t PQ_SinQ15(int16_t val)

Processing function for the fixed sine.

Parameters

- val – value to be calculated. The input value is [-1, 1] in Q15 format, which means [-pi, pi].

Returns

returns $\sin(\text{val})$.

static inline int32_t PQ_CosQ31(int32_t val)

Processing function for the fixed cosine.

Parameters

- val – value to be calculated. The input value is [-1, 1] in Q31 format, which means [-pi, pi].

Returns

returns $\cos(\text{val})$.

static inline int16_t PQ_CosQ15(int16_t val)

Processing function for the fixed sine.

Parameters

- val – value to be calculated. The input value is [-1, 1] in Q15 format, which means [-pi, pi].

Returns

returns $\sin(\text{val})$.

static inline int32_t PQ_BiquadFixed(int32_t val)

Processing function for the fixed biquad.

Parameters

- val – value to be calculated

Returns

returns $\text{biquad}(\text{val})$.

void PQ_VectorLnF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised natural log.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised reciprocal.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSqrtF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvSqrtF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised inverse square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtoxF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised natural exponent.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtonxF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSinF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised sine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorCosF32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised cosine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorLnFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised natural log.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised reciprocal.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSqrtFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvSqrtFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised inverse square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtoxFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised natural exponent.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtonxFixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSinQ15(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the Q15 vectorised sine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorCosQ15(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the Q15 vectorised cosine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSinQ31(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised sine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorCosQ31(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the Q31 vectorised cosine.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorLnFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised natural log.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised reciprocal.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorSqrtFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorInvSqrtFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised inverse square-root.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtoxFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised natural exponent.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorEtonxFixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised natural exponent with negative parameter.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block of input data.

void PQ_VectorBiquadDf2F32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data.

void PQ_VectorBiquadDf2Fixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised biquad direct form II.

Parameters

- *pSrc – points to the block of input data
- *pDst – points to the block of output data
- length – the block size of input data

void PQ_VectorBiquadDf2Fixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised biquad direct form II.

Parameters

- *pSrc* – points to the block of input data
- *pDst* – points to the block of output data
- *length* – the block size of input data

void PQ_VectorBiquadCascadeDf2F32(float *pSrc, float *pDst, int32_t length)

Processing function for the floating-point vectorised biquad direct form II.

Parameters

- *pSrc* – points to the block of input data
- *pDst* – points to the block of output data
- *length* – the block size of input data

void PQ_VectorBiquadCascadeDf2Fixed32(int32_t *pSrc, int32_t *pDst, int32_t length)

Processing function for the 32-bit integer vectorised biquad direct form II.

Parameters

- *pSrc* – points to the block of input data
- *pDst* – points to the block of output data
- *length* – the block size of input data

void PQ_VectorBiquadCascadeDf2Fixed16(int16_t *pSrc, int16_t *pDst, int32_t length)

Processing function for the 16-bit integer vectorised biquad direct form II.

Parameters

- *pSrc* – points to the block of input data
- *pDst* – points to the block of output data
- *length* – the block size of input data

int32_t PQ_ArctanFixed(POWERQUAD_Type *base, int32_t x, int32_t y, *pq_cordic_iter_t* iteration)

Processing function for the fixed inverse trigonometric.

Get the inverse tangent, the behavior is like c function atan.

Note: The sum of x and y should not exceed the range of int32_t.

Note: Larger input number gets higher output accuracy, for example the arctan(0.5), the result of PQ_ArctanFixed(POWERQUAD, 100000, 200000, kPQ_Iteration_24) is more accurate than PQ_ArctanFixed(POWERQUAD, 1, 2, kPQ_Iteration_24).

Parameters

- *base* – POWERQUAD peripheral base address
- *x* – value of opposite
- *y* – value of adjacent
- *iteration* – iteration times

Returns

The return value is in the range of -2^{26} to 2^{26} , which means $-\pi/2$ to $\pi/2$.

```
int32_t PQ_ArctanhFixed(POWERQUAD_Type *base, int32_t x, int32_t y, pq_cordic_iter_t iteration)
```

Processing function for the fixed inverse trigonometric.

Note: The sum of x and y should not exceed the range of int32_t.

Note: Larger input number gets higher output accuracy, for example the $\text{arctanh}(0.5)$, the result of `PQ_ArctanhFixed(POWERQUAD, 100000, 200000, kPQ_Iteration_24)` is more accurate than `PQ_ArctanhFixed(POWERQUAD, 1, 2, kPQ_Iteration_24)`.

Parameters

- base – POWERQUAD peripheral base address
- x – value of opposite
- y – value of adjacent
- iteration – iteration times

Returns

The return value is radians, 2^{27} means π . The range is -1.118 to 1.118 radians.

```
int32_t PQ_Arctan2Fixed(POWERQUAD_Type *base, int32_t x, int32_t y, pq_cordic_iter_t iteration)
```

Processing function for the fixed inverse trigonometric.

Get the inverse tangent, it calculates the angle in radians for the quadrant. The behavior is like c function `atan2`.

Note: The sum of x and y should not exceed the range of int32_t.

Note: Larger input number gets higher output accuracy, for example the $\text{arctan}(0.5)$, the result of `PQ_Arctan2Fixed(POWERQUAD, 100000, 200000, kPQ_Iteration_24)` is more accurate than `PQ_Arctan2Fixed(POWERQUAD, 1, 2, kPQ_Iteration_24)`.

Parameters

- base – POWERQUAD peripheral base address
- x – value of opposite
- y – value of adjacent
- iteration – iteration times

Returns

The return value is in the range of -2^{27} to 2^{27} , which means $-\pi$ to π .

```
static inline int32_t PQ_Biquad1Fixed(int32_t val)
```

Processing function for the fixed biquad.

Parameters

- val – value to be calculated

Returns

returns biquad(val).

void PQ_TransformCFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)

Processing function for the complex FFT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

void PQ_TransformRFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)

Processing function for the real FFT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

void PQ_TransformIFFT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)

Processing function for the inverse complex FFT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

void PQ_TransformCDCT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)

Processing function for the complex DCT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

void PQ_TransformRDCT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)

Processing function for the real DCT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

void PQ_TransformIDCT(POWERQUAD_Type *base, uint32_t length, void *pData, void *pResult)

Processing function for the inverse complex DCT.

Parameters

- base – POWERQUAD peripheral base address
- length – number of input samples
- pData – input data
- pResult – output data.

void PQ_BiquadBackUpInternalState(POWERQUAD_Type *base, int32_t biquad_num,
pq_biquad_state_t *state)

Processing function for backup biquad context.

Parameters

- base – POWERQUAD peripheral base address
- biquad_num – biquad side
- state – point to states.

void PQ_BiquadRestoreInternalState(POWERQUAD_Type *base, int32_t biquad_num,
pq_biquad_state_t *state)

Processing function for restore biquad context.

Parameters

- base – POWERQUAD peripheral base address
- biquad_num – biquad side
- state – point to states.

void PQ_BiquadCascadeDf2Init(pq_biquad_cascade_df2_instance *S, uint8_t numStages,
pq_biquad_state_t *pState)

Initialization function for the direct form II Biquad cascade filter.

Parameters

- *S – **[inout]** points to an instance of the filter data structure.
- numStages – **[in]** number of 2nd order stages in the filter.
- *pState – **[in]** points to the state buffer.

void PQ_BiquadCascadeDf2F32(const pq_biquad_cascade_df2_instance *S, float *pSrc, float
*pDst, uint32_t blockSize)

Processing function for the floating-point direct form II Biquad cascade filter.

Parameters

- *S – **[in]** points to an instance of the filter data structure.
- *pSrc – **[in]** points to the block of input data.
- *pDst – **[out]** points to the block of output data
- blockSize – **[in]** number of samples to process.

void PQ_BiquadCascadeDf2Fixed32(const pq_biquad_cascade_df2_instance *S, int32_t *pSrc,
int32_t *pDst, uint32_t blockSize)

Processing function for the Q31 direct form II Biquad cascade filter.

Parameters

- *S – **[in]** points to an instance of the filter data structure.

- *pSrc – **[in]** points to the block of input data.
- *pDst – **[out]** points to the block of output data
- blockSize – **[in]** number of samples to process.

```
void PQ_BiquadCascadeDf2Fixed16(const pq_biquad_cascade_df2_instance *S, int16_t *pSrc,
                                int16_t *pDst, uint32_t blockSize)
```

Processing function for the Q15 direct form II Biquad cascade filter.

Parameters

- *S – **[in]** points to an instance of the filter data structure.
- *pSrc – **[in]** points to the block of input data.
- *pDst – **[out]** points to the block of output data
- blockSize – **[in]** number of samples to process.

```
void PQ_FIR(POWERQUAD_Type *base, const void *pAData, int32_t ALength, const void
            *pBData, int32_t BLength, void *pResult, uint32_t opType)
```

Processing function for the FIR.

Parameters

- base – POWERQUAD peripheral base address
- pAData – the first input sequence
- ALength – number of the first input sequence
- pBData – the second input sequence
- BLength – number of the second input sequence
- pResult – array for the output data
- opType – operation type, could be PQ_FIR_FIR, PQ_FIR_CONVOLUTION, PQ_FIR_CORRELATION.

```
void PQ_FIRIncrement(POWERQUAD_Type *base, int32_t ALength, int32_t BLength, int32_t
                    xOffset)
```

Processing function for the incremental FIR. This function can be used after pq_fir() for incremental FIR operation when new x data are available.

Parameters

- base – POWERQUAD peripheral base address
- ALength – number of input samples
- BLength – number of taps
- xOffset – offset for number of input samples

```
void PQ_MatrixAddition(POWERQUAD_Type *base, uint32_t length, void *pAData, void
                      *pBData, void *pResult)
```

Processing function for the matrix addition.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pAData – input matrix A

- pBData – input matrix B
- pResult – array for the output data.

```
void PQ_MatrixSubtraction(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the matrix subtraction.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pAData – input matrix A
- pBData – input matrix B
- pResult – array for the output data.

```
void PQ_MatrixMultiplication(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the matrix multiplication.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pAData – input matrix A
- pBData – input matrix B
- pResult – array for the output data.

```
void PQ_MatrixProduct(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the matrix product.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pAData – input matrix A
- pBData – input matrix B
- pResult – array for the output data.

```
void PQ_VectorDotProduct(POWERQUAD_Type *base, uint32_t length, void *pAData, void *pBData, void *pResult)
```

Processing function for the vector dot product.

Parameters

- base – POWERQUAD peripheral base address
- length – length of vector

- pAData – input vector A
- pBData – input vector B
- pResult – array for the output data.

```
void PQ_MatrixInversion(POWERQUAD_Type *base, uint32_t length, void *pData, void
                        *pTmpData, void *pResult)
```

Processing function for the matrix inverse.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pData – input matrix
- pTmpData – input temporary matrix, pTmpData length not less than pData length and 1024 words is sufficient for the largest supported matrix.
- pResult – array for the output data, round down for fixed point.

```
void PQ_MatrixTranspose(POWERQUAD_Type *base, uint32_t length, void *pData, void
                        *pResult)
```

Processing function for the matrix transpose.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- pData – input matrix
- pResult – array for the output data.

```
void PQ_MatrixScale(POWERQUAD_Type *base, uint32_t length, float misc, const void *pData,
                    void *pResult)
```

Processing function for the matrix scale.

Parameters

- base – POWERQUAD peripheral base address
- length – rows and cols for matrix. LENGTH register configuration: LENGTH[23:16] = M2 cols LENGTH[15:8] = M1 cols LENGTH[7:0] = M1 rows This could be constructed using macro POWERQUAD_MAKE_MATRIX_LEN.
- misc – scaling parameters
- pData – input matrix
- pResult – array for the output data.

```
FSL_POWERQUAD_DRIVER_VERSION
Version.
```

```
enum pq_computationengine_t
    powerquad computation engine
```

Values:

enumerator kPQ_CP_PQ

Math engine.

enumerator kPQ_CP_MTX

Matrix engine.

enumerator kPQ_CP_FFT

FFT engine.

enumerator kPQ_CP_FIR

FIR engine.

enumerator kPQ_CP_CORDIC

CORDIC engine.

enum pq_format_t

powerquad data structure format type

Values:

enumerator kPQ_16Bit

Int16 Fixed point.

enumerator kPQ_32Bit

Int32 Fixed point.

enumerator kPQ_Float

Float point.

enum pq_cordic_iter_t

CORDIC iteration.

Values:

enumerator kPQ_Iteration_8

Iterate 8 times.

enumerator kPQ_Iteration_16

Iterate 16 times.

enumerator kPQ_Iteration_24

Iterate 24 times.

typedef struct *pq_biquad_param* pq_biquad_param_t

Struct to save biquad parameters.

typedef struct *pq_biquad_state* pq_biquad_state_t

Struct to save biquad state.

typedef union *pq_float* pq_float_t

Conversion between integer and float type.

PQ_VectorBiquadDf2F32

PQ_VectorBiquadDf2Fixed32

PQ_VectorBiquadDf2Fixed16

PQ_VectorBiquadCascadeDf2F32

PQ_VectorBiquadCascadeDf2Fixed32

PQ_VectorBiquadCascadeDf2Fixed16

PQ_Vector8BiquadDf2CascadeF32
PQ_Vector8BiquadDf2CascadeFixed32
PQ_Vector8BiquadDf2CascadeFixed16
PQ_FLOAT32
PQ_FIXEDPT
CP_PQ
CP_MTX
CP_FFT
CP_FIR
CP_CORDIC
PQ_TRANS
PQ_TRIG
PQ_BIQUAD
PQ_TRANS_FIXED
PQ_TRIG_FIXED
PQ_BIQUAD_FIXED
PQ_INV
PQ_LN
PQ_SQRT
PQ_INVSQRT
PQ_ETOX
PQ_ETONX
PQ_DIV
PQ_SIN
PQ_COS
PQ_BIQ0_CALC
PQ_BIQ1_CALC
PQ_COMP0_ONLY
PQ_COMP1_ONLY
CORDIC_ITER(x)
CORDIC_MIU(x)
CORDIC_T(x)
CORDIC_ARCTAN

CORDIC_ARCTANH
INST_BUSY
PQ_ERRSTAT_OVERFLOW
PQ_ERRSTAT_NAN
PQ_ERRSTAT_FIXEDOVERFLOW
PQ_ERRSTAT_UNDERFLOW
PQ_TRANS_CFFT
PQ_TRANS_IFFT
PQ_TRANS_CDCT
PQ_TRANS_IDCT
PQ_TRANS_RFFT
PQ_TRANS_RDCT
PQ_MTX_SCALE
PQ_MTX_MULT
PQ_MTX_ADD
PQ_MTX_INV
PQ_MTX_PROD
PQ_MTX_SUB
PQ_VEC_DOTP
PQ_MTX_TRAN
PQ_FIR_FIR
PQ_FIR_CONVOLUTION
PQ_FIR_CORRELATION
PQ_FIR_INCREMENTAL
_pq_ln0(x)
_pq_inv0(x)
_pq_sqrt0(x)
_pq_invsqrt0(x)
_pq_etox0(x)
_pq_etonx0(x)
_pq_sin0(x)
_pq_cos0(x)
_pq_biquad0(x)

`_pq_ln_fx0(x)`
`_pq_inv_fx0(x)`
`_pq_sqrt_fx0(x)`
`_pq_invsqrt_fx0(x)`
`_pq_etox_fx0(x)`
`_pq_etonx_fx0(x)`
`_pq_sin_fx0(x)`
`_pq_cos_fx0(x)`
`_pq_biquad0_fx(x)`
`_pq_div0(x)`
`_pq_div1(x)`
`_pq_ln1(x)`
`_pq_inv1(x)`
`_pq_sqrt1(x)`
`_pq_invsqrt1(x)`
`_pq_etox1(x)`
`_pq_etonx1(x)`
`_pq_sin1(x)`
`_pq_cos1(x)`
`_pq_biquad1(x)`
`_pq_ln_fx1(x)`
`_pq_inv_fx1(x)`
`_pq_sqrt_fx1(x)`
`_pq_invsqrt_fx1(x)`
`_pq_etox_fx1(x)`
`_pq_etonx_fx1(x)`
`_pq_sin_fx1(x)`
`_pq_cos_fx1(x)`
`_pq_biquad1_fx(x)`
`_pq_readMult0()`
`_pq_readAdd0()`
`_pq_readMult1()`
`_pq_readAdd1()`

`_pq_readMult0_fx()`

`_pq_readAdd0_fx()`

`_pq_readMult1_fx()`

`_pq_readAdd1_fx()`

`PQ_LN_INF`

Parameter used for vector $\ln(x)$

`PQ_INV_INF`

Parameter used for vector $1/x$

`PQ_SQRT_INF`

Parameter used for vector $\sqrt{x}$

`PQ_ISQRT_INF`

Parameter used for vector $1/\sqrt{x}$

`PQ_ETOX_INF`

Parameter used for vector e^x

`PQ_ETONX_INF`

Parameter used for vector e^{-x}

`PQ_SIN_INF`

Parameter used for vector $\sin(x)$

`PQ_COS_INF`

Parameter used for vector $\cos(x)$

`PQ_RUN_OPCODE_R3_R2(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_RUN_OPCODE_R5_R4(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_RUN_OPCODE_R7_R6(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_Vector8_FP(middle, last, BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)`

`PQ_RUN_OPCODE_R2_R3(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_RUN_OPCODE_R4_R5(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_RUN_OPCODE_R6_R7(BATCH_OPCODE, BATCH_MACHINE)`

`PQ_Vector8_FX(middle, last, BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)`

`PQ_Initiate_Vector_Func(pSrc, pDst)`

Start 32-bit data vector calculation.

Start the vector calculation, the input data could be float, int32_t or Q31.

Parameters

- pSrc – Pointer to the source data.
- pDst – Pointer to the destination data.

`PQ_End_Vector_Func()`

End vector calculation.

This function should be called after vector calculation.

PQ_StartVector(PSRC, PDST, LENGTH)

Start 32-bit data vector calculation.

Start the vector calculation, the input data could be float, int32_t or Q31.

Parameters

- PSRC – Pointer to the source data.
- PDST – Pointer to the destination data.
- LENGTH – Number of the data, must be multiple of 8.

PQ_StartVectorFixed16(PSRC, PDST, LENGTH)

Start 16-bit data vector calculation.

Start the vector calculation, the input data could be int16_t. This function should be use with PQ_Vector8Fixed16.

Parameters

- PSRC – Pointer to the source data.
- PDST – Pointer to the destination data.
- LENGTH – Number of the data, must be multiple of 8.

PQ_StartVectorQ15(PSRC, PDST, LENGTH)

Start Q15-bit data vector calculation.

Start the vector calculation, the input data could be Q15. This function should be use with PQ_Vector8Q15. This function is dedicate for SinQ15/CosQ15 vector calculation. Because PowerQuad only supports Q31 Sin/Cos fixed function, so the input Q15 data is left shift 16 bits first, after Q31 calculation, the output data is right shift 16 bits.

Parameters

- PSRC – Pointer to the source data.
- PDST – Pointer to the destination data.
- LENGTH – Number of the data, must be multiple of 8.

PQ_EndVector()

End vector calculation.

This function should be called after vector calculation.

PQ_Vector8F32(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Float data vector calculation.

Float data vector calculation, the input data should be float. The parameter could be PQ_LN_INF, PQ_INV_INF, PQ_SQRT_INF, PQ_ISQRT_INF, PQ_ETOX_INF, PQ_ETONX_INF. For example, to calculate sqrt of a vector, use like this:

```
#define VECTOR_LEN 8
float input[VECTOR_LEN] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
float output[VECTOR_LEN];

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8F32(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_Vector8Fixed32(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Fixed 32bits data vector calculation.

Float data vector calculation, the input data should be 32-bit integer. The parameter could be PQ_LN_INF, PQ_INV_INF, PQ_SQRT_INF, PQ_ISQRT_INF, PQ_ETOX_INF, PQ_ETONX_INF.

PQ_SIN_INF, PQ_COS_INF. When this function is used for sin/cos calculation, the input data should be in the format Q1.31. For example, to calculate sqrt of a vector, use like this:

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 4, 9, 16, 25, 36, 49, 64};
int32_t output[VECTOR_LEN];

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8F32(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_Vector8Fixed16(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Fixed 32bits data vector calculation.

Float data vector calculation, the input data should be 16-bit integer. The parameter could be PQ_LN_INF, PQ_INV_INF, PQ_SQRT_INF, PQ_ISQRT_INF, PQ_ETOX_INF, PQ_ETONX_INF. For example, to calculate sqrt of a vector, use like this:

```
#define VECTOR_LEN 8
int16_t input[VECTOR_LEN] = {1, 4, 9, 16, 25, 36, 49, 64};
int16_t output[VECTOR_LEN];

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8F32(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_Vector8Q15(BATCH_OPCODE, DOUBLE_READ_ADDERS, BATCH_MACHINE)

Q15 data vector calculation.

Q15 data vector calculation, this function should only be used for sin/cos Q15 calculation, and the coprocessor output prescaler must be set to 31 before this function. This function loads Q15 data and left shift 16 bits, calculate and right shift 16 bits, then stores to the output array. The input range -1 to 1 means -pi to pi. For example, to calculate sin of a vector, use like this:

```
#define VECTOR_LEN 8
int16_t input[VECTOR_LEN] = {...}
int16_t output[VECTOR_LEN];
const pq_prescale_t prescale =
{
    .inputPrescale = 0,
    .outputPrescale = 31,
    .outputSaturate = 0
};

PQ_SetCoproprocessorScaler(POWERQUAD, const pq_prescale_t *prescale);

PQ_StartVectorQ15(pSrc, pDst, length);
PQ_Vector8Q15(PQ_SQRT_INF);
PQ_EndVector();
```

PQ_DF2_Vector8_FP(middle, last)

Float data vector biquad direct form II calculation.

Biquad filter, the input and output data are float data. Biquad side 0 is used. Example:

```
#define VECTOR_LEN 16
float input[VECTOR_LEN] = {1024.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
```

(continues on next page)

(continued from previous page)

```

    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_Initiate_Vector_Func(pSrc,pDst);
PQ_DF2_Vector8_FP(false,false);
PQ_DF2_Vector8_FP(true,true);
PQ_End_Vector_Func();

```

PQ_DF2_Vector8_FX(middle, last)

Fixed data vector biquad direct form II calculation.

Biquad filter, the input and output data are fixed data. Biquad side 0 is used. Example:

```

#define VECTOR_LEN 16
int32_t input[VECTOR_LEN] = {1024, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_Initiate_Vector_Func(pSrc,pDst);
PQ_DF2_Vector8_FX(false,false);
PQ_DF2_Vector8_FX(true,true);
PQ_End_Vector_Func();

```

PQ_Vector8BiquadDf2F32()

Float data vector biquad direct form II calculation.

Biquad filter, the input and output data are float data. Biquad side 0 is used. Example:

```

#define VECTOR_LEN 8
float input[VECTOR_LEN] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
float output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

```

(continues on next page)

(continued from previous page)

```
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2F32();
PQ_EndVector();
```

PQ_Vector8BiquadDf2Fixed32()

Fixed 32-bit data vector biquad direct form II calculation.

Biquad filter, the input and output data are Q31 or 32-bit integer. Biquad side 0 is used.
Example:

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2Fixed32();
PQ_EndVector();
```

PQ_Vector8BiquadDf2Fixed16()

Fixed 16-bit data vector biquad direct form II calculation.

Biquad filter, the input and output data are Q15 or 16-bit integer. Biquad side 0 is used.
Example:

```
#define VECTOR_LEN 8
int16_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int16_t output[VECTOR_LEN];
pq_biquad_state_t state =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2Fixed16();
PQ_EndVector();
```

PQ_DF2_Cascade_Vector8_FP(middle, last)

Float data vector direct form II biquad cascade filter.

The input and output data are float data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 16
float input[VECTOR_LEN] = {1024.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_Initiate_Vector_Func(pSrc, pDst);
PQ_DF2_Cascade_Vector8_FP(false, false);
PQ_DF2_Cascade_Vector8_FP(true, true);
PQ_End_Vector_Func();
```

PQ_DF2_Cascade_Vector8_FX(middle, last)

Fixed data vector direct form II biquad cascade filter.

The input and output data are fixed data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 16
int32_t input[VECTOR_LEN] = {1024.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
```

(continues on next page)

(continued from previous page)

```

{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_Initiate_Vector_Func(pSrc, pDst);
PQ_DF2_Cascade_Vector8_FX(false, false);
PQ_DF2_Cascade_Vector8_FX(true, true);
PQ_End_Vector_Func();

```

PQ_Vector8BiquadDf2CascadeF32()

Float data vector direct form II biquad cascade filter.

The input and output data are float data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```

#define VECTOR_LEN 8
float input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
float output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2CascadeF32();
PQ_EndVector();

```

PQ_Vector8BiquadDf2CascadeFixed32()

Fixed 32-bit data vector direct form II biquad cascade filter.

The input and output data are fixed 32-bit data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);

PQ_StartVector(input, output, VECTOR_LEN);
PQ_Vector8BiquadDf2CascadeFixed32();
PQ_EndVector();
```

PQ_Vector8BiquadDf2CascadeFixed16()

Fixed 16-bit data vector direct form II biquad cascade filter.

The input and output data are fixed 16-bit data. The data flow is input -> biquad side 1 -> biquad side 0 -> output.

```
#define VECTOR_LEN 8
int32_t input[VECTOR_LEN] = {1, 2, 3, 4, 5, 6, 7, 8};
int32_t output[VECTOR_LEN];
pq_biquad_state_t state0 =
{
    .param =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

pq_biquad_state_t state1 =
{
    .param =
    {
        .a_1 = xxx,
```

(continues on next page)

(continued from previous page)

```
.a_2 = xxx,  
.b_0 = xxx,  
.b_1 = xxx,  
.b_2 = xxx,  
},  
};  
  
PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state0);  
PQ_BiquadRestoreInternalState(POWERQUAD, 1, &state1);  
  
PQ_StartVector(input, output, VECTOR_LEN);  
PQ_Vector8BiquadDf2CascadeFixed16();  
PQ_EndVector();
```

POWERQUAD_MAKE_MATRIX_LEN(mat1Row, mat1Col, mat2Col)

Make the length used for matrix functions.

PQ_Q31_2_FLOAT(x)

Convert Q31 to float.

PQ_Q15_2_FLOAT(x)

Convert Q15 to float.

struct pq_prescale_t

#include <fsl_powerquad.h> Coprocessor prescale.

Public Members

int8_t inputPrescale

Input prescale.

int8_t outputPrescale

Output prescale.

int8_t outputSaturate

Output saturate at n bits, for example 0x11 is 8 bit space, the value will be truncated at +127 or -128.

struct pq_config_t

#include <fsl_powerquad.h> powerquad data structure format

Public Members

pq_format_t inputAFormat

Input A format.

int8_t inputAPrescale

Input A prescale, for example 1.5 can be $1.5 \cdot 2^n$ if you scale by 'shifting' ('scaling' by a factor of n).

pq_format_t inputBFormat

Input B format.

int8_t inputBPrescale

Input B prescale.

pq_format_t outputFormat

Out format.

int8_t outputPrescale
Out prescale.

pq_format_t tmpFormat
Temp format.

int8_t tmpPrescale
Temp prescale.

pq_format_t machineFormat
Machine format.

uint32_t *tmpBase
Tmp base address.

struct __pq_biquad_param
#include <fsl_powerquad.h> Struct to save biquad parameters.

Public Members

float v_n_1
v[n-1], set to 0 when initialization.

float v_n
v[n], set to 0 when initialization.

float a_1
a[1]

float a_2
a[2]

float b_0
b[0]

float b_1
b[1]

float b_2
b[2]

struct __pq_biquad_state
#include <fsl_powerquad.h> Struct to save biquad state.

Public Members

pq_biquad_param_t param
Filter parameter.

uint32_t compreg
Internal register, set to 0 when initialization.

struct pq_biquad_cascade_df2_instance
#include <fsl_powerquad.h> Instance structure for the direct form II Biquad cascade filter.

Public Members

uint8_t numStages

Number of 2nd order stages in the filter.

pq_biquad_state_t *pState

Points to the array of state coefficients.

union _pq_float

#include <fsl_powerquad.h> Conversion between integer and float type.

Public Members

float floatX

Float type.

uint32_t integerX

Unsigned interger type.

2.47 Reset Driver

enum _RSTCTL_RSTn

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control bits in RSTCLTx registers

Values:

enumerator kPOWERQUAD_RST_SHIFT_RSTn

POWERQUAD reset control

enumerator kPKC_RST_SHIFT_RSTn

PKC reset control

enumerator kELS_RST_SHIFT_RSTn

ELS reset control

enumerator kPUF_RST_SHIFT_RSTn

Physical unclonable function reset control

enumerator kFLEXSPI_RST_SHIFT_RSTn

FLEXSPI reset control

enumerator kHPU_RST_SHIFT_RSTn

HPU reset control

enumerator kUSB_RST_SHIFT_RSTn

USB reset control

enumerator kSCT_RST_SHIFT_RSTn

Standard ctimers reset control

enumerator kAON_MEM_RST_SHIFT_RSTn

AON MEM reset control

enumerator kGDMA_RST_SHIFT_RSTn

GDMA reset control

enumerator kDMA0_RST_SHIFT_RSTn

DMA0 reset control

enumerator kDMA1_RST_SHIFT_RSTn
DMA1 reset control

enumerator kSDIO_RST_SHIFT_RSTn
SDIO reset control

enumerator kELS_APB_RST_SHIFT_RSTn
ELS_APB reset control

enumerator kELS_GDET_REF_RST_SHIFT_RSTn
ELS_GDET_REF_RST reset control

enumerator kSDIO_SLV_SHIFT_RSTn
SDIO_SLV reset control

enumerator kGAU_RST_SHIFT_RSTn
GAU reset control

enumerator kOTP_RST_SHIFT_RSTn
OTP reset control

enumerator kSECURE_GPIO_RST_SHIFT_RSTn
Security GPIO reset control

enumerator kENET_IPG_RST_SHIFT_RSTn
ENET_IPG reset control

enumerator kENET_IPG_S_RST_SHIFT_RSTn
ENET_IPG_S reset control

enumerator kTRNG_RST_SHIFT_RSTn
TRNG reset control

enumerator kUTICK_RST_SHIFT_RSTn
Micro-tick timer reset control

enumerator kWWDT_RST_SHIFT_RSTn
Windowed Watchdog timer reset control

enumerator kUSIM_RST_SHIFT_RSTn
USIM reset control

enumerator kFREEMRT_RST_SHIFT_RSTn
FREEMRT reset control

enumerator kLCDIC_RST_SHIFT_RSTn
LCDIC reset control

enumerator kFC0_RST_SHIFT_RSTn
Flexcomm Interface 0 reset control

enumerator kFC1_RST_SHIFT_RSTn
Flexcomm Interface 1 reset control

enumerator kFC2_RST_SHIFT_RSTn
Flexcomm Interface 2 reset control

enumerator kFC3_RST_SHIFT_RSTn
Flexcomm Interface 3 reset control

enumerator kFC14_RST_SHIFT_RSTn
Flexcomm Interface 14 reset control

enumerator kDMIC_RST_SHIFT_RSTn
Digital microphone interface reset control

enumerator kOSEVENT_TIMER_RST_SHIFT_RSTn
Osevent Timer reset control

enumerator kHSGPIO0_RST_SHIFT_RSTn
HSGPIO 0 reset control

enumerator kHSGPIO1_RST_SHIFT_RSTn
HSGPIO 1 reset control

enumerator kCRC_RST_SHIFT_RSTn
CRC reset control

enumerator kFREQME_RST_SHIFT_RSTn
Frequency Measure reset control

enumerator kCT32B0_RST_SHIFT_RSTn
CT32B0 reset control

enumerator kCT32B1_RST_SHIFT_RSTn
CT32B1 reset control

enumerator kCT32B2_RST_SHIFT_RSTn
CT32B3 reset control

enumerator kCT32B3_RST_SHIFT_RSTn
CT32B4 reset control

enumerator kCT32B4_RST_SHIFT_RSTn
CT32B4 reset control

enumerator kMRT_RST_SHIFT_RSTn
Multi-rate timer (MRT) reset control

enumerator kPINT_RST_SHIFT_RSTn
GPIO_INT reset control

enumerator kINPUTMUX_RST_SHIFT_RSTn
PMUX reset control

typedef enum *RSTCTL_RSTn* RSTCTL_RSTn_t
Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control bits in RSTCLTx registers

typedef *RSTCTL_RSTn_t* reset_ip_name_t
IP reset handle.

void RESET_SetPeripheralReset(*reset_ip_name_t* peripheral)
Assert reset to peripheral.
Asserts reset signal to specified peripheral module.

Parameters

- peripheral – Assert reset to this peripheral. The enum argument contains encoding of reset register and reset bit position in the reset register.

void RESET_ClearPeripheralReset(*reset_ip_name_t* peripheral)
Clear reset to peripheral.
Clears reset signal to specified peripheral module, allows it to operate.

Parameters

- `peripheral` – Clear reset to this peripheral. The enum argument contains encoding of reset register and reset bit position in the reset register.

`void RESET_PeripheralReset(reset_ip_name_t peripheral)`

Reset peripheral module.

Reset peripheral module.

Parameters

- `peripheral` – Peripheral to reset. The enum argument contains encoding of reset register and reset bit position in the reset register.

`static inline void RESET_ReleasePeripheralReset(reset_ip_name_t peripheral)`

Release peripheral module.

Release peripheral module.

Parameters

- `peripheral` – Peripheral to release. The enum argument contains encoding of reset register and reset bit position in the reset register.

`FSL_RESET_DRIVER_VERSION`

reset driver version 2.1.1.

`RST_CTL0_PSCCTL0`

Reset control registers index.

`RST_CTL0_PSCCTL1`

`RST_CTL0_PSCCTL2`

`RST_CTL1_PSCCTL0`

`RST_CTL1_PSCCTL1`

`RST_CTL1_PSCCTL2`

`CRC_RSTS`

Array initializers with peripheral reset bits

`DMA_RSTS_N`

`DMIC_RSTS`

`FLEXCOMM_RSTS`

`GPIO_RSTS_N`

`MRT_RSTS`

`PINT_RSTS`

`SCT_RSTS`

`CTIMER_RSTS`

`USB_RSTS`

`UTICK_RSTS`

`WWDT_RSTS`

`OSTIMER_RSTS`

POWERQUAD_RSTS

PUF_RSTS

TRNG_RSTS

USIM_RSTS

ENET_RSTS

2.48 RTC: Real Time Clock

`void RTC_Init(RTC_Type *base)`

Un-gate the RTC clock and enable the RTC oscillator.

Note: This API should be called at the beginning of the application using the RTC driver.

Parameters

- `base` – RTC peripheral base address

`static inline void RTC_Deinit(RTC_Type *base)`

Stop the timer and gate the RTC clock.

Parameters

- `base` – RTC peripheral base address

`status_t RTC_SetDatetime(RTC_Type *base, const rtc_datetime_t *datetime)`

Set the RTC date and time according to the given time structure.

The RTC counter must be stopped prior to calling this function as writes to the RTC seconds register will fail if the RTC counter is running.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to structure where the date and time details to set are stored

Returns

`kStatus_Success`: Success in setting the time and starting the RTC
`kStatus_InvalidArgument`: Error because the datetime format is incorrect

`void RTC_GetDatetime(RTC_Type *base, rtc_datetime_t *datetime)`

Get the RTC time and stores it in the given time structure.

Parameters

- `base` – RTC peripheral base address
- `datetime` – Pointer to structure where the date and time details are stored.

`status_t RTC_SetAlarm(RTC_Type *base, const rtc_datetime_t *alarmTime)`

Set the RTC alarm time.

The function checks whether the specified alarm time is greater than the present time. If not, the function does not set the alarm and returns an error.

Parameters

- `base` – RTC peripheral base address

- alarmTime – Pointer to structure where the alarm time is stored.

Returns

kStatus_Success: success in setting the RTC alarm
kStatus_InvalidArgument: Error because the alarm datetime format is incorrect
kStatus_Fail: Error because the alarm time has already passed

```
void RTC_GetAlarm(RTC_Type *base, rtc_datetime_t *datetime)
```

Return the RTC alarm time.

Parameters

- base – RTC peripheral base address
- datetime – Pointer to structure where the alarm date and time details are stored.

```
static inline void RTC_EnableWakeupTimer(RTC_Type *base, bool enable)
```

Enable the RTC wake-up timer (1KHZ).

After calling this function, the RTC driver will use/un-use the RTC wake-up (1KHZ) at the same time.

Parameters

- base – RTC peripheral base address
- enable – Use/Un-use the RTC wake-up timer.
 - true: Use RTC wake-up timer at the same time.
 - false: Un-use RTC wake-up timer, RTC only use the normal seconds timer by default.

```
static inline uint32_t RTC_GetEnabledWakeupTimer(RTC_Type *base)
```

Get the enabled status of the RTC wake-up timer (1KHZ).

Parameters

- base – RTC peripheral base address

Returns

The enabled status of RTC wake-up timer (1KHZ).

```
static inline void RTC_EnableSubsecCounter(RTC_Type *base, bool enable)
```

Enable the RTC Sub-second counter (32KHZ).

Note: Only enable sub-second counter after RTC_ENA bit has been set to 1.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable RTC sub-second counter.
 - true: Enable RTC sub-second counter.
 - false: Disable RTC sub-second counter.

```
static inline uint32_t RTC_GetSubsecValue(const RTC_Type *base)
```

A read of 32KHZ sub-seconds counter.

Parameters

- base – RTC peripheral base address

Returns

Current value of the SUBSEC register

static inline void RTC_EnableWakeUpTimerInterruptFromDPD(RTC_Type *base, bool enable)
Enable the wake-up timer interrupt from deep power down mode.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable wake-up timer interrupt from deep power down mode.
 - true: Enable wake-up timer interrupt from deep power down mode.
 - false: Disable wake-up timer interrupt from deep power down mode.

static inline void RTC_EnableAlarmTimerInterruptFromDPD(RTC_Type *base, bool enable)
Enable the alarm timer interrupt from deep power down mode.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable alarm timer interrupt from deep power down mode.
 - true: Enable alarm timer interrupt from deep power down mode.
 - false: Disable alarm timer interrupt from deep power down mode.

static inline void RTC_EnableInterrupts(RTC_Type *base, uint32_t mask)
Enables the selected RTC interrupts.

Deprecated:

Do not use this function. It has been superseded by RTC_EnableAlarmTimerInterruptFromDPD and RTC_EnableWakeUpTimerInterruptFromDPD

Parameters

- base – RTC peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

static inline void RTC_DisableInterrupts(RTC_Type *base, uint32_t mask)
Disables the selected RTC interrupts.

Deprecated:

Do not use this function. It has been superseded by RTC_EnableAlarmTimerInterruptFromDPD and RTC_EnableWakeUpTimerInterruptFromDPD

Parameters

- base – RTC peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration rtc_interrupt_enable_t

static inline uint32_t RTC_GetEnabledInterrupts(RTC_Type *base)
Get the enabled RTC interrupts.

Deprecated:

Do not use this function. It will be deleted in next release version.

Parameters

- base – RTC peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `rtc_interrupt_enable_t`

```
static inline uint32_t RTC_GetStatusFlags(RTC_Type *base)
```

Get the RTC status flags.

Parameters

- base – RTC peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `rtc_status_flags_t`

```
static inline void RTC_ClearStatusFlags(RTC_Type *base, uint32_t mask)
```

Clear the RTC status flags.

Parameters

- base – RTC peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `rtc_status_flags_t`

```
static inline void RTC_EnableTimer(RTC_Type *base, bool enable)
```

Enable the RTC timer counter.

After calling this function, the RTC inner counter increments once a second when only using the RTC seconds timer (1hz), while the RTC inner wake-up timer countdown once a millisecond when using RTC wake-up timer (1KHZ) at the same time. RTC timer contain two timers, one is the RTC normal seconds timer, the other one is the RTC wake-up timer; the RTC enable bit is the master switch for the whole RTC timer, so user can use the RTC seconds (1HZ) timer independently, but they can't use the RTC wake-up timer (1KHZ) independently.

Parameters

- base – RTC peripheral base address
- enable – Enable/Disable RTC Timer counter.
 - true: Enable RTC Timer counter.
 - false: Disable RTC Timer counter.

```
static inline void RTC_StartTimer(RTC_Type *base)
```

Starts the RTC time counter.

Deprecated:

Do not use this function. It has been superceded by `RTC_EnableTimer`

After calling this function, the timer counter increments once a second provided `SR[TOF]` or `SR[TIF]` are not set.

Parameters

- base – RTC peripheral base address

static inline void RTC_StopTimer(RTC_Type *base)

Stops the RTC time counter.

Deprecated:

Do not use this function. It has been superceded by RTC_EnableTimer
RTC's seconds register can be written to only when the timer is stopped.

Parameters

- base – RTC peripheral base address

FSL_RTC_DRIVER_VERSION

Version 2.2.0

enum _rtc_interrupt_enable

List of RTC interrupts.

Values:

enumerator kRTC_AlarmInterruptEnable

Alarm interrupt.

enumerator kRTC_WakeupInterruptEnable

Wake-up interrupt.

enum _rtc_status_flags

List of RTC flags.

Values:

enumerator kRTC_AlarmFlag

Alarm flag

enumerator kRTC_WakeupFlag

1kHz wake-up timer flag

typedef enum _rtc_interrupt_enable rtc_interrupt_enable_t

List of RTC interrupts.

typedef enum _rtc_status_flags rtc_status_flags_t

List of RTC flags.

typedef struct _rtc_datetime rtc_datetime_t

Structure is used to hold the date and time.

static inline void RTC_SetSecondsTimerMatch(RTC_Type *base, uint32_t matchValue)

Set the RTC seconds timer (1HZ) MATCH value.

Parameters

- base – RTC peripheral base address
- matchValue – The value to be set into the RTC MATCH register

static inline uint32_t RTC_GetSecondsTimerMatch(RTC_Type *base)

Read actual RTC seconds timer (1HZ) MATCH value.

Parameters

- base – RTC peripheral base address

Returns

The actual RTC seconds timer (1HZ) MATCH value.

```
static inline void RTC_SetSecondsTimerCount(RTC_Type *base, uint32_t countValue)
```

Set the RTC seconds timer (1HZ) COUNT value.

Parameters

- base – RTC peripheral base address
- countValue – The value to be loaded into the RTC COUNT register

```
static inline uint32_t RTC_GetSecondsTimerCount(RTC_Type *base)
```

Read the actual RTC seconds timer (1HZ) COUNT value.

Parameters

- base – RTC peripheral base address

Returns

The actual RTC seconds timer (1HZ) COUNT value.

```
static inline void RTC_SetWakeupCount(RTC_Type *base, uint16_t wakeupValue)
```

Enable the RTC wake-up timer (1KHZ) and set countdown value to the RTC WAKE register.

Parameters

- base – RTC peripheral base address
- wakeupValue – The value to be loaded into the WAKE register in RTC wake-up timer (1KHZ).

```
static inline uint16_t RTC_GetWakeupCount(RTC_Type *base)
```

Read the actual value from the WAKE register value in RTC wake-up timer (1KHZ)

Read the WAKE register twice and compare the result, if the value match, the time can be used.

Parameters

- base – RTC peripheral base address

Returns

The actual value of the WAKE register value in RTC wake-up timer (1KHZ).

```
static inline void RTC_Reset(RTC_Type *base)
```

Perform a software reset on the RTC module.

This resets all RTC registers to their reset value. The bit is cleared by software explicitly clearing it.

Parameters

- base – RTC peripheral base address

```
struct _rtc_datetime
```

#include <fsl_rtc.h> Structure is used to hold the date and time.

Public Members

uint16_t year

Range from 1970 to 2099.

uint8_t month

Range from 1 to 12.

uint8_t day

Range from 1 to 31 (depending on month).

uint8_t hour

Range from 0 to 23.

uint8_t minute

Range from 0 to 59.

uint8_t second

Range from 0 to 59.

2.49 Sbloader

enum __bl_status_groups

Values:

enumerator kStatusGroup__SBLoader

enum __sbloader_status

Values:

enumerator kStatusRomLdrSectionOverrun

enumerator kStatusRomLdrSignature

enumerator kStatusRomLdrSectionLength

enumerator kStatusRomLdrUnencryptedOnly

enumerator kStatusRomLdrEOFReached

enumerator kStatusRomLdrChecksum

enumerator kStatusRomLdrCrc32Error

enumerator kStatusRomLdrUnknownCommand

enumerator kStatusRomLdrIdNotFound

enumerator kStatusRomLdrDataUnderrun

enumerator kStatusRomLdrJumpReturned

enumerator kStatusRomLdrCallFailed

enumerator kStatusRomLdrKeyNotFound

enumerator kStatusRomLdrSecureOnly

enumerator kStatusRomLdrResetReturned

enumerator kStatusRomLdrRollbackBlocked

enumerator kStatusRomLdrInvalidSectionMacCount

enumerator kStatusRomLdrUnexpectedCommand

enumerator kStatusRomLdrBadSBKEK

enumerator kStatusRomLdrPendingJumpCommand

enum _sectionType

sb3 section definitions

section type

Values:

enumerator kSectionNone

enumerator kSectionDataRange

enumerator kSectionDiffUpdate

enumerator kSectionDDRConfig

enumerator kSectionRegister

Values:

enumerator kFwVerChk_Id_none

enumerator kFwVerChk_Id_nonsecure

enumerator kFwVerChk_Id_secure

enum _loader_command_sb3

Values:

enumerator kSB3_CmdInvalid

enumerator kSB3_CmdErase

enumerator kSB3_CmdLoad

enumerator kSB3_CmdExecute

enumerator kSB3_CmdCall

enumerator kSB3_CmdProgramFuse

enumerator kSB3_CmdProgramIFR

enumerator kSB3_CmdLoadCmac

enumerator kSB3_CmdCopy

enumerator kSB3_CmdLoadHashLocking

enumerator kSB3_CmdLoadKeyBlob

enumerator kSB3_CmdConfigMem

enumerator kSB3_CmdFillMem

enumerator kSB3_CmdFwVerCheck

typedef uint8_t chunk_v3_t[16]

typedef struct _ldr_buf ldr_buf_t

typedef struct _ldr_Context_v3 ldr_Context_v3_t

typedef status_t (*pLdrFnc_v3_t)(ldr_Context_v3_t *content)

Function pointer definition for all loader action functions.

```
typedef enum _sectionType section_type_t
    sb3 section definitions
    section type
typedef struct range_header sb3_data_range_header_t
    section data range structure
typedef struct range_header_expansion sb3_data_range_expansion_t
typedef struct copy_memory_expansion sb3_copy_memory_expansion_t
typedef struct copy sb3_copy_memory_t
typedef struct load_keyblob sb3_load_keyblob_t
typedef struct fill_memory_expansion sb3_fill_memory_expansion_t
typedef struct fill_memory sb3_fill_memory_t
typedef struct config_memory sb3_config_memory_t
typedef struct fw_ver_check sb3_fw_ver_check_t
typedef struct section_header sb3_section_header_t
    sb3 DATA section header format
typedef enum _loader_command_sb3 sb3_cmd_t

SB3_BYTES_PER_CHUNK
    Defines the number of bytes in a cipher block (chunk). This is dictated by the encryption
    algorithm.

SB3_DATA_RANGE_HEADER_FLAGS_ERASE_MASK
SB3_DATA_RANGE_HEADER_FLAGS_LOAD_MASK
SB3_DATA_RANGE_HEADER_TAG
SB3_DATA_ALIGNMENT_SIZE_IN_BYTE
SB3_LOAD_KEY_BLOB_OTP_MASK
SBLOADER_V3_CMD_SET_ALL
    The all of the allowed command.
SBLOADER_V3_CMD_SET_IN_ISP_MODE
    The allowed command set in ISP mode.
SBLOADER_V3_CMD_SET_IN_REC_MODE
    The allowed command set in recovery mode.
SBLOADER_V3_CMD_SET_IN_SEC_ATE_MODE
    The allowed command set in secure ATE mode.
SB3_DATA_BUFFER_SIZE_IN_BYTE

struct _ldr_buf
    #include <fsl_sbloader_v3.h>

struct range_header
    #include <fsl_sbloader_v3.h> section data range structure

struct range_header_expansion
    #include <fsl_sbloader_v3.h>
```

```
struct copy__memory__expansion
    #include <fsl_sbloader_v3.h>
struct copy
    #include <fsl_sbloader_v3.h>
struct load__keyblob
    #include <fsl_sbloader_v3.h>
struct fill__memory__expansion
    #include <fsl_sbloader_v3.h>
struct fill__memory
    #include <fsl_sbloader_v3.h>
struct config__memory
    #include <fsl_sbloader_v3.h>
struct fw__ver__check
    #include <fsl_sbloader_v3.h>
struct section__header
    #include <fsl_sbloader_v3.h> sb3 DATA section header format
struct __ldr__Context__v3
    #include <fsl_sbloader_v3.h> Loader context definition.
```

Public Members

pLdrFnc_v3_t Action
pointer to loader action function

uint32_t block__size
size of each block in bytes

uint32_t block__data__size
data size in bytes (NBOOT_SB3_CHUNK_SIZE_IN_BYTES)

uint32_t block__data__total
data max size in bytes (block_size * data_size)

uint32_t block__buffer__size
block0 and block size

uint32_t processedBlocks
will be used for both block0 and blockx

bool in__data__block
data block offset in a block.
in progress of handling a data block within a block

bool in__data__section
in progress of handling a data section within a data block

bool in__data__range
in progress of handling a data range within a data section

uint32_t commandSet
support command set during sb file handling

`uint8_t data_buffer[(MAX(128, NBOOT_KEY_BLOB_SIZE_IN_BYTE_MAX))]`
temporary data buffer

`uint8_t fuse_cmd_buffer[32 * 4]`
used for fuse command

`bool do_firmware_load`
special handling for firmware loading outside normal memory regions

2.50 SCTimer: SCTimer/PWM (SCT)

`status_t SCTIMER_Init(SCT_Type *base, const sctimer_config_t *config)`
Ungates the SCTimer clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application using the SCTimer driver.

Parameters

- `base` – SCTimer peripheral base address
- `config` – Pointer to the user configuration structure.

Returns

`kStatus_Success` indicates success; Else indicates failure.

`void SCTIMER_Deinit(SCT_Type *base)`

Gates the SCTimer clock.

Parameters

- `base` – SCTimer peripheral base address

`void SCTIMER_GetDefaultConfig(sctimer_config_t *config)`

Fills in the SCTimer configuration structure with the default settings.

The default values are:

```
config->enableCounterUnify = true;
config->clockMode = kSCTIMER_System_ClockMode;
config->clockSelect = kSCTIMER_Clock_On_Rise_Input_0;
config->enableBidirection_l = false;
config->enableBidirection_h = false;
config->prescale_l = 0U;
config->prescale_h = 0U;
config->outInitState = 0U;
config->inputsync = 0xFU;
```

Parameters

- `config` – Pointer to the user configuration structure.

`status_t SCTIMER_SetupPwm(SCT_Type *base, const sctimer_pwm_signal_param_t *pwmParams, sctimer_pwm_mode_t mode, uint32_t pwmFreq_Hz, uint32_t srcClock_Hz, uint32_t *event)`

Configures the PWM signal parameters.

Call this function to configure the PWM signal period, mode, duty cycle, and edge. This function will create 2 events; one of the events will trigger on match with the pulse value and the other will trigger when the counter matches the PWM period. The PWM period

event is also used as a limit event to reset the counter or change direction. Both events are enabled for the same state. The state number can be retrieved by calling the function `SCTIMER_GetCurrentStateNumber()`. The counter is set to operate as one 32-bit counter (unify bit is set to 1). The counter operates in bi-directional mode when generating a center-aligned PWM.

Note: When setting PWM output from multiple output pins, they all should use the same PWM mode i.e all PWM's should be either edge-aligned or center-aligned. When using this API, the PWM signal frequency of all the initialized channels must be the same. Otherwise all the initialized channels' PWM signal frequency is equal to the last call to the API's `pwmFreq_Hz`.

Parameters

- `base` – SCTimer peripheral base address
- `pwmParams` – PWM parameters to configure the output
- `mode` – PWM operation mode, options available in enumeration `sctimer_pwm_mode_t`
- `pwmFreq_Hz` – PWM signal frequency in Hz
- `srcClock_Hz` – SCTimer counter clock in Hz
- `event` – Pointer to a variable where the PWM period event number is stored

Returns

`kStatus_Success` on success `kStatus_Fail` If we have hit the limit in terms of number of events created or if an incorrect PWM duty cycle is passed in.

```
void SCTIMER_UpdatePwmDutycycle(SCT_Type *base, sctimer_out_t output, uint8_t  
                                dutyCyclePercent, uint32_t event)
```

Updates the duty cycle of an active PWM signal.

Before calling this function, the counter is set to operate as one 32-bit counter (unify bit is set to 1).

Parameters

- `base` – SCTimer peripheral base address
- `output` – The output to configure
- `dutyCyclePercent` – New PWM pulse width; the value should be between 1 to 100
- `event` – Event number associated with this PWM signal. This was returned to the user by the function `SCTIMER_SetupPwm()`.

```
static inline void SCTIMER_EnableInterrupts(SCT_Type *base, uint32_t mask)
```

Enables the selected SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address
- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `sctimer_interrupt_enable_t`

```
static inline void SCTIMER_DisableInterrupts(SCT_Type *base, uint32_t mask)
```

Disables the selected SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address

- `mask` – The interrupts to enable. This is a logical OR of members of the enumeration `sctimer_interrupt_enable_t`

`static inline uint32_t SCTIMER_GetEnabledInterrupts(SCT_Type *base)`

Gets the enabled SCTimer interrupts.

Parameters

- `base` – SCTimer peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `sctimer_interrupt_enable_t`

`static inline uint32_t SCTIMER_GetStatusFlags(SCT_Type *base)`

Gets the SCTimer status flags.

Parameters

- `base` – SCTimer peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `sctimer_status_flags_t`

`static inline void SCTIMER_ClearStatusFlags(SCT_Type *base, uint32_t mask)`

Clears the SCTimer status flags.

Parameters

- `base` – SCTimer peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `sctimer_status_flags_t`

`static inline void SCTIMER_StartTimer(SCT_Type *base, uint32_t countertoStart)`

Starts the SCTimer counter.

Note: In 16-bit mode, we can enable both Counter_L and Counter_H, In 32-bit mode, we only can select Counter_U.

Parameters

- `base` – SCTimer peripheral base address
- `countertoStart` – The SCTimer counters to enable. This is a logical OR of members of the enumeration `sctimer_counter_t`.

`static inline void SCTIMER_StopTimer(SCT_Type *base, uint32_t countertoStop)`

Halts the SCTimer counter.

Parameters

- `base` – SCTimer peripheral base address
- `countertoStop` – The SCTimer counters to stop. This is a logical OR of members of the enumeration `sctimer_counter_t`.

`status_t SCTIMER_CreateAndScheduleEvent(SCT_Type *base, sctimer_event_t howToMonitor, uint32_t matchValue, uint32_t whichIO, sctimer_counter_t whichCounter, uint32_t *event)`

Create an event that is triggered on a match or IO and schedule in current state.

This function will configure an event using the options provided by the user. If the event type uses the counter match, then the function will set the user provided match value into a match register and put this match register number into the event control register. The

event is enabled for the current state and the event number is increased by one at the end. The function returns the event number; this event number can be used to configure actions to be done when this event is triggered.

Parameters

- `base` – SCTimer peripheral base address
- `howToMonitor` – Event type; options are available in the enumeration `sctimer_interrupt_enable_t`
- `matchValue` – The match value that will be programmed to a match register
- `whichIO` – The input or output that will be involved in event triggering. This field is ignored if the event type is “match only”
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `event` – Pointer to a variable where the new event number is stored

Returns

`kStatus_Success` on success `kStatus_Error` if we have hit the limit in terms of number of events created or if we have reached the limit in terms of number of match registers

```
void SCTIMER_ScheduleEvent(SCT_Type *base, uint32_t event)
```

Enable an event in the current state.

This function will allow the event passed in to trigger in the current state. The event must be created earlier by either calling the function `SCTIMER_SetupPwm()` or function `SCTIMER_CreateAndScheduleEvent()`.

Parameters

- `base` – SCTimer peripheral base address
- `event` – Event number to enable in the current state

```
status_t SCTIMER_IncreaseState(SCT_Type *base)
```

Increase the state by 1.

All future events created by calling the function `SCTIMER_ScheduleEvent()` will be enabled in this new state.

Parameters

- `base` – SCTimer peripheral base address

Returns

`kStatus_Success` on success `kStatus_Error` if we have hit the limit in terms of states used

```
uint32_t SCTIMER_GetCurrentState(SCT_Type *base)
```

Provides the current state.

User can use this to set the next state by calling the function `SCTIMER_SetupNextStateAction()`.

Parameters

- `base` – SCTimer peripheral base address

Returns

The current state

```
static inline void SCTIMER_SetCounterState(SCT_Type *base, sctimer_counter_t whichCounter,
uint32_t state)
```

Set the counter current state.

The function is to set the state variable bit field of STATE register. Writing to the STATE_L, STATE_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- `state` – The counter current state number (only support range from 0~31).

```
static inline uint16_t SCTIMER_GetCounterState(SCT_Type *base, sctimer_counter_t  
                                              whichCounter)
```

Get the counter current state value.

The function is to get the state variable bit field of STATE register.

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.

Returns

The the counter current state value.

```
status_t SCTIMER_SetupCaptureAction(SCT_Type *base, sctimer_counter_t whichCounter,  
                                   uint32_t *captureRegister, uint32_t event)
```

Setup capture of the counter value on trigger of a selected event.

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- `captureRegister` – Pointer to a variable where the capture register number will be returned. User can read the captured value from this register when the specified event is triggered.
- `event` – Event number that will trigger the capture

Returns

kStatus_Success on success kStatus_Error if we have hit the limit in terms of number of match/capture registers available

```
void SCTIMER_SetCallback(SCT_Type *base, sctimer_event_callback_t callback, uint32_t event)
```

Receive notification when the event trigger an interrupt.

If the interrupt for the event is enabled by the user, then a callback can be registered which will be invoked when the event is triggered

Parameters

- `base` – SCTimer peripheral base address
- `event` – Event number that will trigger the interrupt
- `callback` – Function to invoke when the event is triggered

```
static inline void SCTIMER_SetupStateLdMethodAction(SCT_Type *base, uint32_t event, bool
                                                    fgLoad)
```

Change the load method of transition to the specified state.

Change the load method of transition, it will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- event – Event number that will change the method to trigger the state transition
- fgLoad – The method to load highest-numbered event occurring for that state to the STATE register.
 - true: Load the STATEV value to STATE when the event occurs to be the next state.
 - false: Add the STATEV value to STATE when the event occurs to be the next state.

```
static inline void SCTIMER_SetupNextStateActionwithLdMethod(SCT_Type *base, uint32_t
                                                            nextState, uint32_t event, bool
                                                            fgLoad)
```

Transition to the specified state with Load method.

This transition will be triggered by the event number that is passed in by the user, the method decide how to load the highest-numbered event occurring for that state to the STATE register.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition
- fgLoad – The method to load the highest-numbered event occurring for that state to the STATE register.
 - true: Load the STATEV value to STATE when the event occurs to be the next state.
 - false: Add the STATEV value to STATE when the event occurs to be the next state.

```
static inline void SCTIMER_SetupNextStateAction(SCT_Type *base, uint32_t nextState, uint32_t
                                                event)
```

Transition to the specified state.

Deprecated:

Do not use this function. It has been superceded by SCTIMER_SetupNextStateActionwithLdMethod

This transition will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- nextState – The next state SCTimer will transition to
- event – Event number that will trigger the state transition

```
static inline void SCTIMER_SetupEventActiveDirection(SCT_Type *base,  
                                                    sctimer_event_active_direction_t  
                                                    activeDirection, uint32_t event)
```

Setup event active direction when the counters are operating in BIDIR mode.

Parameters

- base – SCTimer peripheral base address
- activeDirection – Event generation active direction, see *sctimer_event_active_direction_t*.
- event – Event number that need setup the active direction.

```
static inline void SCTIMER_SetupOutputSetAction(SCT_Type *base, uint32_t whichIO, uint32_t  
                                              event)
```

Set the Output.

This output will be set when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to set
- event – Event number that will trigger the output change

```
static inline void SCTIMER_SetupOutputClearAction(SCT_Type *base, uint32_t whichIO,  
                                                  uint32_t event)
```

Clear the Output.

This output will be cleared when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to clear
- event – Event number that will trigger the output change

```
void SCTIMER_SetupOutputToggleAction(SCT_Type *base, uint32_t whichIO, uint32_t event)
```

Toggle the output level.

This change in the output level is triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- whichIO – The output to toggle
- event – Event number that will trigger the output change

```
static inline void SCTIMER_SetupCounterLimitAction(SCT_Type *base, sctimer_counter_t  
                                                  whichCounter, uint32_t event)
```

Limit the running counter.

The counter is limited when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be limited

```
static inline void SCTIMER_SetupCounterStopAction(SCT_Type *base, sctimer_counter_t
                                                whichCounter, uint32_t event)
```

Stop the running counter.

The counter is stopped when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be stopped

```
static inline void SCTIMER_SetupCounterStartAction(SCT_Type *base, sctimer_counter_t
                                                  whichCounter, uint32_t event)
```

Re-start the stopped counter.

The counter will re-start when the event number that is passed in by the user is triggered.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to re-start

```
static inline void SCTIMER_SetupCounterHaltAction(SCT_Type *base, sctimer_counter_t
                                                  whichCounter, uint32_t event)
```

Halt the running counter.

The counter is disabled (halted) when the event number that is passed in by the user is triggered. When the counter is halted, all further events are disabled. The HALT condition can only be removed by calling the SCTIMER_StartTimer() function.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- event – Event number that will trigger the counter to be halted

```
static inline void SCTIMER_SetupDmaTriggerAction(SCT_Type *base, uint32_t dmaNumber,
                                                uint32_t event)
```

Generate a DMA request.

DMA request will be triggered by the event number that is passed in by the user.

Parameters

- base – SCTimer peripheral base address
- dmaNumber – The DMA request to generate
- event – Event number that will trigger the DMA request

```
static inline void SCTIMER_SetCOUNTValue(SCT_Type *base, sctimer_counter_t whichCounter,
                                          uint32_t value)
```

Set the value of counter.

The function is to set the value of Count register, Writing to the COUNT_L, COUNT_H, or unified register is only allowed when the corresponding counter is halted (HALT bits are set to 1 in the CTRL register).

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.
- `value` – the counter value update to the `COUNT` register.

```
static inline uint32_t SCTIMER_GetCOUNTValue(SCT_Type *base, sctimer_counter_t
                                              whichCounter)
```

Get the value of counter.

The function is to read the value of Count register, software can read the counter registers at any time..

Parameters

- `base` – SCTimer peripheral base address
- `whichCounter` – SCTimer counter to use. In 16-bit mode, we can select `Counter_L` and `Counter_H`, In 32-bit mode, we can select `Counter_U`.

Returns

The value of counter selected.

```
static inline void SCTIMER_SetEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Set the state mask bit field of `EV_STATE` register.

Parameters

- `base` – SCTimer peripheral base address
- `event` – The `EV_STATE` register be set.
- `state` – The state value in which the event is enabled to occur.

```
static inline void SCTIMER_ClearEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Clear the state mask bit field of `EV_STATE` register.

Parameters

- `base` – SCTimer peripheral base address
- `event` – The `EV_STATE` register be clear.
- `state` – The state value in which the event is disabled to occur.

```
static inline bool SCTIMER_GetEventInState(SCT_Type *base, uint32_t event, uint32_t state)
```

Get the state mask bit field of `EV_STATE` register.

Note: This function is to check whether the event is enabled in a specific state.

Parameters

- `base` – SCTimer peripheral base address
- `event` – The `EV_STATE` register be read.
- `state` – The state value.

Returns

The the state mask bit field of `EV_STATE` register.

- `true`: The event is enable in state.
- `false`: The event is disable in state.

```
static inline uint32_t SCTIMER_GetCaptureValue(SCT_Type *base, sctimer_counter_t
                                             whichCounter, uint8_t capChannel)
```

Get the value of capture register.

This function returns the captured value upon occurrence of the events selected by the corresponding Capture Control registers occurred.

Parameters

- base – SCTimer peripheral base address
- whichCounter – SCTimer counter to use. In 16-bit mode, we can select Counter_L and Counter_H, In 32-bit mode, we can select Counter_U.
- capChannel – SCTimer capture register of capture channel.

Returns

The SCTimer counter value at which this register was last captured.

```
void SCTIMER_EventHandleIRQ(SCT_Type *base)
```

SCTimer interrupt handler.

Parameters

- base – SCTimer peripheral base address.

```
FSL_SCTIMER_DRIVER_VERSION
```

Version

```
enum __sctimer_pwm_mode
```

SCTimer PWM operation modes.

Values:

```
enumerator kSCTIMER_EdgeAlignedPwm
```

Edge-aligned PWM

```
enumerator kSCTIMER_CenterAlignedPwm
```

Center-aligned PWM

```
enum __sctimer_counter
```

SCTimer counters type.

Values:

```
enumerator kSCTIMER_Counter_L
```

16-bit Low counter.

```
enumerator kSCTIMER_Counter_H
```

16-bit High counter.

```
enumerator kSCTIMER_Counter_U
```

32-bit Unified counter.

```
enum __sctimer_input
```

List of SCTimer input pins.

Values:

```
enumerator kSCTIMER_Input_0
```

SCTIMER input 0

```
enumerator kSCTIMER_Input_1
```

SCTIMER input 1

enumerator kSCTIMER_Input_2

SCTIMER input 2

enumerator kSCTIMER_Input_3

SCTIMER input 3

enumerator kSCTIMER_Input_4

SCTIMER input 4

enumerator kSCTIMER_Input_5

SCTIMER input 5

enumerator kSCTIMER_Input_6

SCTIMER input 6

enumerator kSCTIMER_Input_7

SCTIMER input 7

enum _sctimer_out

List of SCTimer output pins.

Values:

enumerator kSCTIMER_Out_0

SCTIMER output 0

enumerator kSCTIMER_Out_1

SCTIMER output 1

enumerator kSCTIMER_Out_2

SCTIMER output 2

enumerator kSCTIMER_Out_3

SCTIMER output 3

enumerator kSCTIMER_Out_4

SCTIMER output 4

enumerator kSCTIMER_Out_5

SCTIMER output 5

enumerator kSCTIMER_Out_6

SCTIMER output 6

enumerator kSCTIMER_Out_7

SCTIMER output 7

enumerator kSCTIMER_Out_8

SCTIMER output 8

enumerator kSCTIMER_Out_9

SCTIMER output 9

enum _sctimer_pwm_level_select

SCTimer PWM output pulse mode: high-true, low-true or no output.

Values:

enumerator kSCTIMER_LowTrue

Low true pulses

enumerator kSCTIMER_HighTrue

High true pulses

enum _sctimer_clock_mode

SCTimer clock mode options.

Values:

enumerator kSCTIMER_System_ClockMode
System Clock Mode

enumerator kSCTIMER_Sampled_ClockMode
Sampled System Clock Mode

enumerator kSCTIMER_Input_ClockMode
SCT Input Clock Mode

enumerator kSCTIMER_Asynchronous_ClockMode
Asynchronous Mode

enum _sctimer_clock_select

SCTimer clock select options.

Values:

enumerator kSCTIMER_Clock_On_Rise_Input_0
Rising edges on input 0

enumerator kSCTIMER_Clock_On_Fall_Input_0
Falling edges on input 0

enumerator kSCTIMER_Clock_On_Rise_Input_1
Rising edges on input 1

enumerator kSCTIMER_Clock_On_Fall_Input_1
Falling edges on input 1

enumerator kSCTIMER_Clock_On_Rise_Input_2
Rising edges on input 2

enumerator kSCTIMER_Clock_On_Fall_Input_2
Falling edges on input 2

enumerator kSCTIMER_Clock_On_Rise_Input_3
Rising edges on input 3

enumerator kSCTIMER_Clock_On_Fall_Input_3
Falling edges on input 3

enumerator kSCTIMER_Clock_On_Rise_Input_4
Rising edges on input 4

enumerator kSCTIMER_Clock_On_Fall_Input_4
Falling edges on input 4

enumerator kSCTIMER_Clock_On_Rise_Input_5
Rising edges on input 5

enumerator kSCTIMER_Clock_On_Fall_Input_5
Falling edges on input 5

enumerator kSCTIMER_Clock_On_Rise_Input_6
Rising edges on input 6

enumerator kSCTIMER_Clock_On_Fall_Input_6
Falling edges on input 6

enumerator kSCTIMER_Clock_On_Rise_Input_7

Rising edges on input 7

enumerator kSCTIMER_Clock_On_Fall_Input_7

Falling edges on input 7

enum _sctimer_conflict_resolution

SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

Values:

enumerator kSCTIMER_ResolveNone

No change

enumerator kSCTIMER_ResolveSet

Set output

enumerator kSCTIMER_ResolveClear

Clear output

enumerator kSCTIMER_ResolveToggle

Toggle output

enum _sctimer_event_active_direction

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

Values:

enumerator kSCTIMER_ActiveIndependent

This event is triggered regardless of the count direction.

enumerator kSCTIMER_ActiveInCountUp

This event is triggered only during up-counting when BIDIR = 1.

enumerator kSCTIMER_ActiveInCountDown

This event is triggered only during down-counting when BIDIR = 1.

enum _sctimer_event

List of SCTimer event types.

Values:

enumerator kSCTIMER_InputLowOrMatchEvent

enumerator kSCTIMER_InputRiseOrMatchEvent

enumerator kSCTIMER_InputFallOrMatchEvent

enumerator kSCTIMER_InputHighOrMatchEvent

enumerator kSCTIMER_MatchEventOnly

enumerator kSCTIMER_InputLowEvent

enumerator kSCTIMER_InputRiseEvent

enumerator kSCTIMER_InputFallEvent

enumerator kSCTIMER_InputHighEvent

enumerator kSCTIMER_InputLowAndMatchEvent

enumerator kSCTIMER_InputRiseAndMatchEvent
enumerator kSCTIMER_InputFallAndMatchEvent
enumerator kSCTIMER_InputHighAndMatchEvent
enumerator kSCTIMER_OutputLowOrMatchEvent
enumerator kSCTIMER_OutputRiseOrMatchEvent
enumerator kSCTIMER_OutputFallOrMatchEvent
enumerator kSCTIMER_OutputHighOrMatchEvent
enumerator kSCTIMER_OutputLowEvent
enumerator kSCTIMER_OutputRiseEvent
enumerator kSCTIMER_OutputFallEvent
enumerator kSCTIMER_OutputHighEvent
enumerator kSCTIMER_OutputLowAndMatchEvent
enumerator kSCTIMER_OutputRiseAndMatchEvent
enumerator kSCTIMER_OutputFallAndMatchEvent
enumerator kSCTIMER_OutputHighAndMatchEvent

enum _sctimer_interrupt_enable

List of SCTimer interrupts.

Values:

enumerator kSCTIMER_Event0InterruptEnable
 Event 0 interrupt
enumerator kSCTIMER_Event1InterruptEnable
 Event 1 interrupt
enumerator kSCTIMER_Event2InterruptEnable
 Event 2 interrupt
enumerator kSCTIMER_Event3InterruptEnable
 Event 3 interrupt
enumerator kSCTIMER_Event4InterruptEnable
 Event 4 interrupt
enumerator kSCTIMER_Event5InterruptEnable
 Event 5 interrupt
enumerator kSCTIMER_Event6InterruptEnable
 Event 6 interrupt
enumerator kSCTIMER_Event7InterruptEnable
 Event 7 interrupt
enumerator kSCTIMER_Event8InterruptEnable
 Event 8 interrupt
enumerator kSCTIMER_Event9InterruptEnable
 Event 9 interrupt

enumerator kSCTIMER_Event10InterruptEnable

Event 10 interrupt

enumerator kSCTIMER_Event11InterruptEnable

Event 11 interrupt

enumerator kSCTIMER_Event12InterruptEnable

Event 12 interrupt

enum _sctimer_status_flags

List of SCTimer flags.

Values:

enumerator kSCTIMER_Event0Flag

Event 0 Flag

enumerator kSCTIMER_Event1Flag

Event 1 Flag

enumerator kSCTIMER_Event2Flag

Event 2 Flag

enumerator kSCTIMER_Event3Flag

Event 3 Flag

enumerator kSCTIMER_Event4Flag

Event 4 Flag

enumerator kSCTIMER_Event5Flag

Event 5 Flag

enumerator kSCTIMER_Event6Flag

Event 6 Flag

enumerator kSCTIMER_Event7Flag

Event 7 Flag

enumerator kSCTIMER_Event8Flag

Event 8 Flag

enumerator kSCTIMER_Event9Flag

Event 9 Flag

enumerator kSCTIMER_Event10Flag

Event 10 Flag

enumerator kSCTIMER_Event11Flag

Event 11 Flag

enumerator kSCTIMER_Event12Flag

Event 12 Flag

enumerator kSCTIMER_BusErrorLFlag

Bus error due to write when L counter was not halted

enumerator kSCTIMER_BusErrorHFlag

Bus error due to write when H counter was not halted

typedef enum _sctimer_pwm_mode sctimer_pwm_mode_t

SCTimer PWM operation modes.

```
typedef enum _sctimer_counter sctimer_counter_t
```

SCTimer counters type.

```
typedef enum _sctimer_input sctimer_input_t
```

List of SCTimer input pins.

```
typedef enum _sctimer_out sctimer_out_t
```

List of SCTimer output pins.

```
typedef enum _sctimer_pwm_level_select sctimer_pwm_level_select_t
```

SCTimer PWM output pulse mode: high-true, low-true or no output.

```
typedef struct _sctimer_pwm_signal_param sctimer_pwm_signal_param_t
```

Options to configure a SCTimer PWM signal.

```
typedef enum _sctimer_clock_mode sctimer_clock_mode_t
```

SCTimer clock mode options.

```
typedef enum _sctimer_clock_select sctimer_clock_select_t
```

SCTimer clock select options.

```
typedef enum _sctimer_conflict_resolution sctimer_conflict_resolution_t
```

SCTimer output conflict resolution options.

Specifies what action should be taken if multiple events dictate that a given output should be both set and cleared at the same time

```
typedef enum _sctimer_event_active_direction sctimer_event_active_direction_t
```

List of SCTimer event generation active direction when the counters are operating in BIDIR mode.

```
typedef enum _sctimer_event sctimer_event_t
```

List of SCTimer event types.

```
typedef void (*sctimer_event_callback_t)(void)
```

SCTimer callback typedef.

```
typedef enum _sctimer_interrupt_enable sctimer_interrupt_enable_t
```

List of SCTimer interrupts.

```
typedef enum _sctimer_status_flags sctimer_status_flags_t
```

List of SCTimer flags.

```
typedef struct _sctimer_config sctimer_config_t
```

SCTimer configuration structure.

This structure holds the configuration settings for the SCTimer peripheral. To initialize this structure to reasonable defaults, call the SCTMR_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

```
SCT_EV_STATE_STATEMSKN(x)
```

```
struct _sctimer_pwm_signal_param
```

#include <fsl_sctimer.h> Options to configure a SCTimer PWM signal.

Public Members

sctimer_out_t output

The output pin to use to generate the PWM signal

sctimer_pwm_level_select_t level

PWM output active level select.

uint8_t dutyCyclePercent

PWM pulse width, value should be between 0 to 100 0 = always inactive signal (0% duty cycle) 100 = always active signal (100% duty cycle).

struct _sctimer_config

#include <fsl_sctimer.h> SCTimer configuration structure.

This structure holds the configuration settings for the SCTimer peripheral. To initialize this structure to reasonable defaults, call the SCTMR_GetDefaultConfig() function and pass a pointer to the configuration structure instance.

The configuration structure can be made constant so as to reside in flash.

Public Members

bool enableCounterUnify

true: SCT operates as a unified 32-bit counter; false: SCT operates as two 16-bit counters. User can use the 16-bit low counter and the 16-bit high counters at the same time; for Hardware limit, user can not use unified 32-bit counter and any 16-bit low/high counter at the same time.

sctimer_clock_mode_t clockMode

SCT clock mode value

sctimer_clock_select_t clockSelect

SCT clock select value

bool enableBidirection_l

true: Up-down count mode for the L or unified counter false: Up count mode only for the L or unified counter

bool enableBidirection_h

true: Up-down count mode for the H or unified counter false: Up count mode only for the H or unified counter. This field is used only if the enableCounterUnify is set to false

uint8_t prescale_l

Prescale value to produce the L or unified counter clock

uint8_t prescale_h

Prescale value to produce the H counter clock. This field is used only if the enableCounterUnify is set to false

uint8_t outInitState

Defines the initial output value

uint8_t inputsync

SCT INSYNC value, INSYNC field in the CONFIG register, from bit9 to bit 16. it is used to define synchronization for input N: bit 9 = input 0 bit 10 = input 1 bit 11 = input 2 bit 12 = input 3 All other bits are reserved (bit13 ~bit 16). How User to set the the value for the member inputsync. IE: delay for input0, and input 1, bypasses for input 2 and input 3 MACRO definition in user level. #define INPUTSYNC0 (0U) #define INPUTSYNC1 (1U) #define INPUTSYNC2 (2U) #define INPUTSYNC3 (3U) User Code. sctimerInfo.inputsync = (1 « INPUTSYNC2) | (1 « INPUTSYNC3);

2.51 Sdioslv_sdu_driver

enum _sdioslv_status

SDIO status.

Values:

enumerator kStatus_SDIOSLV_CmdPending
previous command is under working.

enumerator kStatus_SDIOSLV_SendFull
all data slots are occupied.

enumerator kStatus_SDIOSLV_FuncEnabled
function enabled

enumerator kStatus_SDIOSLV_FuncDisabled
function disabled

enumerator kStatus_SDIOSLV_FuncSuspended
function suspended

enumerator kStatus_SDIOSLV_FuncResumed
function resumed

enumerator kStatus_SDIOSLV_FuncSendComplete
function send complete

enumerator kStatus_SDIOSLV_FuncReadComplete
function read complete

enumerator kStatus_SDIOSLV_FuncRequestBuffer
function request read buffer

enum _sdioslv_int_cpu_num

SDIO card function number.

Values:

enumerator kSDIOSLV_INT_CPUNum1
sdio interrupt to CPU1

enumerator kSDIOSLV_INT_CPUNum2
sdio interrupt to CPU2

enumerator kSDIOSLV_INT_CPUNum3
sdio interrupt to CPU3

enum _sdioslv_func_num

SDIO card function number.

Values:

enumerator kSDIOSLV_FunctionNum1
sdio function1

enumerator kSDIOSLV_FunctionNum2
sdio function2

enumerator kSDIOSLV_FunctionNum3
sdio function3

enumerator kSDIOSLV_FunctionNum4
sdio function4

```
enumerator kSDIOSLV__FunctionNum5
    sdio function5
enumerator kSDIOSLV__FunctionNum6
    sdio function6
enumerator kSDIOSLV__FunctionNum7
    sdio function7
enum __sdioslv__port__num
    SDIO port number (per function)
Values:
enumerator kSDIOSLV__DataPortNum0
    sdio dataport0
enumerator kSDIOSLV__DataPortNum1
    sdio dataport1
enumerator kSDIOSLV__DataPortNum2
    sdio dataport2
enumerator kSDIOSLV__DataPortNum3
    sdio dataport3
enumerator kSDIOSLV__DataPortNum4
    sdio dataport4
enumerator kSDIOSLV__DataPortNum5
    sdio dataport5
enumerator kSDIOSLV__DataPortNum6
    sdio dataport6
enumerator kSDIOSLV__DataPortNum7
    sdio dataport7
enumerator kSDIOSLV__DataPortNum8
    sdio dataport8
enumerator kSDIOSLV__DataPortNum9
    sdio dataport9
enumerator kSDIOSLV__DataPortNum10
    sdio dataport10
enumerator kSDIOSLV__DataPortNum11
    sdio dataport11
enumerator kSDIOSLV__DataPortNum12
    sdio dataport12
enumerator kSDIOSLV__DataPortNum13
    sdio dataport13
enumerator kSDIOSLV__DataPortNum14
    sdio dataport14
enumerator kSDIOSLV__DataPortNum15
    sdio dataport15
```

```
enumerator kSDIOSLV__DataPortNum16
    sdio dataport16
enumerator kSDIOSLV__DataPortNum17
    sdio dataport17
enumerator kSDIOSLV__DataPortNum18
    sdio dataport18
enumerator kSDIOSLV__DataPortNum19
    sdio dataport19
enumerator kSDIOSLV__DataPortNum20
    sdio dataport20
enumerator kSDIOSLV__DataPortNum21
    sdio dataport21
enumerator kSDIOSLV__DataPortNum22
    sdio dataport22
enumerator kSDIOSLV__DataPortNum23
    sdio dataport23
enumerator kSDIOSLV__DataPortNum24
    sdio dataport24
enumerator kSDIOSLV__DataPortNum25
    sdio dataport25
enumerator kSDIOSLV__DataPortNum26
    sdio dataport26
enumerator kSDIOSLV__DataPortNum27
    sdio dataport27
enumerator kSDIOSLV__DataPortNum28
    sdio dataport28
enumerator kSDIOSLV__DataPortNum29
    sdio dataport29
enumerator kSDIOSLV__DataPortNum30
    sdio dataport30
enumerator kSDIOSLV__DataPortNum31
    sdio dataport31
enumerator kSDIOSLV__CmdPortNum0
    sdio cmdport0
enum __sdioslv__bus__speed
    SDIO Bus Speed.
    Values:
enumerator kSDIOSLV__SDR12__MODE
    SDR12 mode => 25Mhz
enumerator kSDIOSLV__SDR25__MODE
    SDR25 mode => 50Mhz
```

enumerator kSDIOSLV_SDR50_MODE

SDR50 mode => 100Mhz

enumerator kSDIOSLV_SDR104_MODE

SDR104 mode => 208Mhz

enum _sdioslv_scratch_group

Scratch register group.

Values:

enumerator kSDIOSLV_ScratchGroup0

sdio scratch1 in FW18 0xnD4 n:1..7 16 bits

enumerator kSDIOSLV_ScratchGroup1

sdio scratch2 in FW18 0xnB0 n:1..7 16 bits

enumerator kSDIOSLV_ScratchGroup2

sdio scratch group 2 in SDU 0xnE8 n:1..7 32 bits

enumerator kSDIOSLV_ScratchGroup3

sdio scratch group 3 in SDU 0xnEC n:1..7 32 bits

enumerator kSDIOSLV_ScratchGroup4

sdio scratch group 4 in SDU 0xnF0 n:1..7 32 bits

enumerator kSDIOSLV_ScratchGroup5

sdio scratch group 5 in SDU 0xnF4 n:1..7 32 bits

enumerator kSDIOSLV_ScratchGroup6

sdio scratch group 6 in SDU 0xnF8 n:1..7 32 bits

enumerator kSDIOSLV_ScratchGroup7

sdio scratch group 7 in SDU 0xnFC n:1..7 32 bits

enum _sdioslv_scratch_offset

Scratch register offset in a group.

Values:

enumerator kSDIOSLV_ScratchOffset0

sdio scratchoffset0

enumerator kSDIOSLV_ScratchOffset1

sdio scratchoffset1

enumerator kSDIOSLV_ScratchOffset3

sdio scratchoffset2

enumerator kSDIOSLV_ScratchOffset4

sdio scratchoffset3

typedef enum _sdioslv_int_cpu_num sdioslv_int_cpu_num_t

SDIO card function number.

typedef enum _sdioslv_func_num sdioslv_func_t

SDIO card function number.

typedef enum _sdioslv_port_num sdioslv_port_t

SDIO port number (per function)

typedef enum _sdioslv_bus_speed sdioslv_bus_speed_t

SDIO Bus Speed.

typedef enum *_sdioslv_scratch_group* sdioslv__scratch__group__t
Scratch register group.

typedef enum *_sdioslv_scratch_offset* sdioslv__scratch__offset__t
Scratch register offset in a group.

typedef struct *_sdioslv_sdu_regmap* sdioslv__sdu__regmap__t
SDU register map version 4.

typedef void (*sdioslv__cis__table__callback__t)(const uint32_t SDU_BASE)
SDIO CIS table callback.

typedef struct *sdio_slave_config* sdio_slave_config__t
Data structure to configure SDIO handle for specific function.

FSL_SDIOSLV_SDU_DRIVER_VERSION
Driver version 1.0.0.

void SDIOSLV_Init0(void)
SDIOSLV Init0.
Call this API to Init SDIOSLV phase0.

Parameters

- void – None.

Return values

void – None.

status_t SDIOSLV_Init1(SDU_FN_CARD_Type *base, *sdio_slave_config_t* *config)
SDIOSLV Init1.

Call this API to Init SDIOSLV phase1.

Parameters

- base – FN FSR pointer.
- config – Configure for SDIO Slave.

Return values

- kStatus_Success – command is ready to be sent to host driver.
- kStatus_InvalidArgument – Invalid argument.

status_t SDIOSLV_SendCmdNonBlocking(*sdioslv_sdu_regmap_t* *regmap, uint8_t *data_addr,
uint16_t data_len)

SDIOSLV send command.

Call this API to send command to host driver. The callback is always invoked from the interrupt context.

Parameters

- regmap – FN FSR pointer.
- data_addr – Data Address.
- data_len – Data Length.

Return values

- kStatus_Success – command is ready to be sent to host driver.
- kStatus_InvalidArgument – Invalid argument.

status_t SDIOSLV_RefillCmdBuffer(*sdioslv_sdu_regmap_t* *regmap, *uint8_t* *data_addr)

SDIOSLV provide command buffer.

Call this API to provide receive command buffer to SDU driver.

Parameters

- regmap – FN FSR pointer.
- data_addr – Data Address.
- data_len – Data Length.

Return values

- kStatus_Success – buffer refill successfully.
- kStatus_Fail – fail to refill buffer.

status_t SDIOSLV_SendDataNonBlocking(*sdioslv_sdu_regmap_t* *regmap, *sdioslv_port_t* tx_port, *uint8_t* *data_addr, *uint16_t* data_len)

SDIOSLV send data transfer.

Call this API to send data to host driver. The callback is always invoked from the interrupt context.

Parameters

- regmap – FN FSR pointer.
- port – Data Port.
- data_addr – Data Address.
- data_len – Data Length.

Return values

- kStatus_Success – buffer is added to data slot with problem.
- kStatus_InvalidArgument – Invalid argument.
- kStatus_SDIOSLV_SendFull – all data slots are occupied, application

status_t SDIOSLV_RefillDataBuffer(*sdioslv_sdu_regmap_t* *regmap, *sdioslv_port_t* port, *uint8_t* *data_addr)

SDIOSLV provide receive data buffer.

Call this API to provide receive data buffer to SDU driver.

Parameters

- regmap – FN FSR pointer.
- port – Data Port.
- data_addr – Data Address.
- data_len – Data Length.

Return values

- kStatus_Success – refill buffer successfully.
- kStatus_Fail – fail to refill buffer.

sdioslv_bus_speed_t SDIOSLV_GetBusSpeed(void)

Get SDIO bus speed selection.

Call this API to get current bus speed selected for SDIO.

Parameters

- void – None.

Return values

`sdioslv__bus_speed_t` – Bus speed selected for SDIO.

`uint32_t` `SDIOSLV_GetBlockSize(uint8_t fn_num)`

Get SDIO the block size in FBR.

For block mode, block size equals to block size in FBR.

Parameters

- handle – Created by `SDIOSLV_CreateHanle()`.

Return values

the – block size in FBR.

`status_t` `SDIOSLV_ReadScratchRegister(sdioslv_func_t fun_num, sdioslv_scratch_group_t group, sdioslv_scratch_offset_t offset, uint8_t *value)`

SDIOSLV read scratch register of SDU.

Call this API to read scratch register of SDU (based on group and offset).

Parameters

- fun_num – Specify which function.
- group – Specify which group scratch register.
- offset – Specify offset of the scratch group.
- value – Value read from the register.

Return values

- `kStatus__Success` – read sucessfully.
- `kStatus__Fail` – fail to read.

`status_t` `SDIOSLV_WriteScratchRegister(sdioslv_func_t fun_num, sdioslv_scratch_group_t group, sdioslv_scratch_offset_t offset, uint8_t value)`

SDIOSLV write value to scratch register of SDU.

Call this API to write value to scratch register of SDU (based on group and offset).

Parameters

- fun_num – Specify which function.
- group – Specify which group scratch register.
- offset – Specify offset of the scratch group.
- value – Value write to the register.

Return values

- `kStatus__Success` – write sucessfully.
- `kStatus__Fail` – fail to write.

`uint32_t` `HostToCardEvent`

0x100/200.../700

`uint32_t` `HostIntCause`

0x104/204.../704

`uint32_t` `HostIntMask`

0x108/208.../708

uint32_t HostIntStatus
0x10C/20C.../70C

uint32_t RdBitMap
0x110/210.../710

uint32_t WrBitMap
0x114/214.../714

uint16_t RdLen[32]
0x118/218.../718

uint8_t HostTransferStatus
0x158/258.../758

uint8_t FunctionCardIntMsk
0x159/259.../759

uint8_t Card_Q_PTR_RANGE0
0x15A/25A.../75A

uint8_t Card_Q_PTR_RANGE1
0x15B/25B.../75B

uint16_t CardToHostEvent
0x15C/25C.../75C

uint8_t reserved2[2]

uint32_t CardIntMask
0x160/260.../760

uint32_t CardIntStatus
0x164/264.../764

uint32_t CardIntMode
0x168/268.../768

uint32_t SqReadBase
0x16C/26C.../76C

uint32_t SqWriteBase
0x170/270.../770

uint8_t RdIdx
0x174/274.../774

uint8_t WrIdx
0x175/275.../775

uint8_t Reserved6[2]
0x176/276.../776

uint8_t Card_APU_SLP_RDY_EN
0x178/278.../778

uint8_t Reserved7[3]

uint8_t Card_HOST_ERR_WKUP_EN
0x17C/27C.../77C

uint8_t Reserved8[3]

uint8_t HOST_ERR_CMD0
0x180/280.../780

uint8_t HOST_ERR_CMD1
0x181/281.../781

uint8_t HOST_ERR_CMD2
0x182/282.../782

uint8_t HOST_ERR_CMD3
0x183/283.../783

uint8_t HOST_ERR_CMD4
0x184/284.../784

uint8_t HOST_ERR_CMD5
0x185/285.../785

uint8_t Reserved9[2]

uint32_t PktWrBitmapClr
0x188/288.../788

uint32_t PktRdBitmapClr
0x18C/28C.../78C

uint32_t HostIntActMskEn
0x190/290.../790

uint32_t HostIntActMskClr
0x194/294.../794

uint32_t HostIntActMskStat
0x198/298.../798

uint32_t CardIntActMskEn
0x19C/29C.../79C

uint32_t CardIntActMskClr
0x1A0/2A0.../7A0

uint32_t CardIntActMskStat
0x1A4/2A4.../7A4

uint32_t TestbusBitSelect
0x1A8/2A8.../7A8

uint32_t TestbusBitSelect1
0x1AC/2AC.../7AC

uint16_t Scratch2
0x1B0/2B0.../7B0

uint8_t Scratch[6]
0x1B2/2B2.../7B2

uint32_t CmdPortSqWriteBase
0x1B8/2B8.../7B8

uint32_t CmdPortSqReadBase
0x1BC/2BC.../7BC

`uint16_t CmdPortRdLen`
0x1C0/2C0.../7C0

`uint16_t Reserved10`
0x1C2/2C2.../7C2

`uint32_t CmdPortConfig`
0x1C4/2C4.../7C4

`uint8_t ChipRev`
0x1C8/2C8.../7C8

`uint8_t reserved11`

`uint8_t SDUMinorIPRev`
0x1CA/2CA.../7CA

`uint8_t SDUMajorIPRev`
0x1CB/2CB.../7CB

`uint32_t Card_PKT_END_RADDR`
0x1CC/2CC.../7CC

`uint32_t Card_PKT_END_WADDR`
0x1D0/2D0.../7D0

`uint16_t Scratch1`
0x1D4/2D4.../7D4

`uint8_t Ocr2`
0x1D6/2D6.../7D6

`uint8_t Config`
0x1D7/2D7.../7D7

`uint32_t Config2`
0x1D8/2D8.../7D8

`uint32_t Debug`
0x1DC/2DC.../7DC

`uint32_t DmaAddr`
0x1E0/2E0.../7E0

`uint8_t IoPort[3]`
0x1E4/2E4.../7E4

`uint8_t fun_num`
SDIO function number (1..7).

`sdioslv_int_cpu_num_t` `cpu_num`
Specify interrupt should be generated to which CPU

`uint8_t used_port_num`
How many data ports are used inside this function

`uint8_t cmd_tx_format`
Command Tx length format. 0: no tx_len, 1: 2 bytes, 2: 3 bytes

`uint8_t cmd_rd_format`
Command Rx length format. 0: blk_num * blk_size, 1: CMD_PORT_RD_LEN

uint8_t data_tx_format

Data Tx length format. 0: no tx_len, 1: 2 bytes, 2: 3 bytes

uint8_t data_rd_format

Data Rx length format. 0: blk_num * blk_size, 1: PORT_RD_LEN[15:0], 2: PORT1_RD_LEN[7:0] && PORT0_RD_LEN[15:0]

sdioslv_cis_table_callback_t cis_table_callback

Callback function for initializing the CIS table.

SDU_INT_CPU_NUM

SDU_USED_FUN_NUM

SDU_USED_PORT_NUM

SDU_MAX_FUNCTION_NUM

Maximum functions supported by SDU.

SDU_MAX_PORT_NUM

Maximum data ports supported by SDU per function.

sdu_fbr_fnN_base(FN)

sdu_fbr_fnN_fn_code(FN)

sdu_fbr_fnN_fn_code_code_HI

sdu_fbr_fnN_fn_code_code_LO

SDU_REGS8(x)

MACRO used to access register of SDU.

SDU_READ_REGS8(reg, val)

MACRO used to read SDU register.

SDU_WRITE_REGS8(reg, val)

MACRO used to write SDU register.

SDU_REGS8_SETBITS(reg, val)

MACRO used to set bits of SDU register.

SDU_REGS8_CLRBITS(reg, val)

MACRO used to clear bits of SDU register.

SDU_SCRATCH2_OFFSET0_ADDR

Address of scratch register (group 2, offset 0) within function.

SDU_SDIO_CFG_BASE

SDU SDIO configuration base (SDU_FN0_CARD_BASE defined in device)

SDIO_CCR_FUNC_OFFSET

Address offset of CCR between two functions.

SDIO_IO_ENABLE

SDIO I/O Enable.

SDIO_FUNC0_BSS

SDIO Bus Speed Select.

SDIO_FUNC0_BSS_SUPPORT_MASK

SDIO_FUNC0_BSS_MODE_BIT

SDIO_FUNC0_BSS_MODE_MASK

SDIO_CCR_FUNC0_CARD_INT_MSK

Interrupt mask register for function 0.

SDIO_CCR_HOST_DnLdReStart

Bit Def. Host Transfer Status (HostTransferStatus)

SDIO_CCR_HOST_UpLdReStart

SDIO_CCR_HOST_DnLdCRC_err

SDIO_CCR_CS_DnLdRdy

Bit Def. Card To Host Interrupt Event (CardToHostEvent)

SDIO_CCR_CS_UpLdRdy

SDIO_CCR_CS_ReadCISRdy

SDIO_CCR_CS_CmdUpLdRdy

SDIO_CCR_CS_CmdDnLdRdy

SDIO_CCR_CIM_DnLdOvr

Bit Def. Card Interrupt Mask (CardIntMask)

SDIO_CCR_CIM_UpLdOvr

SDIO_CCR_CIM_Abort

SDIO_CCR_CIM_PwrDn

SDIO_CCR_CIM_PwrUp

SDIO_CCR_CIM_CmdUpLdOvr

SDIO_CCR_CIM_CmdDnLdOvr

SDIO_CCR_CIM_MASK

SDIO_CCR_CIC_DnLdOvr

Bit Def. Card Interrupt Status (CardIntStatus)

SDIO_CCR_CIC_UpLdOvr

SDIO_CCR_CIC_Abort

SDIO_CCR_CIC_PwrDn

SDIO_CCR_CIC_PwrUp

SDIO_CCR_CIC_CmdUpLdOvr

SDIO_CCR_CIC_CmdDnLdOvr

SDIO_CCR_CIC_ALL

SDIO_CCR_CIC_MASK

CARD_INT_MODE_MSK

Bit Def. Default setting ISR bit clear after read (CardIntMode)

CMD_TX_LEN_BIT_OFFSET

Bit Def. Command port configuration register (CmdPortConfig)

CMD_RD_LEN_BIT_OFFSET

CONFIG2_ASYNC_INT

Bit Def. Config2 register (Config2)

CONFIG2_CMD53_NEW_MODE

CONFIG2_DNLD_RDY_AUTO_RESET

CONFIG2_UPLD_RDY_AUTO_RESET

CONFIG2_TX_LEN_BIT_OFFSET

CONFIG2_RD_LEN_BIT_OFFSET

CONFIG2_DEFAULT_SETTING

struct _sdioslv_sdu_regmap

#include <fsl_sdioslv_sdu.h> SDU register map version 4.

struct sdio_slave_config

#include <fsl_sdioslv_sdu.h> Data structure to configure SDIO handle for specific function.

2.52 Smart Card

FSL_SMARTCARD_DRIVER_VERSION

Smart card driver version 2.3.0.

Smart card Error codes.

Values:

enumerator kStatus_SMARTCARD_Success

Transfer ends successfully

enumerator kStatus_SMARTCARD_TxBusy

Transmit in progress

enumerator kStatus_SMARTCARD_RxBusy

Receiving in progress

enumerator kStatus_SMARTCARD_NoTransferInProgress

No transfer in progress

enumerator kStatus_SMARTCARD_Timeout

Transfer ends with time-out

enumerator kStatus_SMARTCARD_Initialized

Smart card driver is already initialized

enumerator kStatus_SMARTCARD_PhyInitialized

Smart card PHY drive is already initialized

enumerator kStatus_SMARTCARD_CardNotActivated

Smart card is not activated

enumerator kStatus_SMARTCARD_InvalidInput

Function called with invalid input arguments

enumerator kStatus_SMARTCARD_OtherError

Some other error occur

enum _smartcard_control

Control codes for the Smart card protocol timers and misc.

Values:

enumerator kSMARTCARD_EnableADT

enumerator kSMARTCARD_DisableADT

enumerator kSMARTCARD_EnableGTV

enumerator kSMARTCARD_DisableGTV

enumerator kSMARTCARD_ResetWWT

enumerator kSMARTCARD_EnableWWT

enumerator kSMARTCARD_DisableWWT

enumerator kSMARTCARD_ResetCWT

enumerator kSMARTCARD_EnableCWT

enumerator kSMARTCARD_DisableCWT

enumerator kSMARTCARD_ResetBWT

enumerator kSMARTCARD_EnableBWT

enumerator kSMARTCARD_DisableBWT

enumerator kSMARTCARD_EnableInitDetect

enumerator kSMARTCARD_EnableAnack

enumerator kSMARTCARD_DisableAnack

enumerator kSMARTCARD_ConfigureBaudrate

enumerator kSMARTCARD_SetupATRMode

enumerator kSMARTCARD_SetupT0Mode

enumerator kSMARTCARD_SetupT1Mode

enumerator kSMARTCARD_EnableReceiverMode

enumerator kSMARTCARD_DisableReceiverMode

enumerator kSMARTCARD_EnableTransmitterMode

enumerator kSMARTCARD_DisableTransmitterMode

enumerator kSMARTCARD_ResetWaitTimeMultiplier

enum _smartcard_card_voltage_class

Defines Smart card interface voltage class values.

Values:

enumerator kSMARTCARD_VoltageClassUnknown

enumerator kSMARTCARD_VoltageClassA5_0V

enumerator kSMARTCARD_VoltageClassB3_3V

enumerator kSMARTCARD_VoltageClassC1_8V

enum _smartcard_transfer_state

Defines Smart card I/O transfer states.

Values:

enumerator kSMARTCARD_IdleState

enumerator kSMARTCARD_WaitingForTSSState

enumerator kSMARTCARD_InvalidTSDetectedState

enumerator kSMARTCARD_ReceivingState

enumerator kSMARTCARD_TransmittingState

enum _smartcard_reset_type

Defines Smart card reset types.

Values:

enumerator kSMARTCARD_ColdReset

enumerator kSMARTCARD_WarmReset

enumerator kSMARTCARD_NoColdReset

enumerator kSMARTCARD_NoWarmReset

enum _smartcard_transport_type

Defines Smart card transport protocol types.

Values:

enumerator kSMARTCARD_T0Transport

enumerator kSMARTCARD_T1Transport

enum _smartcard_parity_type

Defines Smart card data parity types.

Values:

enumerator kSMARTCARD_EvenParity

enumerator kSMARTCARD_OddParity

enum _smartcard_card_convention

Defines data Convention format.

Values:

enumerator kSMARTCARD_DirectConvention

enumerator kSMARTCARD_InverseConvention

enum _smartcard_interface_control

Defines Smart card interface IC control types.

Values:

enumerator kSMARTCARD_InterfaceSetVcc

enumerator kSMARTCARD_InterfaceSetClockToResetDelay

enumerator kSMARTCARD_InterfaceReadStatus

enum *_smartcard_direction*

Defines transfer direction.

Values:

enumerator kSMARTCARD_Receive

enumerator kSMARTCARD_Transmit

typedef enum *_smartcard_control* smartcard_control_t

Control codes for the Smart card protocol timers and misc.

typedef enum *_smartcard_card_voltage_class* smartcard_card_voltage_class_t

Defines Smart card interface voltage class values.

typedef enum *_smartcard_transfer_state* smartcard_transfer_state_t

Defines Smart card I/O transfer states.

typedef enum *_smartcard_reset_type* smartcard_reset_type_t

Defines Smart card reset types.

typedef enum *_smartcard_transport_type* smartcard_transport_type_t

Defines Smart card transport protocol types.

typedef enum *_smartcard_parity_type* smartcard_parity_type_t

Defines Smart card data parity types.

typedef enum *_smartcard_card_convention* smartcard_card_convention_t

Defines data Convention format.

typedef enum *_smartcard_interface_control* smartcard_interface_control_t

Defines Smart card interface IC control types.

typedef enum *_smartcard_direction* smartcard_direction_t

Defines transfer direction.

typedef void (*smartcard_interface_callback_t)(void *smartcardContext, void *param)

Smart card interface interrupt callback function type.

typedef void (*smartcard_transfer_callback_t)(void *smartcardContext, void *param)

Smart card transfer interrupt callback function type.

typedef void (*smartcard_time_delay_t)(uint32_t us)

Time Delay function used to passive waiting using RTOS [us].

typedef struct *_smartcard_card_params* smartcard_card_params_t

Defines card-specific parameters for Smart card driver.

typedef struct *_smartcard_timers_state* smartcard_timers_state_t

Smart card defines the state of the EMV timers in the Smart card driver.

typedef struct *_smartcard_interface_config* smartcard_interface_config_t

Defines user specified configuration of Smart card interface.

typedef struct *_smartcard_xfer* smartcard_xfer_t

Defines user transfer structure used to initialize transfer.

typedef struct *_smartcard_context* smartcard_context_t

Runtime state of the Smart card driver.

SMARTCARD_INIT_DELAY_CLOCK_CYCLES

Smart card global define which specify number of clock cycles until initial 'TS' character has to be received.

SMARTCARD_EMV_ATR_DURATION_ETU

Smart card global define which specify number of clock cycles during which ATR string has to be received.

SMARTCARD_TS_DIRECT_CONVENTION

Smart card specification initial TS character definition of direct convention.

SMARTCARD_TS_INVERSE_CONVENTION

Smart card specification initial TS character definition of inverse convention.

struct __smartcard_card_params

#include <fsl_smartcard.h> Defines card-specific parameters for Smart card driver.

Public Members

uint16_t Fi

4 bits Fi - clock rate conversion integer

uint8_t fMax

Maximum Smart card frequency in MHz

uint8_t WI

8 bits WI - work wait time integer

uint8_t Di

4 bits DI - baud rate divisor

uint8_t BWI

4 bits BWI - block wait time integer

uint8_t CWI

4 bits CWI - character wait time integer

uint8_t BGI

4 bits BGI - block guard time integer

uint8_t GTN

8 bits GTN - extended guard time integer

uint8_t IFSC

Indicates IFSC value of the card

uint8_t modeNegotiable

Indicates if the card acts in negotiable or a specific mode.

uint8_t currentD

4 bits DI - current baud rate divisor

uint8_t status

Indicates smart card status

bool t0Indicated

Indicates ff T=0 indicated in TD1 byte

bool t1Indicated

Indicates if T=1 indicated in TD2 byte

bool atrComplete

Indicates whether the ATR received from the card was complete or not

bool atrValid

Indicates whether the ATR received from the card was valid or not

bool present

Indicates if a smart card is present

bool active

Indicates if the smart card is activated

bool faulty

Indicates whether smart card/interface is faulty

smartcard_card_convention_t convention

Card convention, kSMARTCARD_DirectConvention for direct convention, kSMARTCARD_InverseConvention for inverse convention

struct __smartcard_timers_state

#include <fsl_smartcard.h> Smart card defines the state of the EMV timers in the Smart card driver.

Public Members

volatile bool adtExpired

Indicates whether ADT timer expired

volatile bool wwtExpired

Indicates whether WWT timer expired

volatile bool cwtExpired

Indicates whether CWT timer expired

volatile bool bwtExpired

Indicates whether BWT timer expired

volatile bool initCharTimerExpired

Indicates whether reception timer for initialization character (TS) after the RST has expired

struct __smartcard_interface_config

#include <fsl_smartcard.h> Defines user specified configuration of Smart card interface.

Public Members

uint32_t smartCardClock

Smart card interface clock [Hz]

uint32_t clockToResetDelay

Indicates clock to RST apply delay [smart card clock cycles]

uint8_t clockModule

Smart card clock module number

uint8_t clockModuleChannel

Smart card clock module channel number

uint8_t clockModuleSourceClock

Smart card clock module source clock [e.g., BusClk]

```

smartcard_card_voltage_class_t vcc
    Smart card voltage class
uint8_t controlPort
    Smart card PHY control port instance
uint8_t controlPin
    Smart card PHY control pin instance
uint8_t irqPort
    Smart card PHY Interrupt port instance
uint8_t irqPin
    Smart card PHY Interrupt pin instance
uint8_t resetPort
    Smart card reset port instance
uint8_t resetPin
    Smart card reset pin instance
uint8_t vsel0Port
    Smart card PHY Vsel0 control port instance
uint8_t vsel0Pin
    Smart card PHY Vsel0 control pin instance
uint8_t vsel1Port
    Smart card PHY Vsel1 control port instance
uint8_t vsel1Pin
    Smart card PHY Vsel1 control pin instance
uint8_t dataPort
    Smart card PHY data port instance
uint8_t dataPin
    Smart card PHY data pin instance
uint8_t dataPinMux
    Smart card PHY data pin mux option
uint8_t tsTimerId
    Numerical identifier of the External HW timer for Initial character detection
struct __smartcard_xfer
    #include <fsl_smartcard.h> Defines user transfer structure used to initialize transfer.

```

Public Members

```

smartcard_direction_t direction
    Direction of communication. (RX/TX)
uint8_t *buff
    The buffer of data.
size_t size
    The number of transferred units.
struct __smartcard_context
    #include <fsl_smartcard.h> Runtime state of the Smart card driver.

```

Public Members

`void *base`

Smart card module base address

`smartcard_direction_t direction`

Direction of communication. (RX/TX)

`uint8_t *xBuff`

The buffer of data being transferred.

`volatile size_t xSize`

The number of bytes to be transferred.

`volatile bool xIsBusy`

True if there is an active transfer.

`uint8_t txFifoEntryCount`

Number of data word entries in transmit FIFO.

`uint8_t rxFifoThreshold`

The max value of the receiver FIFO threshold.

`smartcard_interface_callback_t interfaceCallback`

Callback to invoke after interface IC raised interrupt.

`smartcard_transfer_callback_t transferCallback`

Callback to invoke after transfer event occur.

`void *interfaceCallbackParam`

Interface callback parameter pointer.

`void *transferCallbackParam`

Transfer callback parameter pointer.

`smartcard_time_delay_t timeDelay`

Function which handles time delay defined by user or RTOS.

`smartcard_reset_type_t resetType`

Indicates whether a Cold reset or Warm reset was requested.

`smartcard_transport_type_t tType`

Indicates current transfer protocol (T0 or T1)

`volatile smartcard_transfer_state_t transferState`

Indicates the current transfer state

`smartcard_timers_state_t timersState`

Indicates the state of different protocol timers used in driver

`smartcard_card_params_t cardParams`

Smart card parameters(ATR and current) and interface slots states(ATR and current)

`uint8_t IFSD`

Indicates the terminal IFSD

`smartcard_parity_type_t parity`

Indicates current parity even/odd

`volatile bool rxtCrossed`

Indicates whether RXT thresholds has been crossed

`volatile bool txtCrossed`

Indicates whether TXT thresholds has been crossed

`volatile bool wtxRequested`

Indicates whether WTX has been requested or not

`volatile bool parityError`

Indicates whether a parity error has been detected

`uint8_t statusBytes[2]`

Used to store Status bytes SW1, SW2 of the last executed card command response

`smartcard_interface_config_t` interfaceConfig

Smart card interface configuration structure

`bool abortTransfer`

Used to abort transfer.

2.53 Smart Card PHY Driver

`void SMARTCARD_PHY_GetDefaultConfig(smartcard_interface_config_t *config)`

Fills in the configuration structure with default values.

Parameters

- `config` – The Smart card user configuration structure which contains configuration structure of type `smartcard_interface_config_t`. Function fill in members: `clockToResetDelay` = 42000, `vcc` = `kSmartcardVoltageClassB3_3V`, with default values.

`status_t SMARTCARD_PHY_Init(void *base, smartcard_interface_config_t const *config, uint32_t srcClock_Hz)`

Initializes a Smart card interface instance.

Parameters

- `base` – The Smart card peripheral base address.
- `config` – The user configuration structure of type `smartcard_interface_config_t`. Call the function `SMARTCARD_PHY_GetDefaultConfig()` to fill the configuration structure.
- `srcClock_Hz` – Smart card clock generation module source clock.

Return values

`kStatus_SMARTCARD_Success` – or `kStatus_SMARTCARD_OtherError` in case of error.

`void SMARTCARD_PHY_Deinit(void *base, smartcard_interface_config_t const *config)`

De-initializes a Smart card interface, stops the Smart card clock, and disables the VCC.

Parameters

- `base` – The Smart card peripheral module base address.
- `config` – The user configuration structure of type `smartcard_interface_config_t`.

`status_t SMARTCARD_PHY_Activate(void *base, smartcard_context_t *context, smartcard_reset_type_t resetType)`

Activates the Smart card IC.

Parameters

- `base` – The Smart card peripheral module base address.
- `context` – A pointer to a Smart card driver context structure.
- `resetType` – type of reset to be performed, possible values = `kSmartcardColdReset`, `kSmartcardWarmReset`

Return values

`kStatus_SMARTCARD_Success` – or `kStatus_SMARTCARD_OtherError` in case of error.

`status_t SMARTCARD_PHY_Deactivate(void *base, smartcard_context_t *context)`

De-activates the Smart card IC.

Parameters

- `base` – The Smart card peripheral module base address.
- `context` – A pointer to a Smart card driver context structure.

Return values

`kStatus_SMARTCARD_Success` – or `kStatus_SMARTCARD_OtherError` in case of error.

`status_t SMARTCARD_PHY_Control(void *base, smartcard_context_t *context, smartcard_interface_control_t control, uint32_t param)`

Controls the Smart card interface IC.

Parameters

- `base` – The Smart card peripheral module base address.
- `context` – A pointer to a Smart card driver context structure.
- `control` – A interface command type.
- `param` – Integer value specific to control type

Return values

`kStatus_SMARTCARD_Success` – or `kStatus_SMARTCARD_OtherError` in case of error.

`SMARTCARD_ATR_DURATION_ADJUSTMENT`

Smart card definition which specifies the adjustment number of clock cycles during which an ATR string has to be received.

`SMARTCARD_INIT_DELAY_CLOCK_CYCLES_ADJUSTMENT`

Smart card definition which specifies the adjustment number of clock cycles until an initial 'TS' character has to be received.

2.54 Smart Card PHY USIM W

2.55 Smart Card USIM Driver

`void SMARTCARD_USIM_GetDefaultConfig(smartcard_card_params_t *cardParams)`

Fills in the `smartcard_card_params` structure with default values according to the EMV 4.3 specification.

Parameters

- `cardParams` – The configuration structure of type `smartcard_interface_config_t`. Function fill in members: `Fi` = 372; `Di` = 1; `currentD` = 1; `WI` = 0x0A; `GTN` = 0x00; with default values.

status_t SMARTCARD_USIM_Init(*USIM_Type* *base, *smartcard_context_t* *context, *uint32_t* srcClock_Hz)

Initializes an USIM peripheral for the Smart card/ISO-7816 operation.

This function un-gates the USIM clock, initializes the module to EMV default settings, configures the IRQ, enables the module-level interrupt to the core and, initializes the driver context.

Parameters

- base – The USIM peripheral base address.
- context – A pointer to the smart card driver context structure.
- srcClock_Hz – Smart card clock generation module source clock.

Returns

An error code or *kStatus_SMARTCARD_Success*.

void SMARTCARD_USIM_Deinit(*USIM_Type* *base)

This function disables the USIM interrupts, disables the transmitter and receiver, flushes the FIFOs, and gates USIM clock in SIM.

Parameters

- base – The USIM module base address.

int32_t SMARTCARD_USIM_GetTransferRemainingBytes(*USIM_Type* *base, *smartcard_context_t* *context)

Returns whether the previous USIM transfer has finished.

When performing an async transfer, call this function to ascertain the context of the current transfer: in progress (or busy) or complete (success). If the transfer is still in progress, the user can obtain the number of words that have not been transferred.

Parameters

- base – The USIM module base address.
- context – A pointer to a smart card driver context structure.

Returns

The number of bytes not transferred.

status_t SMARTCARD_USIM_AbortTransfer(*USIM_Type* *base, *smartcard_context_t* *context)

Terminates an asynchronous USIM transfer early.

During an async USIM transfer, the user can terminate the transfer early if the transfer is still in progress.

Parameters

- base – The USIM peripheral address.
- context – A pointer to a smart card driver context structure.

Returns

kStatus_SMARTCARD_Success The transmit abort was successful.

Returns

kStatus_SMARTCARD_NoTransmitInProgress No transmission is currently in progress.

status_t SMARTCARD_USIM_TransferNonBlocking(*USIM_Type* *base, *smartcard_context_t* *context, *smartcard_xfer_t* *xfer)

Transfer data using interrupts.

A non-blocking (also known as asynchronous) function means that the function returns immediately after initiating the transfer function. The application has to get the transfer

status to see when the transfer is complete. In other words, after calling the non-blocking (asynchronous) transfer function, the application must get the transfer status to check if the transmit is completed or not.

Parameters

- `base` – The USIM peripheral base address.
- `context` – A pointer to a smart card driver context structure.
- `xfer` – A pointer to the smart card transfer structure where the linked buffers and sizes are stored.

Returns

An error code or `kStatus_SMARTCARD_Success`.

```
status_t SMARTCARD_USIM_Control(USIM_Type *base, smartcard_context_t *context,  
                                smartcard_control_t control, uint32_t param)
```

Controls the USIM module per different user request.

Parameters

- `base` – The USIM peripheral base address.
- `context` – A pointer to a smart card driver context structure.
- `control` – Control type.
- `param` – Integer value of specific to control command.

Returns

`kStatus_SMARTCARD_Success` in success.

Returns

`kStatus_SMARTCARD_OtherError` in case of error.

```
void SMARTCARD_USIM_IRQHandler(USIM_Type *base, smartcard_context_t *context)
```

Handles USIM module interrupts.

Parameters

- `base` – The USIM peripheral base address.
- `context` – A pointer to a smart card driver context structure.

```
void SMARTCARD_USIM_TSExpiryCallback(USIM_Type *base, smartcard_context_t *context)
```

Handles initial TS character timer time-out event.

Parameters

- `base` – The USIM peripheral base address.
- `context` – A pointer to a Smart card driver context structure.

```
void SMARTCARD_USIM_TimerStart(uint32_t time)
```

Initializes timer with input period, enable interrupt and start counter.

Parameters

- `time` – The time period.

```
enum __usim_rx_fifo_trigger_level
```

USIM Rx FIFO receiver trigger level enumeration.

Values:

```
enumerator kUSIM_1ByteOrMore
```

1 byte or more in the RX-FIFO can trigger receiver data ready interrupt.

```

enumerator kUSIM_4ByteOrMore
    4 byte or more in the RX-FIFO can trigger receiver data ready interrupt.
enumerator kUSIM_8ByteOrMore
    8 byte or more in the RX-FIFO can trigger receiver data ready interrupt.
enumerator kUSIM_12ByteOrMore
    12 byte or more in the RX-FIFO can trigger receiver data ready interrupt.
typedef enum usim_rx_fifo_trigger_level usim_rx_fifo_trigger_level_t
    USIM Rx FIFO receiver trigger level enumeration.
SMARTCARD_Control(base, handle, control, param)
SMARTCARD_TransferNonBlocking(base, handle, xfer)
SMARTCARD_Init(base, handle, sourceClockHz)
SMARTCARD_Deinit(base)
SMARTCARD_GetTransferRemainingBytes(base, handle)
SMARTCARD_AbortTransfer(base, handle)
SMARTCARD_EMV_RX_NACK_THRESHOLD
    EMV RX NACK interrupt generation threshold.
SMARTCARD_EMV_TX_NACK_THRESHOLD
    EMV TX NACK interrupt generation threshold.
SMARTCARD_T0_CWT_ADJUSTMENT
    Smart card T0 Character Wait Timer adjustment value.
SMARTCARD_T1_CWT_ADJUSTMENT
    Smart card T1 Character Wait Timer adjustment value.
SMARTCARD_T0_BWT_ADJUSTMENT
    Smart card T0 Block Wait Timer adjustment value.
SMARTCARD_T1_BWT_ADJUSTMENT
    Smart card T1 Block Wait Timer adjustment value.
SMARTCARD_MAX_RX_TRIGGER_LEVEL
    Rx FIFO max receive trigger level.
USIM_FIND_RX_FIFO_TRIGGER_LEVEL(x)
    USIM Find max Rx FIFO receiver trigger level according to bytes numbers.

```

2.56 SPI: Serial Peripheral Interface Driver

2.57 SPI DMA Driver

```

status_t SPI_MasterTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t *handle,
                                           spi_dma_callback_t callback, void *userData,
                                           dma_handle_t *txHandle, dma_handle_t
                                           *rxHandle)

```

Initialize the SPI master DMA handle.

This function initializes the SPI master DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
status_t SPI_MasterTransferDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_transfer_t *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

```
status_t SPI_MasterHalfDuplexTransferDMA(SPI_Type *base, spi_dma_handle_t *handle, spi_half_duplex_transfer_t *xfer)
```

Transfers a block of data using a DMA method.

This function using polling way to do the first half transimission and using DMA way to do the srcond half transimission, the transfer mechanism is half-duplex. When do the second half transimission, code will return right away. When all data is transferred, the callback function is called.

Parameters

- base – SPI base pointer
- handle – A pointer to the spi_master_dma_handle_t structure which stores the transfer state.
- xfer – A pointer to the spi_half_duplex_transfer_t structure.

Returns

status of status_t.

```
static inline status_t SPI_SlaveTransferCreateHandleDMA(SPI_Type *base, spi_dma_handle_t
                                                         *handle, spi_dma_callback_t callback,
                                                         void *userData, dma_handle_t
                                                         *txHandle, dma_handle_t *rxHandle)
```

Initialize the SPI slave DMA handle.

This function initializes the SPI slave DMA handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, user need only call this API once to get the initialized handle.

Parameters

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – User callback function called at the end of a transfer.
- userData – User data for callback.
- txHandle – DMA handle pointer for SPI Tx, the handle shall be static allocated by users.
- rxHandle – DMA handle pointer for SPI Rx, the handle shall be static allocated by users.

```
static inline status_t SPI_SlaveTransferDMA(SPI_Type *base, spi_dma_handle_t *handle,
                                              spi_transfer_t *xfer)
```

Perform a non-blocking SPI transfer using DMA.

Note: This interface returned immediately after transfer initiates, users should call SPI_GetTransferStatus to poll the transfer status to check whether SPI transfer finished.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.
- xfer – Pointer to dma transfer structure.

Return values

- kStatus_Success – Successfully start a transfer.
- kStatus_InvalidArgument – Input argument is invalid.
- kStatus_SPI_Busy – SPI is not idle, is running another transfer.

```
void SPI_MasterTransferAbortDMA(SPI_Type *base, spi_dma_handle_t *handle)
```

Abort a SPI transfer using DMA.

Parameters

- base – SPI peripheral base address.
- handle – SPI DMA handle pointer.

```
status_t SPI_MasterTransferGetCountDMA(SPI_Type *base, spi_dma_handle_t *handle, size_t
                                         *count)
```

Gets the master DMA transfer remaining bytes.

This function gets the master DMA transfer remaining bytes.

Parameters

- base – SPI peripheral base address.

- `handle` – A pointer to the `spi_dma_handle_t` structure which stores the transfer state.
- `count` – A number of bytes transferred by the non-blocking transaction.

Returns

status of `status_t`.

static inline void SPI_SlaveTransferAbortDMA(SPI_Type *base, *spi_dma_handle_t* *handle)

Abort a SPI transfer using DMA.

Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI DMA handle pointer.

static inline *status_t* SPI_SlaveTransferGetCountDMA(SPI_Type *base, *spi_dma_handle_t* *handle, *size_t* *count)

Gets the slave DMA transfer remaining bytes.

This function gets the slave DMA transfer remaining bytes.

Parameters

- `base` – SPI peripheral base address.
- `handle` – A pointer to the `spi_dma_handle_t` structure which stores the transfer state.
- `count` – A number of bytes transferred by the non-blocking transaction.

Returns

status of `status_t`.

FSL_SPI_DMA_DRIVER_VERSION

SPI DMA driver version 2.1.1.

typedef struct *spi_dma_handle* spi_dma_handle_t

typedef void (*spi_dma_callback_t)(SPI_Type *base, *spi_dma_handle_t* *handle, *status_t* status, void *userData)

SPI DMA callback called at the end of transfer.

struct *_spi_dma_handle*

#include <fsl_spi_dma.h> SPI DMA transfer handle, users should not touch the content of the handle.

Public Members

volatile bool txInProgress

Send transfer finished

volatile bool rxInProgress

Receive transfer finished

uint8_t bytesPerFrame

Bytes in a frame for SPI transfer

uint8_t lastwordBytes

The Bytes of lastword for master

dma_handle_t *txHandle

DMA handler for SPI send

dma_handle_t *rxHandle
DMA handler for SPI receive

spi_dma_callback_t callback
Callback for SPI DMA transfer

void *userData
User Data for SPI DMA callback

uint32_t state
Internal state of SPI DMA transfer

size_t transferSize
Bytes need to be transfer

uint32_t instance
Index of SPI instance

const uint8_t *txNextData
The pointer of next time tx data

const uint8_t *txEndData
The pointer of end of data

uint8_t *rxNextData
The pointer of next time rx data

uint8_t *rxEndData
The pointer of end of rx data

uint32_t dataBytesEveryTime
Bytes in a time for DMA transfer, default is DMA_MAX_TRANSFER_COUNT

2.58 SPI Driver

FSL_SPI_DRIVER_VERSION

SPI driver version.

enum _spi_xfer_option

SPI transfer option.

Values:

enumerator kSPI_FrameDelay

A delay may be inserted, defined in the DLY register.

enumerator kSPI_FrameAssert

SSEL will be deasserted at the end of a transfer

enum _spi_shift_direction

SPI data shifter direction options.

Values:

enumerator kSPI_MsbFirst

Data transfers start with most significant bit.

enumerator kSPI_LsbFirst

Data transfers start with least significant bit.

enum `_spi_clock_polarity`

SPI clock polarity configuration.

Values:

enumerator `kSPI_ClockPolarityActiveHigh`

Active-high SPI clock (idles low).

enumerator `kSPI_ClockPolarityActiveLow`

Active-low SPI clock (idles high).

enum `_spi_clock_phase`

SPI clock phase configuration.

Values:

enumerator `kSPI_ClockPhaseFirstEdge`

First edge on SCK occurs at the middle of the first cycle of a data transfer.

enumerator `kSPI_ClockPhaseSecondEdge`

First edge on SCK occurs at the start of the first cycle of a data transfer.

enum `_spi_txfifo_watermark`

txFIFO watermark values

Values:

enumerator `kSPI_TxFifo0`

SPI tx watermark is empty

enumerator `kSPI_TxFifo1`

SPI tx watermark at 1 item

enumerator `kSPI_TxFifo2`

SPI tx watermark at 2 items

enumerator `kSPI_TxFifo3`

SPI tx watermark at 3 items

enumerator `kSPI_TxFifo4`

SPI tx watermark at 4 items

enumerator `kSPI_TxFifo5`

SPI tx watermark at 5 items

enumerator `kSPI_TxFifo6`

SPI tx watermark at 6 items

enumerator `kSPI_TxFifo7`

SPI tx watermark at 7 items

enum `_spi_rxfifo_watermark`

rxFIFO watermark values

Values:

enumerator `kSPI_RxFifo1`

SPI rx watermark at 1 item

enumerator `kSPI_RxFifo2`

SPI rx watermark at 2 items

enumerator `kSPI_RxFifo3`

SPI rx watermark at 3 items

enumerator kSPI_RxFifo4
SPI rx watermark at 4 items

enumerator kSPI_RxFifo5
SPI rx watermark at 5 items

enumerator kSPI_RxFifo6
SPI rx watermark at 6 items

enumerator kSPI_RxFifo7
SPI rx watermark at 7 items

enumerator kSPI_RxFifo8
SPI rx watermark at 8 items

enum _spi_data_width
Transfer data width.

Values:

enumerator kSPI_Data4Bits
4 bits data width

enumerator kSPI_Data5Bits
5 bits data width

enumerator kSPI_Data6Bits
6 bits data width

enumerator kSPI_Data7Bits
7 bits data width

enumerator kSPI_Data8Bits
8 bits data width

enumerator kSPI_Data9Bits
9 bits data width

enumerator kSPI_Data10Bits
10 bits data width

enumerator kSPI_Data11Bits
11 bits data width

enumerator kSPI_Data12Bits
12 bits data width

enumerator kSPI_Data13Bits
13 bits data width

enumerator kSPI_Data14Bits
14 bits data width

enumerator kSPI_Data15Bits
15 bits data width

enumerator kSPI_Data16Bits
16 bits data width

enum _spi_ssel
Slave select.

Values:

enumerator kSPI_Ssel0

Slave select 0

enumerator kSPI_Ssel1

Slave select 1

enumerator kSPI_Ssel2

Slave select 2

enumerator kSPI_Ssel3

Slave select 3

enum _spi_spol

ssel polarity

Values:

enumerator kSPI_Spol0ActiveHigh

enumerator kSPI_Spol1ActiveHigh

enumerator kSPI_Spol3ActiveHigh

enumerator kSPI_SpolActiveAllHigh

enumerator kSPI_SpolActiveAllLow

SPI transfer status.

Values:

enumerator kStatus_SPI_Busy

SPI bus is busy

enumerator kStatus_SPI_Idle

SPI is idle

enumerator kStatus_SPI_Error

SPI error

enumerator kStatus_SPI_BaudrateNotSupport

Baudrate is not support in current clock source

enumerator kStatus_SPI_Timeout

SPI timeout polling status flags.

enum _spi_interrupt_enable

SPI interrupt sources.

Values:

enumerator kSPI_RxLvlIrq

Rx level interrupt

enumerator kSPI_TxLvlIrq

Tx level interrupt

enum _spi_statusflags

SPI status flags.

Values:

enumerator kSPI_TxEmptyFlag

txFifo is empty

enumerator `kSPI_TxNotFullFlag`
txFifo is not full

enumerator `kSPI_RxNotEmptyFlag`
rxFIFO is not empty

enumerator `kSPI_RxFullFlag`
rxFIFO is full

typedef enum `_spi_xfer_option` `spi_xfer_option_t`
SPI transfer option.

typedef enum `_spi_shift_direction` `spi_shift_direction_t`
SPI data shifter direction options.

typedef enum `_spi_clock_polarity` `spi_clock_polarity_t`
SPI clock polarity configuration.

typedef enum `_spi_clock_phase` `spi_clock_phase_t`
SPI clock phase configuration.

typedef enum `_spi_txfifo_watermark` `spi_txfifo_watermark_t`
txFIFO watermark values

typedef enum `_spi_rxfifo_watermark` `spi_rxfifo_watermark_t`
rxFIFO watermark values

typedef enum `_spi_data_width` `spi_data_width_t`
Transfer data width.

typedef enum `_spi_ssel` `spi_ssel_t`
Slave select.

typedef enum `_spi_spol` `spi_spol_t`
ssel polarity

typedef struct `_spi_delay_config` `spi_delay_config_t`
SPI delay time configure structure. Note: The DLY register controls several programmable delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The maxinum value of these delay time is 15.

typedef struct `_spi_master_config` `spi_master_config_t`
SPI master user configure structure.

typedef struct `_spi_slave_config` `spi_slave_config_t`
SPI slave user configure structure.

typedef struct `_spi_transfer` `spi_transfer_t`
SPI transfer structure.

typedef struct `_spi_half_duplex_transfer` `spi_half_duplex_transfer_t`
SPI half-duplex(master only) transfer structure.

typedef struct `_spi_config` `spi_config_t`
Internal configuration structure used in 'spi' and 'spi_dma' driver.

typedef struct `_spi_master_handle` `spi_master_handle_t`
Master handle type.

typedef `spi_master_handle_t` `spi_slave_handle_t`
Slave handle type.

```
typedef void (*spi_master_callback_t)(SPI_Type *base, spi_master_handle_t *handle, status_t status, void *userData)
```

SPI master callback for finished transmit.

```
typedef void (*spi_slave_callback_t)(SPI_Type *base, spi_slave_handle_t *handle, status_t status, void *userData)
```

SPI slave callback for finished transmit.

```
typedef void (*flexcomm_spi_master_irq_handler_t)(SPI_Type *base, spi_master_handle_t *handle)
```

Typedef for master interrupt handler.

```
typedef void (*flexcomm_spi_slave_irq_handler_t)(SPI_Type *base, spi_slave_handle_t *handle)
```

Typedef for slave interrupt handler.

```
volatile uint8_t s_dummyData[]
```

SPI default SSEL COUNT.

Global variable for dummy data value setting.

```
SPI_DUMMYDATA
```

SPI dummy transfer data, the data is sent while txBuff is NULL.

```
SPI_RETRY_TIMES
```

Retry times for waiting flag.

```
SPI_DATA(n)
```

```
SPI_CTRLMASK
```

```
SPI_ASSERTNUM_SSEL(n)
```

```
SPI_DEASSERTNUM_SSEL(n)
```

```
SPI_DEASSERT_ALL
```

```
SPI_FIFOWR_FLAGS_MASK
```

```
SPI_FIFOTRIG_TXLVL_GET(base)
```

```
SPI_FIFOTRIG_RXLVL_GET(base)
```

```
struct __spi_delay_config
```

#include <fsl_spi.h> SPI delay time configure structure. Note: The DLY register controls several programmable delays related to SPI signalling, it stands for how many SPI clock time will be inserted. The maximum value of these delay time is 15.

Public Members

```
uint8_t preDelay
```

Delay between SSEL assertion and the beginning of transfer.

```
uint8_t postDelay
```

Delay between the end of transfer and SSEL deassertion.

```
uint8_t frameDelay
```

Delay between frame to frame.

```
uint8_t transferDelay
```

Delay between transfer to transfer.

```
struct __spi_master_config
```

#include <fsl_spi.h> SPI master user configure structure.

Public Members

`bool enableLoopback`
Enable loopback for test purpose

`bool enableMaster`
Enable SPI at initialization time

`spi_clock_polarity_t polarity`
Clock polarity

`spi_clock_phase_t phase`
Clock phase

`spi_shift_direction_t direction`
MSB or LSB

`uint32_t baudRate_Bps`
Baud Rate for SPI in Hz

`spi_data_width_t dataWidth`
Width of the data

`spi_ssel_t sselNum`
Slave select number

`spi_spol_t sselPol`
Configure active CS polarity

`uint8_t txWatermark`
txFIFO watermark

`uint8_t rxWatermark`
rxFIFO watermark

`spi_delay_config_t delayConfig`
Delay configuration.

`struct __spi_slave_config`
`#include <fsl_spi.h>` SPI slave user configure structure.

Public Members

`bool enableSlave`
Enable SPI at initialization time

`spi_clock_polarity_t polarity`
Clock polarity

`spi_clock_phase_t phase`
Clock phase

`spi_shift_direction_t direction`
MSB or LSB

`spi_data_width_t dataWidth`
Width of the data

`spi_spol_t sselPol`
Configure active CS polarity

```
uint8_t txWatermark
    txFIFO watermark
uint8_t rxWatermark
    rxFIFO watermark
struct __spi__transfer
    #include <fsl_spi.h> SPI transfer structure.
```

Public Members

```
const uint8_t *txData
    Send buffer
uint8_t *rxData
    Receive buffer
uint32_t configFlags
    Additional option to control transfer, spi_xfer_option_t.
size_t dataSize
    Transfer bytes
struct __spi__half_duplex_transfer
    #include <fsl_spi.h> SPI half-duplex(master only) transfer structure.
```

Public Members

```
const uint8_t *txData
    Send buffer
uint8_t *rxData
    Receive buffer
size_t txDataSize
    Transfer bytes for transmit
size_t rxDataSize
    Transfer bytes
uint32_t configFlags
    Transfer configuration flags, spi_xfer_option_t.
bool isPcsAssertInTransfer
    If PCS pin keep assert between transmit and receive. true for assert and false for de-assert.
bool isTransmitFirst
    True for transmit first and false for receive first.
struct __spi__config
    #include <fsl_spi.h> Internal configuration structure used in 'spi' and 'spi_dma' driver.
struct __spi__master_handle
    #include <fsl_spi.h> SPI transfer handle structure.
```

Public Members

const uint8_t *volatile txData

Transfer buffer

uint8_t *volatile rxData

Receive buffer

volatile size_t txRemainingBytes

Number of data to be transmitted [in bytes]

volatile size_t rxRemainingBytes

Number of data to be received [in bytes]

volatile int8_t toReceiveCount

The number of data expected to receive in data width. Since the received count and sent count should be the same to complete the transfer, if the sent count is x and the received count is y, toReceiveCount is x-y.

size_t totalByteCount

A number of transfer bytes

volatile uint32_t state

SPI internal state

spi_master_callback_t callback

SPI callback

void *userData

Callback parameter

uint8_t dataWidth

Width of the data [Valid values: 1 to 16]

uint8_t sselNum

Slave select number to be asserted when transferring data [Valid values: 0 to 3]

uint32_t configFlags

Additional option to control transfer

uint8_t txWatermark

txFIFO watermark

uint8_t rxWatermark

rxFIFO watermark

2.59 TRNG: True Random Number Generator

FSL_TRNG_DRIVER_VERSION

TRNG driver version 2.0.18.

Current version: 2.0.18

Change log:

- version 2.0.18
 - TRNG health checks now done in software on RT5xx and RT6xx.
- version 2.0.17
 - Added support for RT700.

- version 2.0.16
 - Added support for Dual oscillator mode.
- version 2.0.15
 - Changed `TRNG_USER_CONFIG_DEFAULT_XXX` values according to latest recommended by design team.
- version 2.0.14
 - add support for RW610 and RW612
- version 2.0.13
 - After deepsleep it might return error, added clearing bits in `TRNG_GetRandomData()` and generating new entropy.
 - Modified reloading entropy in `TRNG_GetRandomData()`, for some data length it doesn't reloading entropy correctly.
- version 2.0.12
 - For `KW34A4_SERIES`, `KW35A4_SERIES`, `KW36A4_SERIES` set `TRNG_USER_CONFIG_DEFAULT_OSC_DIV` to `kTRNG_RingOscDiv8`.
- version 2.0.11
 - Add clearing pending errors in `TRNG_Init()`.
- version 2.0.10
 - Fixed doxygen issues.
- version 2.0.9
 - Fix `HIS_CCM` metrics issues.
- version 2.0.8
 - For `K32L2A41A_SERIES` set `TRNG_USER_CONFIG_DEFAULT_OSC_DIV` to `kTRNG_RingOscDiv4`.
- version 2.0.7
 - Fix MISRA 2004 issue rule 12.5.
- version 2.0.6
 - For `KW35Z4_SERIES` set `TRNG_USER_CONFIG_DEFAULT_OSC_DIV` to `kTRNG_RingOscDiv8`.
- version 2.0.5
 - Add possibility to define default TRNG configuration by device specific preprocessor macros for `FRQMIN`, `FRQMAX` and `OSCDIV`.
- version 2.0.4
 - Fix MISRA-2012 issues.
- Version 2.0.3
 - update `TRNG_Init` to restart entropy generation
- Version 2.0.2
 - fix MISRA issues
- Version 2.0.1
 - add support for KL8x and KL28Z
 - update default `OSCDIV` for K81 to divide by 2

enum *_trng_sample_mode*

TRNG sample mode. Used by *trng_config_t*.

Values:

enumerator *kTRNG_SampleModeVonNeumann*

Use von Neumann data in both Entropy shifter and Statistical Checker.

enumerator *kTRNG_SampleModeRaw*

Use raw data into both Entropy shifter and Statistical Checker.

enumerator *kTRNG_SampleModeVonNeumannRaw*

Use von Neumann data in Entropy shifter. Use raw data into Statistical Checker.

enum *_trng_clock_mode*

TRNG clock mode. Used by *trng_config_t*.

Values:

enumerator *kTRNG_ClockModeRingOscillator*

Ring oscillator is used to operate the TRNG (default).

enumerator *kTRNG_ClockModeSystem*

System clock is used to operate the TRNG. This is for test use only, and indeterminate results may occur.

enum *_trng_ring_osc_div*

TRNG ring oscillator divide. Used by *trng_config_t*.

Values:

enumerator *kTRNG_RingOscDiv0*

Ring oscillator with no divide

enumerator *kTRNG_RingOscDiv2*

Ring oscillator divided-by-2.

enumerator *kTRNG_RingOscDiv4*

Ring oscillator divided-by-4.

enumerator *kTRNG_RingOscDiv8*

Ring oscillator divided-by-8.

enum *trng_oscillator_mode_t*

TRNG oscillator mode . Used by *trng_config_t*.

Values:

enumerator *kTRNG_SingleOscillatorModeOsc1*

Single oscillator mode, using OSC1 (default)

enumerator *kTRNG_DualOscillatorMode*

Dual oscillator mode

enumerator *kTRNG_SingleOscillatorModeOsc2*

Single oscillator mode, using OSC2

typedef enum *_trng_sample_mode* *trng_sample_mode_t*

TRNG sample mode. Used by *trng_config_t*.

typedef enum *_trng_clock_mode* *trng_clock_mode_t*

TRNG clock mode. Used by *trng_config_t*.

typedef enum *trng_ring_osc_div* trng_ring_osc_div_t

TRNG ring oscillator divide. Used by trng_config_t.

typedef enum *trng_oscillator_mode_t* trng_oscillator_mode_t

TRNG oscillator mode . Used by trng_config_t.

typedef struct *trng_statistical_check_limit* trng_statistical_check_limit_t

Data structure for definition of statistical check limits. Used by trng_config_t.

typedef struct *trng_user_config* trng_config_t

Data structure for the TRNG initialization.

This structure initializes the TRNG by calling the TRNG_Init() function. It contains all TRNG configurations.

status_t TRNG_GetDefaultConfig(*trng_config_t* *userConfig)

Initializes the user configuration structure to default values.

This function initializes the configuration structure to default values. The default values are platform dependent.

Parameters

- userConfig – User configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

status_t TRNG_Init(TRNG_Type *base, const *trng_config_t* *userConfig)

Initializes the TRNG.

This function initializes the TRNG. When called, the TRNG entropy generation starts immediately.

Parameters

- base – TRNG base address
- userConfig – Pointer to the initialization configuration structure.

Returns

If successful, returns the kStatus_TRNG_Success. Otherwise, it returns an error.

void TRNG_Deinit(TRNG_Type *base)

Shuts down the TRNG.

This function shuts down the TRNG.

Parameters

- base – TRNG base address.

status_t TRNG_GetRandomData(TRNG_Type *base, void *data, size_t dataSize)

Gets random data.

This function gets random data from the TRNG.

Parameters

- base – TRNG base address.
- data – Pointer address used to store random data.
- dataSize – Size of the buffer pointed by the data parameter.

Returns

random data

struct `_trng_statistical_check_limit`

#include <fsl_trng.h> Data structure for definition of statistical check limits. Used by `trng_config_t`.

Public Members

`uint32_t` maximum

Maximum limit.

`int32_t` minimum

Minimum limit.

struct `_trng_user_config`

#include <fsl_trng.h> Data structure for the TRNG initialization.

This structure initializes the TRNG by calling the `TRNG_Init()` function. It contains all TRNG configurations.

Public Members

`bool` lock

Disable programmability of TRNG registers.

`trng_clock_mode_t` clockMode

Clock mode used to operate TRNG.

`trng_ring_osc_div_t` ringOscDiv

Ring oscillator divide used by TRNG.

`trng_sample_mode_t` sampleMode

Sample mode of the TRNG ring oscillator.

`trng_oscillator_mode_t` oscillatorMode

TRNG oscillator mode .

`trng_ring_osc_div_t` ringOsc2Div

Divider used for Ring oscillator 2.

`uint16_t` entropyDelay

Entropy Delay. Defines the length (in system clocks) of each Entropy sample taken.

`uint16_t` sampleSize

Sample Size. Defines the total number of Entropy samples that will be taken during Entropy generation.

`uint16_t` sparseBitLimit

Sparse Bit Limit which defines the maximum number of consecutive samples that may be discarded before an error is generated. This limit is used only for during von Neumann sampling (enabled by `TRNG_HAL_SetSampleMode()`). Samples are discarded if two consecutive raw samples are both 0 or both 1. If this discarding occurs for a long period of time, it indicates that there is insufficient Entropy.

`uint8_t` retryCount

Retry count. It defines the number of times a statistical check may fails during the TRNG Entropy Generation before generating an error.

`uint8_t` longRunMaxLimit

Largest allowable number of consecutive samples of all 1, or all 0, that is allowed during the Entropy generation.

trng_statistical_check_limit_t monobitLimit

Maximum and minimum limits for statistical check of number of ones/zero detected during entropy generation.

trng_statistical_check_limit_t runBit1Limit

Maximum and minimum limits for statistical check of number of runs of length 1 detected during entropy generation.

trng_statistical_check_limit_t runBit2Limit

Maximum and minimum limits for statistical check of number of runs of length 2 detected during entropy generation.

trng_statistical_check_limit_t runBit3Limit

Maximum and minimum limits for statistical check of number of runs of length 3 detected during entropy generation.

trng_statistical_check_limit_t runBit4Limit

Maximum and minimum limits for statistical check of number of runs of length 4 detected during entropy generation.

trng_statistical_check_limit_t runBit5Limit

Maximum and minimum limits for statistical check of number of runs of length 5 detected during entropy generation.

trng_statistical_check_limit_t runBit6PlusLimit

Maximum and minimum limits for statistical check of number of runs of length 6 or more detected during entropy generation.

trng_statistical_check_limit_t pokerLimit

Maximum and minimum limits for statistical check of “Poker Test”.

trng_statistical_check_limit_t frequencyCountLimit

Maximum and minimum limits for statistical check of entropy sample frequency count.

2.60 USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver

2.61 USART DMA Driver

status_t USART_TransferCreateHandleDMA(USART_Type *base, *usart_dma_handle_t* *handle, *usart_dma_transfer_callback_t* callback, void *userData, *dma_handle_t* *txDmaHandle, *dma_handle_t* *rxDmaHandle)

Initializes the USART handle which is used in transactional functions.

Parameters

- base – USART peripheral base address.
- handle – Pointer to *usart_dma_handle_t* structure.
- callback – Callback function.
- userData – User data.
- txDmaHandle – User-requested DMA handle for TX DMA transfer.
- rxDmaHandle – User-requested DMA handle for RX DMA transfer.

```
status_t USART_TransferSendDMA(USART_Type *base, usart_dma_handle_t *handle,  
                               usart_transfer_t *xfer)
```

Sends data using DMA.

This function sends data using DMA. This is a non-blocking function, which returns right away. When all data is sent, the send callback function is called.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- xfer – USART DMA transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_USART_TxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
status_t USART_TransferReceiveDMA(USART_Type *base, usart_dma_handle_t *handle,  
                                  usart_transfer_t *xfer)
```

Receives data using DMA.

This function receives data using DMA. This is a non-blocking function, which returns right away. When all data is received, the receive callback function is called.

Parameters

- base – USART peripheral base address.
- handle – Pointer to `usart_dma_handle_t` structure.
- xfer – USART DMA transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – if succeed, others failed.
- `kStatus_USART_RxBusy` – Previous transfer on going.
- `kStatus_InvalidArgument` – Invalid argument.

```
void USART_TransferAbortSendDMA(USART_Type *base, usart_dma_handle_t *handle)  
Aborts the sent data using DMA.
```

This function aborts send data using DMA.

Parameters

- base – USART peripheral base address
- handle – Pointer to `usart_dma_handle_t` structure

```
void USART_TransferAbortReceiveDMA(USART_Type *base, usart_dma_handle_t *handle)  
Aborts the received data using DMA.
```

This function aborts the received data using DMA.

Parameters

- base – USART peripheral base address
- handle – Pointer to `usart_dma_handle_t` structure

```
status_t USART_TransferGetReceiveCountDMA(USART_Type *base, usart_dma_handle_t *handle,
                                           uint32_t *count)
```

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
status_t USART_TransferGetSendCountDMA(USART_Type *base, usart_dma_handle_t *handle,
                                        uint32_t *count)
```

Get the number of bytes that have been sent.

This function gets the number of bytes that have been sent.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Sent bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

```
FSL_USART_DMA_DRIVER_VERSION
```

USART dma driver version.

```
typedef struct _usart_dma_handle usart_dma_handle_t
```

```
typedef void (*usart_dma_transfer_callback_t)(USART_Type *base, usart_dma_handle_t *handle,
status_t status, void *userData)
```

UART transfer callback function.

```
struct _usart_dma_handle
```

```
#include <fsl_usart_dma.h> UART DMA handle.
```

Public Members

```
USART_Type *base
```

UART peripheral base address.

```
usart_dma_transfer_callback_t callback
```

Callback function.

```
void *userData
```

UART callback function parameter.

`size_t rxDataSizeAll`

Size of the data to receive.

`size_t txDataSizeAll`

Size of the data to send out.

`dma_handle_t *txDmaHandle`

The DMA TX channel used.

`dma_handle_t *rxDmaHandle`

The DMA RX channel used.

`volatile uint8_t txState`

TX transfer state.

`volatile uint8_t rxState`

RX transfer state

2.62 USART Driver

`status_t USART_Init(USART_Type *base, const usart_config_t *config, uint32_t srcClock_Hz)`

Initializes a USART instance with user configuration structure and peripheral clock.

This function configures the USART module with the user-defined settings. The user can configure the configuration structure and also get the default configuration by using the `USART_GetDefaultConfig()` function. Example below shows how to use this API to configure USART.

```
usart_config_t usartConfig;
usartConfig.baudRate_Bps = 115200U;
usartConfig.parityMode = kUSART_ParityDisabled;
usartConfig.stopBitCount = kUSART_OneStopBit;
USART_Init(USART1, &usartConfig, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `config` – Pointer to user-defined configuration structure.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_InvalidArgument` – USART base address is not valid
- `kStatus_Success` – Status USART initialize succeed

`void USART_CalcTimeoutConfig(uint32_t target_us, uint8_t *rxTimeoutPrescaler, uint32_t *rxTimeoutcounter, uint32_t srcClock_Hz)`

Calculate the USART instance RX timeout prescaler and counter.

This function for calculate the USART RXFIFO timeout config. This function is used to calculate suitable prescaler and counter for `target_us`.

```
usart_config_t config;
config.rxWatermark = kUSART_RxFifo2;
config.rxTimeout.enable = true;
```

(continues on next page)

(continued from previous page)

```
config.rxTimeout.resetCounterOnEmpty = true;
config.rxTimeout.resetCounterOnReceive = true;
USART_CalcTimeoutConfig(200U, &config.rxTimeout.prescaler, &config.rxTimeout.counter,
                        CLOCK_GetFreq(kCLOCK_BusClk));
```

Parameters

- `target_us` – Time for rx timeout unit us.
- `rxTimeoutPrescaler` – The prescaler to be setted after function.
- `rxTimeoutcounter` – The counter to be setted after function.
- `srcClock_Hz` – The clockSrc for rx timeout.

`void USART_SetRxTimeoutConfig(USART_Type *base, const usart_rx_timeout_config *config)`
Sets the USART instance RX timeout config.

This function configures the USART RXFIFO timeout config. This function is used to config the USART RXFIFO timeout config after the USART module is initialized by the USART_Init.

Parameters

- `base` – USART peripheral base address.
- `config` – pointer to receive timeout configuration structure.

`void USART_Deinit(USART_Type *base)`
Deinitializes a USART instance.

This function waits for TX complete, disables TX and RX, and disables the USART clock.

Parameters

- `base` – USART peripheral base address.

`void USART_GetDefaultConfig(usart_config_t *config)`
Gets the default configuration structure.

This function initializes the USART configuration structure to a default value. The default values are: `usartConfig->baudRate_Bps = 115200U`; `usartConfig->parityMode = kUSART_ParityDisabled`; `usartConfig->stopBitCount = kUSART_OneStopBit`; `usartConfig->bitCountPerChar = kUSART_8BitsPerChar`; `usartConfig->loopback = false`; `usartConfig->enableTx = false`; `usartConfig->enableRx = false`;

Parameters

- `config` – Pointer to configuration structure.

`status_t USART_SetBaudRate(USART_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)`
Sets the USART instance baud rate.

This function configures the USART module baud rate. This function is used to update the USART module baud rate after the USART module is initialized by the USART_Init.

```
USART_SetBaudRate(USART1, 115200U, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `baudrate_Bps` – USART baudrate to be set.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

`status_t` `USART_Enable32kMode(USART_Type *base, uint32_t baudRate_Bps, bool enableMode32k, uint32_t srcClock_Hz)`

Enable 32 kHz mode which USART uses clock from the RTC oscillator as the clock source.

Please note that in order to use a 32 kHz clock to operate USART properly, the RTC oscillator and its 32 kHz output must be manually enabled by user, by calling `RTC_Init` and setting `SYSCON_RTCOSCCTRL_EN` bit to 1. And in 32kHz clocking mode the USART can only work at 9600 baudrate or at the baudrate that 9600 can evenly divide, eg: 4800, 3200.

Parameters

- `base` – USART peripheral base address.
- `baudRate_Bps` – USART baudrate to be set..
- `enableMode32k` – true is 32k mode, false is normal mode.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

`void` `USART_Enable9bitMode(USART_Type *base, bool enable)`

Enable 9-bit data mode for USART.

This function set the 9-bit mode for USART module. The 9th bit is not used for parity thus can be modified by user.

Parameters

- `base` – USART peripheral base address.
- `enable` – true to enable, false to disable.

`static inline void` `USART_SetMatchAddress(USART_Type *base, uint8_t address)`

Set the USART slave address.

This function configures the address for USART module that works as slave in 9-bit data mode. When the address detection is enabled, the frame it receives with MSB being 1 is considered as an address frame, otherwise it is considered as data frame. Once the address frame matches slave's own addresses, this slave is addressed. This address frame and its following data frames are stored in the receive buffer, otherwise the frames will be discarded. To un-address a slave, just send an address frame with unmatched address.

Note: Any USART instance joined in the multi-slave system can work as slave. The position of the address mark is the same as the parity bit when parity is enabled for 8 bit and 9 bit data formats.

Parameters

- `base` – USART peripheral base address.
- `address` – USART slave address.

static inline void USART__EnableMatchAddress(USART_Type *base, bool match)

Enable the USART match address feature.

Parameters

- base – USART peripheral base address.
- match – true to enable match address, false to disable.

static inline uint32_t USART__GetStatusFlags(USART_Type *base)

Get USART status flags.

This function get all USART status flags, the flags are returned as the logical OR value of the enumerators `_usart_flags`. To check a specific status, compare the return value with enumerators in `_usart_flags`. For example, to check whether the TX is empty:

```
if (kUSART_TxFifoNotFullFlag & USART__GetStatusFlags(USART1))
{
    ...
}
```

Parameters

- base – USART peripheral base address.

Returns

USART status flags which are ORed by the enumerators in the `_usart_flags`.

static inline void USART__ClearStatusFlags(USART_Type *base, uint32_t mask)

Clear USART status flags.

This function clear supported USART status flags. The mask is a logical OR of enumeration members. See `kUSART_AllClearFlags`. For example:

```
USART__ClearStatusFlags(USART1, kUSART_TxError | kUSART_RxError)
```

Parameters

- base – USART peripheral base address.
- mask – status flags to be cleared.

static inline void USART__EnableInterrupts(USART_Type *base, uint32_t mask)

Enables USART interrupts according to the provided mask.

This function enables the USART interrupts according to the provided mask. The mask is a logical OR of enumeration members. See `_usart_interrupt_enable`. For example, to enable TX empty interrupt and RX full interrupt:

```
USART__EnableInterrupts(USART1, kUSART_TxLevelInterruptEnable | kUSART__
↳ RxLevelInterruptEnable);
```

Parameters

- base – USART peripheral base address.
- mask – The interrupts to enable. Logical OR of `_usart_interrupt_enable`.

static inline void USART__DisableInterrupts(USART_Type *base, uint32_t mask)

Disables USART interrupts according to a provided mask.

This function disables the USART interrupts according to a provided mask. The mask is a logical OR of enumeration members. See `_usart_interrupt_enable`. This example shows how to disable the TX empty interrupt and RX full interrupt:

```
USART_DisableInterrupts(USART1, kUSART_TxLevelInterruptEnable | kUSART_
↳ RxLevelInterruptEnable);
```

Parameters

- base – USART peripheral base address.
- mask – The interrupts to disable. Logical OR of _usart_interrupt_enable.

static inline uint32_t USART_GetEnabledInterrupts(USART_Type *base)

Returns enabled USART interrupts.

This function returns the enabled USART interrupts.

Parameters

- base – USART peripheral base address.

static inline void USART_EnableTxDMA(USART_Type *base, bool enable)

Enable DMA for Tx.

static inline void USART_EnableRxDMA(USART_Type *base, bool enable)

Enable DMA for Rx.

static inline void USART_EnableCTS(USART_Type *base, bool enable)

Enable CTS. This function will determine whether CTS is used for flow control.

Parameters

- base – USART peripheral base address.
- enable – Enable CTS or not, true for enable and false for disable.

static inline void USART_EnableContinuousSCLK(USART_Type *base, bool enable)

Continuous Clock generation. By default, SCLK is only output while data is being transmitted in synchronous mode. Enable this function, SCLK will run continuously in synchronous mode, allowing characters to be received on Un_RxD independently from transmission on Un_TXD).

Parameters

- base – USART peripheral base address.
- enable – Enable Continuous Clock generation mode or not, true for enable and false for disable.

static inline void USART_EnableAutoClearSCLK(USART_Type *base, bool enable)

Enable Continuous Clock generation bit auto clear. While enable this function, the Continuous Clock bit is automatically cleared when a complete character has been received. This bit is cleared at the same time.

Parameters

- base – USART peripheral base address.
- enable – Enable auto clear or not, true for enable and false for disable.

static inline void USART_SetRxFifoWatermark(USART_Type *base, uint8_t water)

Sets the rx FIFO watermark.

Parameters

- base – USART peripheral base address.
- water – Rx FIFO watermark.

```
static inline void USART_SetTxFifoWatermark(USART_Type *base, uint8_t water)
```

Sets the tx FIFO watermark.

Parameters

- base – USART peripheral base address.
- water – Tx FIFO watermark.

```
static inline void USART_WriteByte(USART_Type *base, uint8_t data)
```

Writes to the FIFOWR register.

This function writes data to the txFIFO directly. The upper layer must ensure that txFIFO has space for data to write before calling this function.

Parameters

- base – USART peripheral base address.
- data – The byte to write.

```
static inline uint8_t USART_ReadByte(USART_Type *base)
```

Reads the FIFORD register directly.

This function reads data from the rxFIFO directly. The upper layer must ensure that the rxFIFO is not empty before calling this function.

Parameters

- base – USART peripheral base address.

Returns

The byte read from USART data register.

```
static inline uint8_t USART_GetRxFifoCount(USART_Type *base)
```

Gets the rx FIFO data count.

Parameters

- base – USART peripheral base address.

Returns

rx FIFO data count.

```
static inline uint8_t USART_GetTxFifoCount(USART_Type *base)
```

Gets the tx FIFO data count.

Parameters

- base – USART peripheral base address.

Returns

tx FIFO data count.

```
void USART_SendAddress(USART_Type *base, uint8_t address)
```

Transmit an address frame in 9-bit data mode.

Parameters

- base – USART peripheral base address.
- address – USART slave address.

```
status_t USART_WriteBlocking(USART_Type *base, const uint8_t *data, size_t length)
```

Writes to the TX register using a blocking method.

This function polls the TX register, waits for the TX register to be empty or for the TX FIFO to have room and writes data to the TX buffer.

Parameters

- `base` – USART peripheral base address.
- `data` – Start address of the data to write.
- `length` – Size of the data to write.

Return values

- `kStatus_USART_Timeout` – Transmission timed out and was aborted.
- `kStatus_InvalidArgument` – Invalid argument.
- `kStatus_Success` – Successfully wrote all data.

status_t USART_ReadBlocking(USART_Type *base, uint8_t *data, size_t length)

Read RX data register using a blocking method.

This function polls the RX register, waits for the RX register to be full or for RX FIFO to have data and read data from the TX register.

Parameters

- `base` – USART peripheral base address.
- `data` – Start address of the buffer to store the received data.
- `length` – Size of the buffer.

Return values

- `kStatus_USART_FramingError` – Receiver overrun happened while receiving data.
- `kStatus_USART_ParityError` – Noise error happened while receiving data.
- `kStatus_USART_NoiseError` – Framing error happened while receiving data.
- `kStatus_USART_RxError` – Overflow or underflow rxFIFO happened.
- `kStatus_USART_Timeout` – Transmission timed out and was aborted.
- `kStatus_Success` – Successfully received all data.

status_t USART_TransferCreateHandle(USART_Type *base, usart_handle_t *handle, usart_transfer_callback_t callback, void *userData)

Initializes the USART handle.

This function initializes the USART handle which can be used for other USART transactional APIs. Usually, for a specified USART instance, call this API once to get the initialized handle.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.
- `callback` – The callback function.
- `userData` – The parameter of the callback function.

status_t USART_TransferSendNonBlocking(USART_Type *base, usart_handle_t *handle, usart_transfer_t *xfer)

Transmits a buffer of data using the interrupt method.

This function sends data using an interrupt method. This is a non-blocking function, which returns directly without waiting for all data to be written to the TX register. When all data is written to the TX register in the IRQ handler, the USART driver calls the callback function and passes the `kStatus_USART_TxIdle` as status parameter.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- xfer – USART transfer structure. See `usart_transfer_t`.

Return values

- `kStatus_Success` – Successfully start the data transmission.
- `kStatus_USART_TxBusy` – Previous transmission still not finished, data not all written to TX register yet.
- `kStatus_InvalidArgument` – Invalid argument.

```
void USART_TransferStartRingBuffer(USART_Type *base, usart_handle_t *handle, uint8_t  
                                *ringBuffer, size_t ringBufferSize)
```

Sets up the RX ring buffer.

This function sets up the RX ring buffer to a specific USART handle.

When the RX ring buffer is used, data received are stored into the ring buffer even when the user doesn't call the `USART_TransferReceiveNonBlocking()` API. If there is already data received in the ring buffer, the user can get the received data from the ring buffer directly.

Note: When using the RX ring buffer, one byte is reserved for internal use. In other words, if `ringBufferSize` is 32, then only 31 bytes are used for saving data.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- ringBuffer – Start address of the ring buffer for background receiving. Pass `NULL` to disable the ring buffer.
- ringBufferSize – size of the ring buffer.

```
void USART_TransferStopRingBuffer(USART_Type *base, usart_handle_t *handle)
```

Aborts the background transfer and uninstalls the ring buffer.

This function aborts the background transfer and uninstalls the ring buffer.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.

```
size_t USART_TransferGetRxRingBufferLength(usart_handle_t *handle)
```

Get the length of received data in RX ring buffer.

Parameters

- handle – USART handle pointer.

Returns

Length of received data in RX ring buffer.

```
void USART_TransferAbortSend(USART_Type *base, usart_handle_t *handle)
```

Aborts the interrupt-driven data transmit.

This function aborts the interrupt driven data sending. The user can get the `remainBtyes` to find out how many bytes are still not sent out.

Parameters

- base – USART peripheral base address.

- *handle* – USART handle pointer.

status_t USART_TransferGetSendCount(USART_Type **base*, *usart_handle_t* **handle*, uint32_t **count*)

Get the number of bytes that have been sent out to bus.

This function gets the number of bytes that have been sent out to bus by interrupt method.

Parameters

- *base* – USART peripheral base address.
- *handle* – USART handle pointer.
- *count* – Send bytes count.

Return values

- *kStatus_NoTransferInProgress* – No send in progress.
- *kStatus_InvalidArgument* – Parameter is invalid.
- *kStatus_Success* – Get successfully through the parameter *count*;

status_t USART_TransferReceiveNonBlocking(USART_Type **base*, *usart_handle_t* **handle*, *usart_transfer_t* **xfer*, size_t **receivedBytes*)

Receives a buffer of data using an interrupt method.

This function receives data using an interrupt method. This is a non-blocking function, which returns without waiting for all data to be received. If the RX ring buffer is used and not empty, the data in the ring buffer is copied and the parameter *receivedBytes* shows how many bytes are copied from the ring buffer. After copying, if the data in the ring buffer is not enough to read, the receive request is saved by the USART driver. When the new data arrives, the receive request is serviced first. When all data is received, the USART driver notifies the upper layer through a callback function and passes the status parameter *kStatus_USART_RxIdle*. For example, the upper layer needs 10 bytes but there are only 5 bytes in the ring buffer. The 5 bytes are copied to the *xfer->data* and this function returns with the parameter *receivedBytes* set to 5. For the left 5 bytes, newly arrived data is saved from the *xfer->data[5]*. When 5 bytes are received, the USART driver notifies the upper layer. If the RX ring buffer is not enabled, this function enables the RX and RX interrupt to receive data to the *xfer->data*. When all data is received, the upper layer is notified.

Parameters

- *base* – USART peripheral base address.
- *handle* – USART handle pointer.
- *xfer* – USART transfer structure, see *usart_transfer_t*.
- *receivedBytes* – Bytes received from the ring buffer directly.

Return values

- *kStatus_Success* – Successfully queue the transfer into transmit queue.
- *kStatus_USART_RxBusy* – Previous receive request is not finished.
- *kStatus_InvalidArgument* – Invalid argument.

void USART_TransferAbortReceive(USART_Type **base*, *usart_handle_t* **handle*)

Aborts the interrupt-driven data receiving.

This function aborts the interrupt-driven data receiving. The user can get the *remainBytes* to find out how many bytes not received yet.

Parameters

- *base* – USART peripheral base address.

- handle – USART handle pointer.

`status_t` USART_TransferGetReceiveCount(USART_Type *base, *usart_handle_t* *handle, uint32_t *count)

Get the number of bytes that have been received.

This function gets the number of bytes that have been received.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.
- count – Receive bytes count.

Return values

- kStatus_NoTransferInProgress – No receive in progress.
- kStatus_InvalidArgument – Parameter is invalid.
- kStatus_Success – Get successfully through the parameter count;

`void` USART_TransferHandleIRQ(USART_Type *base, *usart_handle_t* *handle)

USART IRQ handle function.

This function handles the USART transmit and receive IRQ request.

Parameters

- base – USART peripheral base address.
- handle – USART handle pointer.

FSL_USART_DRIVER_VERSION

USART driver version.

Error codes for the USART driver.

Values:

enumerator kStatus_USART_TxBusy

Transmitter is busy.

enumerator kStatus_USART_RxBusy

Receiver is busy.

enumerator kStatus_USART_TxIdle

USART transmitter is idle.

enumerator kStatus_USART_RxIdle

USART receiver is idle.

enumerator kStatus_USART_TxError

Error happens on txFIFO.

enumerator kStatus_USART_RxError

Error happens on rxFIFO.

enumerator kStatus_USART_RxRingBufferOverrun

Error happens on rx ring buffer

enumerator kStatus_USART_NoiseError

USART noise error.

enumerator kStatus_USART_FramingError
USART framing error.

enumerator kStatus_USART_ParityError
USART parity error.

enumerator kStatus_USART_BaudrateNotSupport
Baudrate is not support in current clock source

enum __usart_sync_mode
USART synchronous mode.

Values:

enumerator kUSART_SyncModeDisabled
Asynchronous mode.

enumerator kUSART_SyncModeSlave
Synchronous slave mode.

enumerator kUSART_SyncModeMaster
Synchronous master mode.

enum __usart_parity_mode
USART parity mode.

Values:

enumerator kUSART_ParityDisabled
Parity disabled

enumerator kUSART_ParityEven
Parity enabled, type even, bit setting: PE|PT = 10

enumerator kUSART_ParityOdd
Parity enabled, type odd, bit setting: PE|PT = 11

enum __usart_stop_bit_count
USART stop bit count.

Values:

enumerator kUSART_OneStopBit
One stop bit

enumerator kUSART_TwoStopBit
Two stop bits

enum __usart_data_len
USART data size.

Values:

enumerator kUSART_7BitsPerChar
Seven bit mode

enumerator kUSART_8BitsPerChar
Eight bit mode

enum __usart_clock_polarity
USART clock polarity configuration, used in sync mode.

Values:

enumerator kUSART_RxSampleOnFallingEdge
Un_RXD is sampled on the falling edge of SCLK.

enumerator kUSART_RxSampleOnRisingEdge
Un_RXD is sampled on the rising edge of SCLK.

enum _usart_txfifo_watermark
txFIFO watermark values

Values:

enumerator kUSART_TxFifo0
USART tx watermark is empty

enumerator kUSART_TxFifo1
USART tx watermark at 1 item

enumerator kUSART_TxFifo2
USART tx watermark at 2 items

enumerator kUSART_TxFifo3
USART tx watermark at 3 items

enumerator kUSART_TxFifo4
USART tx watermark at 4 items

enumerator kUSART_TxFifo5
USART tx watermark at 5 items

enumerator kUSART_TxFifo6
USART tx watermark at 6 items

enumerator kUSART_TxFifo7
USART tx watermark at 7 items

enum _usart_rxfifo_watermark
rxFIFO watermark values

Values:

enumerator kUSART_RxFifo1
USART rx watermark at 1 item

enumerator kUSART_RxFifo2
USART rx watermark at 2 items

enumerator kUSART_RxFifo3
USART rx watermark at 3 items

enumerator kUSART_RxFifo4
USART rx watermark at 4 items

enumerator kUSART_RxFifo5
USART rx watermark at 5 items

enumerator kUSART_RxFifo6
USART rx watermark at 6 items

enumerator kUSART_RxFifo7
USART rx watermark at 7 items

enumerator kUSART_RxFifo8
USART rx watermark at 8 items

enum `_usart_interrupt_enable`

USART interrupt configuration structure, default settings all disabled.

Values:

enumerator `kUSART_TxErrorInterruptEnable`

enumerator `kUSART_RxErrorInterruptEnable`

enumerator `kUSART_TxLevelInterruptEnable`

enumerator `kUSART_RxLevelInterruptEnable`

enumerator `kUSART_TxIdleInterruptEnable`

Transmitter idle.

enumerator `kUSART_CtsChangeInterruptEnable`

Change in the state of the CTS input.

enumerator `kUSART_RxBreakChangeInterruptEnable`

Break condition asserted or deasserted.

enumerator `kUSART_RxStartInterruptEnable`

Rx start bit detected.

enumerator `kUSART_FramingErrorInterruptEnable`

Framing error detected.

enumerator `kUSART_ParityErrorInterruptEnable`

Parity error detected.

enumerator `kUSART_NoiseErrorInterruptEnable`

Noise error detected.

enumerator `kUSART_AutoBaudErrorInterruptEnable`

Auto baudrate error detected.

enumerator `kUSART_RxTimeoutInterruptEnable`

Receive timeout detected.

enumerator `kUSART_AllInterruptEnables`

enum `_usart_flags`

USART status flags.

This provides constants for the USART status flags for use in the USART functions.

Values:

enumerator `kUSART_TxError`

TXERR bit, sets if TX buffer is error

enumerator `kUSART_RxError`

RXERR bit, sets if RX buffer is error

enumerator `kUSART_TxFifoEmptyFlag`

TXEMPTY bit, sets if TX buffer is empty

enumerator `kUSART_TxFifoNotFullFlag`

TXNOTFULL bit, sets if TX buffer is not full

enumerator `kUSART_RxFifoNotEmptyFlag`

RXNOEMPTY bit, sets if RX buffer is not empty

enumerator kUSART_RxFifoFullFlag

RXFULL bit, sets if RX buffer is full

enumerator kUSART_RxIdleFlag

Receiver idle.

enumerator kUSART_TxIdleFlag

Transmitter idle.

enumerator kUSART_CtsAssertFlag

CTS signal high.

enumerator kUSART_CtsChangeFlag

CTS signal changed interrupt status.

enumerator kUSART_BreakDetectFlag

Break detected. Self cleared when rx pin goes high again.

enumerator kUSART_BreakDetectChangeFlag

Break detect change interrupt flag. A change in the state of receiver break detection.

enumerator kUSART_RxStartFlag

Rx start bit detected interrupt flag.

enumerator kUSART_FramingErrorFlag

Framing error interrupt flag.

enumerator kUSART_ParityErrorFlag

parity error interrupt flag.

enumerator kUSART_NoiseErrorFlag

Noise error interrupt flag.

enumerator kUSART_AutobaudErrorFlag

Auto baudrate error interrupt flag, caused by the baudrate counter timeout before the end of start bit.

enumerator kUSART_RxTimeoutFlag

RXTIMEOUT bit, sets if RX FIFO Timeout.

enumerator kUSART_AllClearFlags

typedef enum *_usart_sync_mode* usart_sync_mode_t

USART synchronous mode.

typedef enum *_usart_parity_mode* usart_parity_mode_t

USART parity mode.

typedef enum *_usart_stop_bit_count* usart_stop_bit_count_t

USART stop bit count.

typedef enum *_usart_data_len* usart_data_len_t

USART data size.

typedef enum *_usart_clock_polarity* usart_clock_polarity_t

USART clock polarity configuration, used in sync mode.

typedef enum *_usart_txfifo_watermark* usart_txfifo_watermark_t

txFIFO watermark values

typedef enum *_usart_rxfifo_watermark* usart_rxfifo_watermark_t

rxFIFO watermark values

```
typedef struct _usart_rx_timeout_config usart_rx_timeout_config
```

USART receive timeout configuration structure.

```
typedef struct _usart_config usart_config_t
```

USART configuration structure.

```
typedef struct _usart_transfer usart_transfer_t
```

USART transfer structure.

```
typedef struct _usart_handle usart_handle_t
```

```
typedef void (*usart_transfer_callback_t)(USART_Type *base, usart_handle_t *handle, status_t status, void *userData)
```

USART transfer callback function.

```
typedef void (*flexcomm_usart_irq_handler_t)(USART_Type *base, usart_handle_t *handle)
```

Typedef for usart interrupt handler.

```
uint32_t USART_GetInstance(USART_Type *base)
```

Returns instance number for USART peripheral base address.

```
USART_FIFOTRIG_TXLVL_GET(base)
```

```
USART_FIFOTRIG_RXLVL_GET(base)
```

```
UART_RETRY_TIMES
```

Retry times for waiting flag.

Defining to zero means to keep waiting for the flag until it is assert/deassert in blocking transfer; otherwise the program will wait until the UART_RETRY_TIMES counts down to 0, if the flag still remains unchanged then program will return kStatus_USART_Timeout. It is not advised to use this macro in formal application to prevent any hardware error because the actual wait period is affected by the compiler and optimization.

```
struct _usart_rx_timeout_config
```

#include <fsl_usart.h> USART receive timeout configuration structure.

Public Members

```
bool enable
```

Enable RX timeout

```
bool resetCounterOnEmpty
```

Enable RX timeout counter reset when RX FIFO becomes empty.

```
bool resetCounterOnReceive
```

Enable RX timeout counter reset when RX FIFO receives data from the transmitter side.

```
uint32_t counter
```

RX timeout counter

```
uint8_t prescaler
```

RX timeout prescaler

```
struct _usart_config
```

#include <fsl_usart.h> USART configuration structure.

Public Members

uint32_t baudRate_Bps

USART baud rate

usart_parity_mode_t parityMode

Parity mode, disabled (default), even, odd

usart_stop_bit_count_t stopBitCount

Number of stop bits, 1 stop bit (default) or 2 stop bits

usart_data_len_t bitCountPerChar

Data length - 7 bit, 8 bit

bool loopback

Enable peripheral loopback

bool enableRx

Enable RX

bool enableTx

Enable TX

bool enableContinuousSCLK

USART continuous Clock generation enable in synchronous master mode.

bool enableMode32k

USART uses 32 kHz clock from the RTC oscillator as the clock source.

bool enableHardwareFlowControl

Enable hardware control RTS/CTS

usart_txfifo_watermark_t txWatermark

txFIFO watermark

usart_rxfifo_watermark_t rxWatermark

rxFIFO watermark

usart_sync_mode_t syncMode

Transfer mode select - asynchronous, synchronous master, synchronous slave.

usart_clock_polarity_t clockPolarity

Selects the clock polarity and sampling edge in synchronous mode.

usart_rx_timeout_config rxTimeout

rx timeout configuration

struct __usart_transfer

#include <fsl_usart.h> USART transfer structure.

Public Members

size_t dataSize

The byte count to be transfer.

struct __usart_handle

#include <fsl_usart.h> USART handle structure.

Public Members

`const uint8_t *volatile txData`
Address of remaining data to send.

`volatile size_t txDataSize`
Size of the remaining data to send.

`size_t txDataSizeAll`
Size of the data to send out.

`uint8_t *volatile rxData`
Address of remaining data to receive.

`volatile size_t rxDataSize`
Size of the remaining data to receive.

`size_t rxDataSizeAll`
Size of the data to receive.

`uint8_t *rxRingBuffer`
Start address of the receiver ring buffer.

`size_t rxRingBufferSize`
Size of the ring buffer.

`volatile uint16_t rxRingBufferHead`
Index for the driver to store received data into ring buffer.

`volatile uint16_t rxRingBufferTail`
Index for the user to get data from the ring buffer.

`usart_transfer_callback_t callback`
Callback function.

`void *userData`
USART callback function parameter.

`volatile uint8_t txState`
TX transfer state.

`volatile uint8_t rxState`
RX transfer state

`uint8_t txWatermark`
txFIFO watermark

`uint8_t rxWatermark`
rxFIFO watermark

`union __unnamed50__`

Public Members

`uint8_t *data`
The buffer of data to be transfer.

`uint8_t *rxData`
The buffer to receive data.

`const uint8_t *txData`
The buffer of data to be sent.

2.63 UTICK: Mictotick Timer Driver

`void UTICK_Init(UTICK_Type *base)`

Initializes an UTICK by turning its bus clock on.

`void UTICK_Deinit(UTICK_Type *base)`

Deinitializes a UTICK instance.

This function shuts down Utick bus clock

Parameters

- `base` – UTICK peripheral base address.

`uint32_t UTICK_GetStatusFlags(UTICK_Type *base)`

Get Status Flags.

This returns the status flag

Parameters

- `base` – UTICK peripheral base address.

Returns

status register value

`void UTICK_ClearStatusFlags(UTICK_Type *base)`

Clear Status Interrupt Flags.

This clears intr status flag

Parameters

- `base` – UTICK peripheral base address.

Returns

none

`void UTICK_SetTick(UTICK_Type *base, utick_mode_t mode, uint32_t count, utick_callback_t cb)`

Starts UTICK.

This function starts a repeat/onetime countdown with an optional callback

Parameters

- `base` – UTICK peripheral base address.
- `mode` – UTICK timer mode (ie `kUTICK_onetime` or `kUTICK_repeat`)
- `count` – UTICK timer mode (ie `kUTICK_onetime` or `kUTICK_repeat`)
- `cb` – UTICK callback (can be left as `NULL` if none, otherwise should be a `void func(void)`)

Returns

none

`void UTICK_HandleIRQ(UTICK_Type *base, utick_callback_t cb)`

UTICK Interrupt Service Handler.

This function handles the interrupt and refers to the callback array in the driver to callback user (as per request in `UTICK_SetTick()`). if no user callback is scheduled, the interrupt will simply be cleared.

Parameters

- `base` – UTICK peripheral base address.

- `cb` – callback scheduled for this instance of UTICK

Returns

none

FSL_UTICK_DRIVER_VERSION

UTICK driver version 2.0.5.

enum `_utick_mode`

UTICK timer operational mode.

*Values:*enumerator `kUTICK_Onetime`

Trigger once

enumerator `kUTICK_Repeat`

Trigger repeatedly

typedef enum `_utick_mode` `utick_mode_t`

UTICK timer operational mode.

typedef void (`*utick_callback_t`)(void)

UTICK callback function.

2.64 WWDT: Windowed Watchdog Timer Driver

void WWDT_GetDefaultConfig(`wwdt_config_t` *config)

Initializes WWDT configure structure.

This function initializes the WWDT configure structure to default value. The default value are:

```
config->enableWwdt = true;
config->enableWatchdogReset = false;
config->enableWatchdogProtect = false;
config->enableLockOscillator = false;
config->windowValue = 0xFFFFF0U;
config->timeoutValue = 0xFFFFF0U;
config->warningValue = 0;
```

See also:`wwdt_config_t`**Parameters**

- `config` – Pointer to WWDT config structure.

void WWDT_Init(WWDT_Type *base, const `wwdt_config_t` *config)

Initializes the WWDT.

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
wwdt_config_t config;
WWDT_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
WWDT_Init(wwdt_base,&config);
```

Parameters

- base – WWDT peripheral base address
- config – The configuration of WWDT

void WWDT_Deinit(WWDT_Type *base)

Shuts down the WWDT.

This function shuts down the WWDT.

Parameters

- base – WWDT peripheral base address

static inline void WWDT_Enable(WWDT_Type *base)

Enables the WWDT module.

This function write value into WWDT_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

- base – WWDT peripheral base address

static inline void WWDT_Disable(WWDT_Type *base)

Disables the WWDT module.

Deprecated:

Do not use this function. It will be deleted in next release version, for once the bit field of WDEN written with a 1, it can not be re-written with a 0.

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

- base – WWDT peripheral base address

static inline uint32_t WWDT_GetStatusFlags(WWDT_Type *base)

Gets all WWDT status flags.

This function gets all status flags.

Example for getting Timeout Flag:

```
uint32_t status;
status = WWDT_GetStatusFlags(wwdt_base) & kWWDT_TimeoutFlag;
```

Parameters

- base – WWDT peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `_wwdt_status_flags_t`

void WWDT_ClearStatusFlags(WWDT_Type *base, uint32_t mask)

Clear WWDT flag.

This function clears WWDT status flag.

Example for clearing warning flag:

```
WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
```

Parameters

- `base` – WWDT peripheral base address
- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `_wwdt_status_flags_t`

`static inline void WWDT_SetWarningValue(WWDT_Type *base, uint32_t warningValue)`

Set the WWDT warning value.

The `WDWARNINT` register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by `WARNINT`, an interrupt will be generated after the subsequent `WDCLK`.

Parameters

- `base` – WWDT peripheral base address
- `warningValue` – WWDT warning value.

`static inline void WWDT_SetTimeoutValue(WWDT_Type *base, uint32_t timeoutCount)`

Set the WWDT timeout value.

This function sets the timeout value. Every time a feed sequence occurs the value in the `TC` register is loaded into the Watchdog timer. Writing a value below `0xFF` will cause `0xFF` to be loaded into the `TC` register. Thus the minimum time-out interval is $TWDCLK * 256 * 4$. If `enableWatchdogProtect` flag is true in `wwdt_config_t` config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the `WDTOF` flag.

Parameters

- `base` – WWDT peripheral base address
- `timeoutCount` – WWDT timeout value, count of WWDT clock tick.

`static inline void WWDT_SetWindowValue(WWDT_Type *base, uint32_t windowValue)`

Sets the WWDT window value.

The `WINDOW` register determines the highest `TV` value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in `WINDOW`, a watchdog event will occur. To disable windowing, set `windowValue` to `0xFFFFFFFF` (maximum possible timer value) so windowing is not in effect.

Parameters

- `base` – WWDT peripheral base address
- `windowValue` – WWDT window value.

`void WWDT_Refresh(WWDT_Type *base)`

Refreshes the WWDT timer.

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

- `base` – WWDT peripheral base address

`FSL_WWDT_DRIVER_VERSION`

Defines WWDT driver version.

`WWDT_FIRST_WORD_OF_REFRESH`

First word of refresh sequence

`WWDT_SECOND_WORD_OF_REFRESH`

Second word of refresh sequence

enum __wwdt_status_flags_t

WWDT status flags.

This structure contains the WWDT status flags for use in the WWDT functions.

Values:

enumerator kWWDT__TimeoutFlag

Time-out flag, set when the timer times out

enumerator kWWDT__WarningFlag

Warning interrupt flag, set when timer is below the value WDWARNINT

typedef struct __wwdt_config wwdt_config_t

Describes WWDT configuration structure.

struct __wwdt_config

#include <fsl_wwdt.h> Describes WWDT configuration structure.

Public Members

bool enableWwdt

Enables or disables WWDT

bool enableWatchdogReset

true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset

bool enableWatchdogProtect

true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time

uint32_t windowValue

Window value, set this to 0xFFFFFFFF if windowing is not in effect

uint32_t timeoutValue

Timeout value

uint32_t warningValue

Watchdog time counter value that will generate a warning interrupt. Set this to 0 for no warning

uint32_t clockFreq_Hz

Watchdog clock source frequency.

Chapter 3

Middleware

3.1 Boot

3.1.1 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.

3.1.2 MCUboot



This is MCUboot version 2.1.0

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 Slack channel](#) [Get your invite](#)

3.2 Motor Control

3.2.1 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 applications 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_QUEUE_SIZE

```
#define FMSTR_COMM_QUEUE_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_QUEUE_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_QUEUE_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-language 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_UINTn	Unsigned integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_SINTn	Signed integer type of size <i>n</i> bits (n=8,16,32,64)
FMSTR_TSA_FRACn	Fractional number of size <i>n</i> bits (n=16,32,64).
FMSTR_TSA_FRAC_Q(<i>m,n</i>)	Signed fractional number in general Q form (m+n+1 total bits)
FMSTR_TSA_FRAC_UQ(<i>m,n</i>)	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(<i>name</i>)	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
<i>FM-STR_ADDR</i> 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.	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.
<i>FM-STR_SIZE</i> It is required that this type is unsigned and at least 16 bits wide integer.	Data type used to hold the memory block size.
<i>FM-STR_BOOL</i> This type is used only in zero/non-zero conditions in the driver code.	Data type used as a general boolean type.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command code.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to create the Application Command data buffer.
<i>FM-STR_APPCM</i> Generally, this is an unsigned 8-bit value.	Data type used to hold the Application Command result code.

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.

<i>FM-STR_TSA_TII</i>	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 <i>FM-STR_SIZE</i> .
<i>FM-STR_TSA_TS</i>	Data type used to hold a memory block size, as used in the TSA descriptors. By default, this is defined as <i>FM-STR_SIZE</i> .

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

<i>FM-STR_HPIPE</i>	Pipe handle that identifies the open-pipe object. Generally, this is a pointer to a void type.
<i>FM-STR_PIPE_PC</i>	Integer type required to hold at least 7 bits of data. Generally, this is an unsigned 8-bit or 16-bit type.
<i>FM-STR_PIPE_SI</i>	Integer type required to hold at least 16 bits of data. This is used to store the data buffer sizes.
<i>FM-STR_PPIPEF</i>	Pointer to the pipe handler function. See FM-STR_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.

<i>FMSTR_U8</i>	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.	
<i>FMSTR_U16</i>	Unsigned 16-bit integer.
<i>FMSTR_U32</i>	Unsigned 32-bit integer.
<i>FMSTR_S8</i>	Signed 8-bit integer.
<i>FMSTR_S16</i>	Signed 16-bit integer.
<i>FMSTR_S32</i>	Signed 32-bit integer.
<i>FMSTR_FLOAT</i>	4-byte standard IEEE floating-point type.
<i>FMSTR_FLAGS</i>	Data type forming a union with a structure of flag bit-fields.
<i>FMSTR_SIZE8</i>	Data type holding a general size value, at least 8 bits wide.
<i>FMSTR_INDEX</i>	General for-loop index. Must be signed, at least 16 bits wide.
<i>FMSTR_BCHR</i>	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.	
<i>FMSTR_BPTR</i>	A pointer to the communication buffer (an array of <i>FMSTR_BCHR</i>).

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.

3.3 Wireless

3.3.1 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
- HWPParameters: 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

7.0.2 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.
- [HWPParameter][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 timer 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 `gRng-MaxRequests_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 algorithm 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_config.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 gHwParamsProdDataMainFlash2IfirMode_c to gHwParamsProdDataIfirMode_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, frdm-mcxw72 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 variable. Allow the application to override the trig sample number parameter when gFsciOverRpmMsg_c is set to 1
- [DBG][btsnoop] Add a mechanism to dump raw HCI data via UART using SBT-SNOOP_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.
 - * Explicated RNG error codes
 - * Added argument checks for all APIs and return gRngBadArguments_d (-2) when wrong
 - * added checks of RNG initialization and return gRngNotInitialized_d (-3) when not done
 - * fixed correctness 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 defragmentation 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 clod 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()` renamed 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

Unsupported 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 Seed 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 firmware 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_HSL0T 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() has an important fix for reading the Timestamp in TSTMRO
 - 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_INITRFSWITCHCONTROL Pins 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 the 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 gPlatformUseUniqueIdFor15_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 functionality, 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 trimming 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 oscillator 32k to 3
- [platform] Switch back to RNG for MAC Address 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 gBoard-UseUart0HwFlowControl, 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 re-generated

6.0.11: KW45/K32W1 PRC3.1

FSCI: Framework Serial Communication Interface

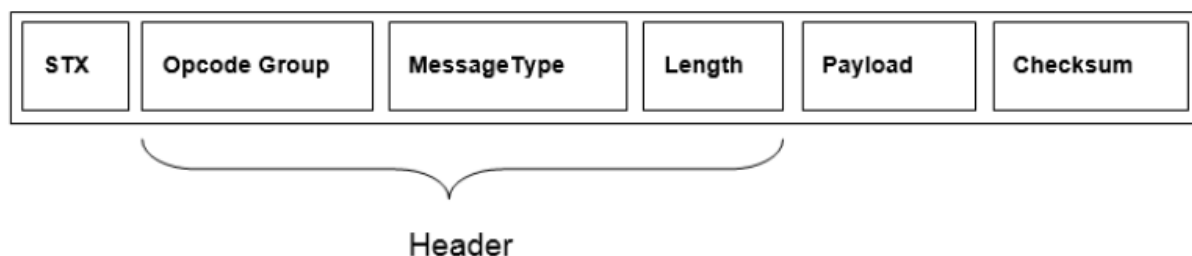
Overview The Framework Serial Communication Interface (FSCI) is both a software module and a protocol that allows monitoring and extensive testing of the protocol layers. It also allows separation of the protocol stack between two protocol layers in a two processing entities setup, the host processor (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.

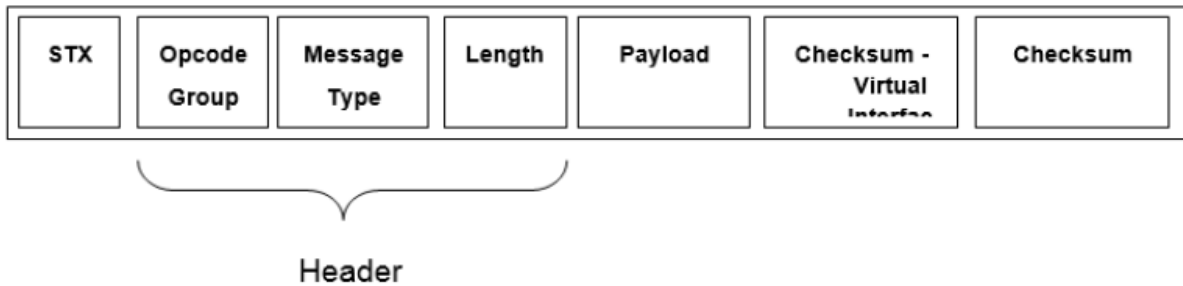
An entity which needs to be interfaced to the FSCI module can use the API to register opcodes to specific interfaces. After doing so, any packet coming from that interface with the same opcode triggers a callback execution. Two or more entities cannot register the same opcode on the same interface, but 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 configures 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_transmitFormattedPacket(pClientPacket, interfaceId);
            return;
        }
        case 0x02:
        {
            /* Allocate 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_transmitFormattedPacket(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 allotted 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) gHwParamsProdDataMainFlash2IfirMode_c(1) : - PROD_DATA are located in IFR1, but Main-Flash 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: gHwParamsProdDataPlacementIfirMode_c) may be programmed.

3) gHwParamsProdDataIfirMode_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 fulfill 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 resources (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.

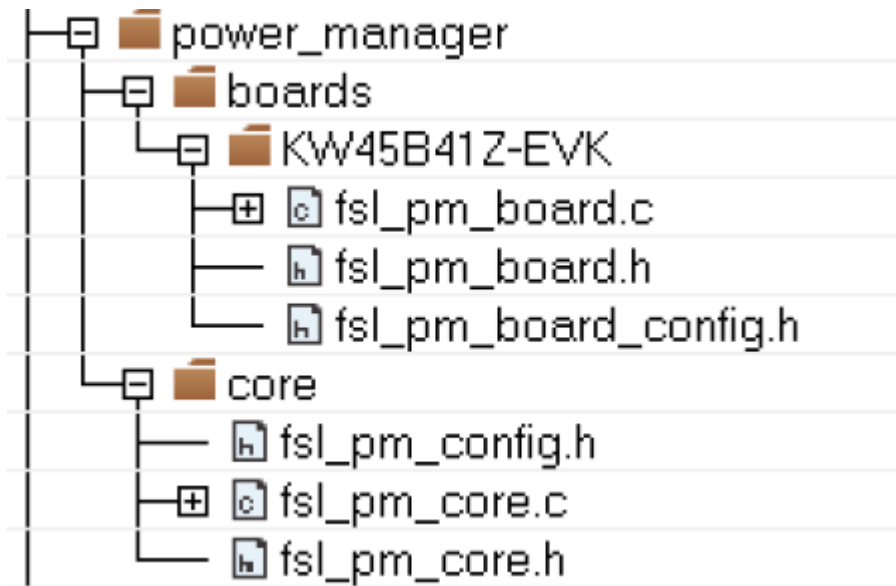
Other peripherals that require specific action on low power entry and restore on low power exit should be added to low power board files. For more details, refer to section Low power board file update

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.

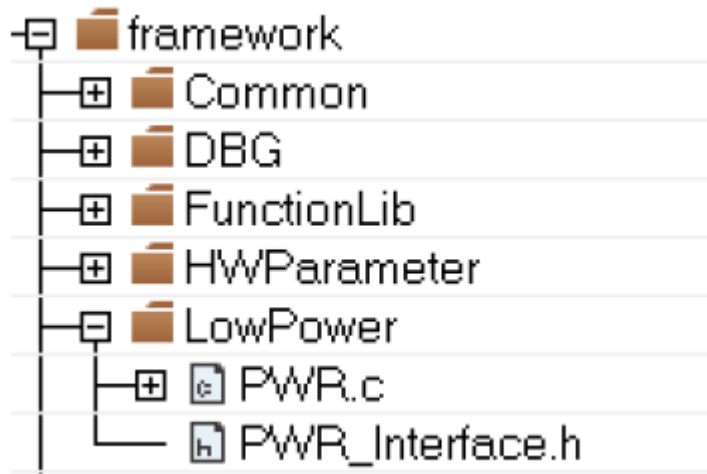
However, users can start directly from the application project and implement low power in it, by performing the steps described in the following sections.

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.

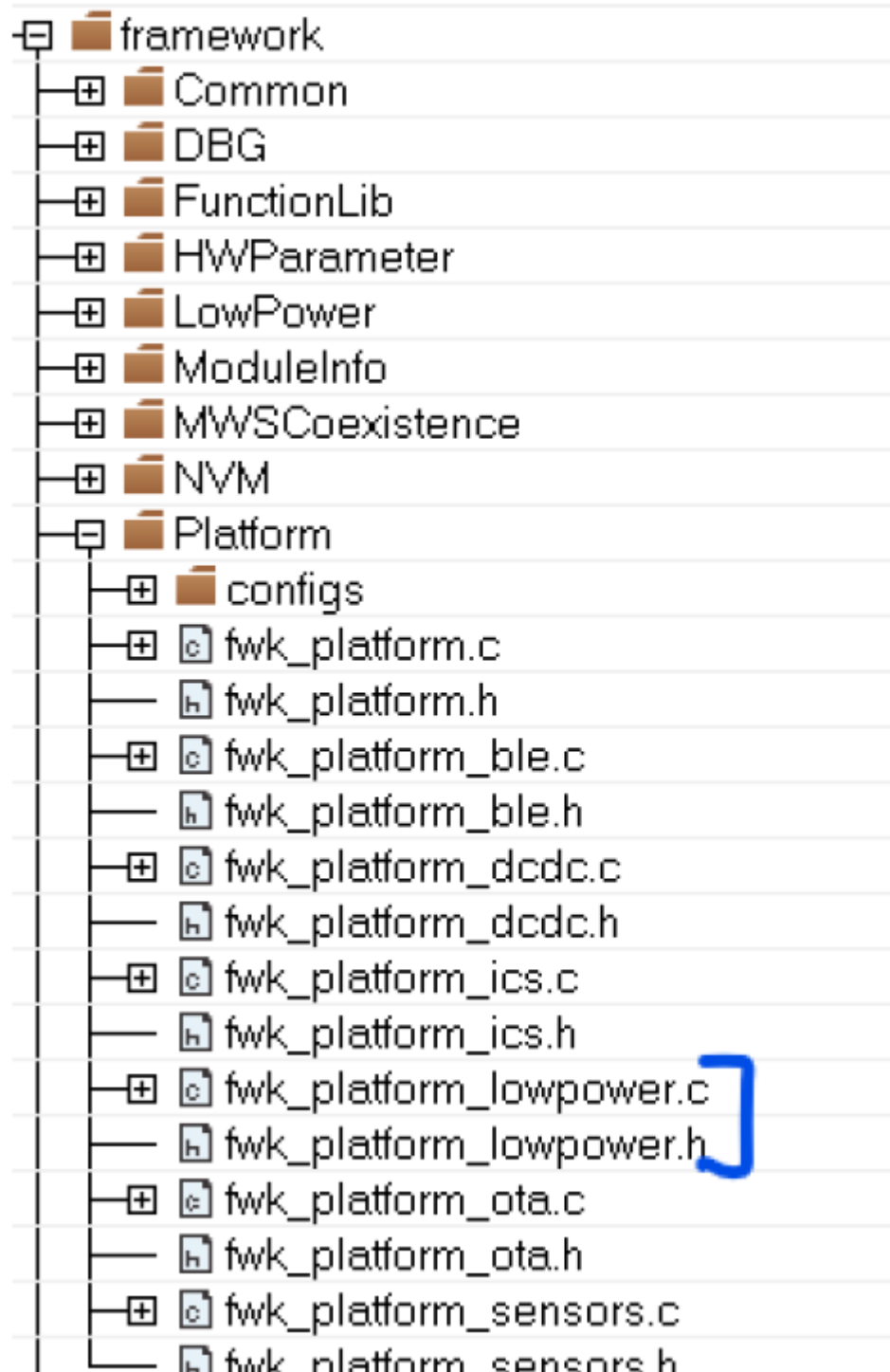
2.1.2 - PWR connectivity framework module PWR.c PWR_Interface.h shall be added to your application projects :



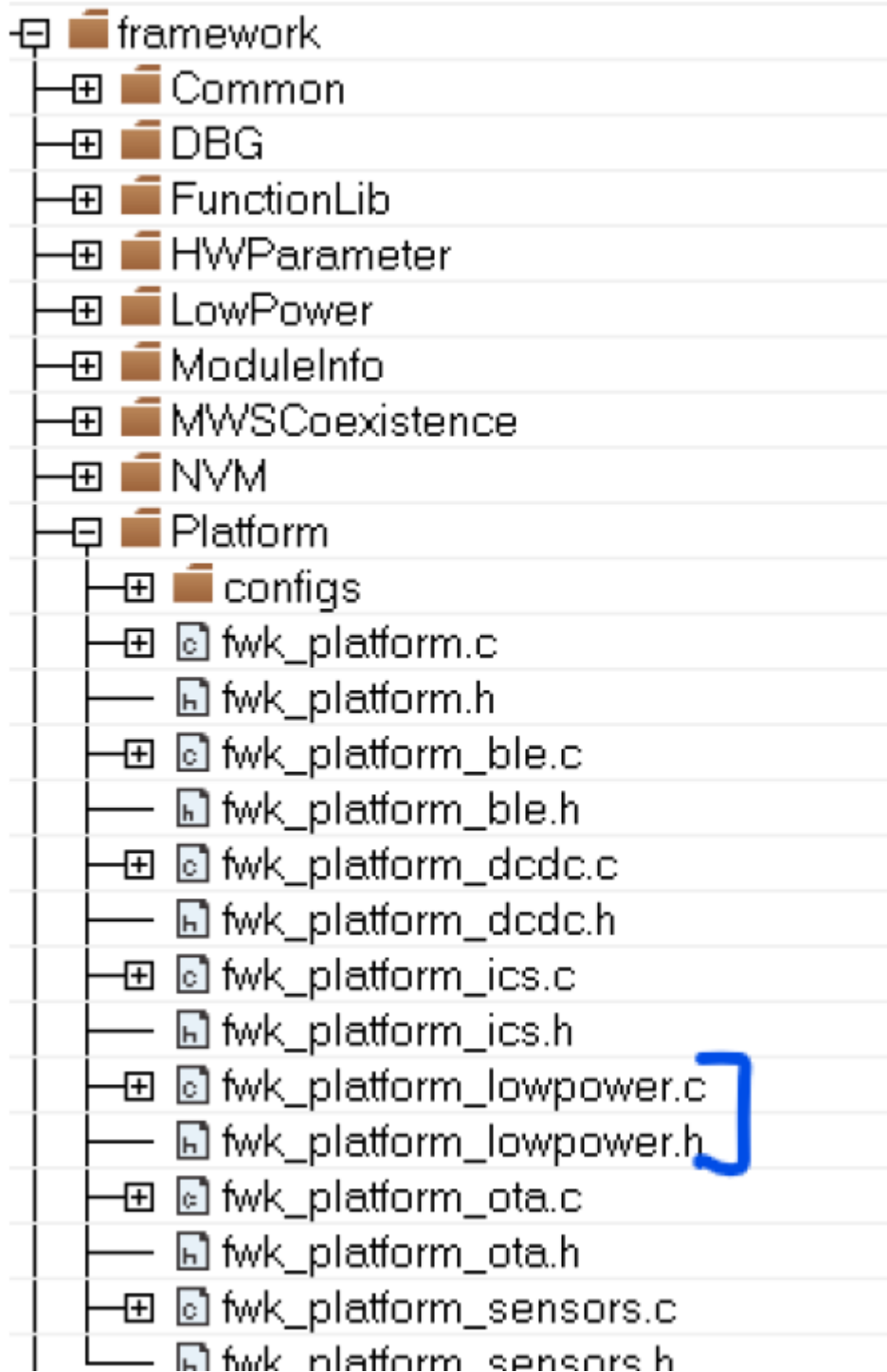
Optionally, in order to support SysTick less mode, `PWR_systicks.c` or `PWR_systicks_bm.c` could also be added.

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 tickless 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 serial 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 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 aggressive 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 definition 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 written 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 `gApp-TaskWaitTimeout_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 Manager 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 suspended, 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 manual 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 save 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 take 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 than 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

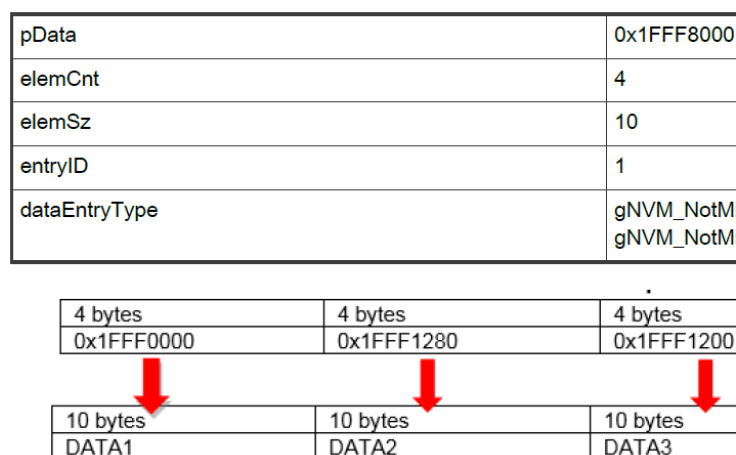
NVM Table entry As show above, A NVM table entry contains a generic pointer to a contiguous RAM data structure, the number of elements the structure contains, the size of a single element, a table entry ID, and an entry type.

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 :

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 $0x1FFF8000 + 2 * \text{elemSz}$. 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 $0x1FFF8000 + 2 * \text{sizeof(void*)}$.

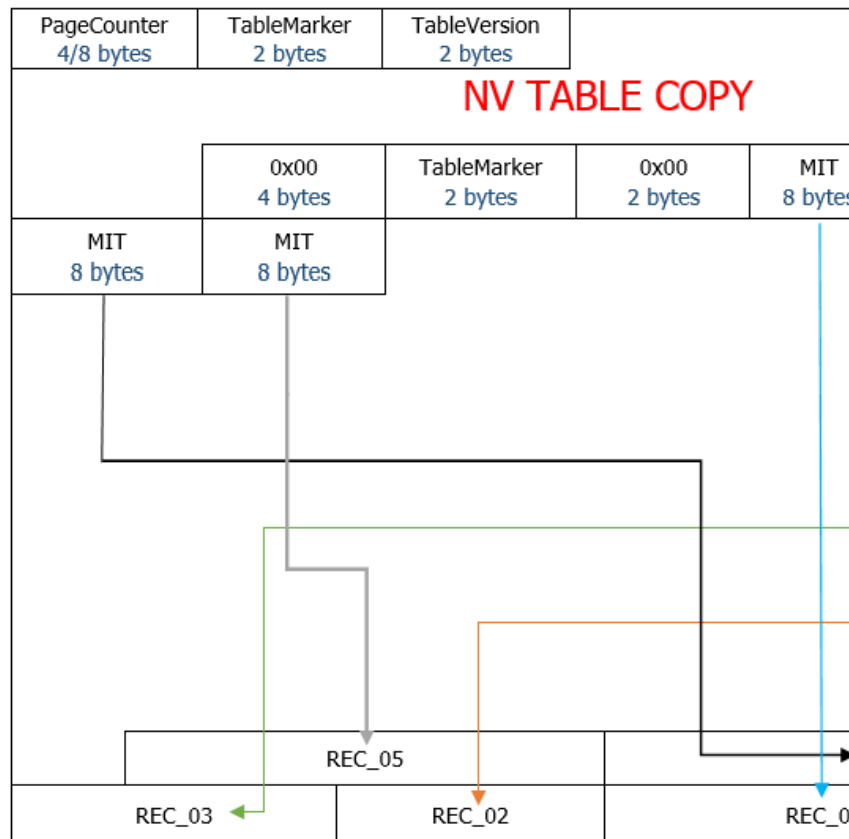
The page validity is guaranteed by the page counter. The page counter is a 32-bit value and is written at the beginning and at the end of the active page. The values need to be equal to consider the page a valid one. The value of the page counter is incremented after each page copy operation. A page 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

entryId	entryType	elemCnt	elemSz
2 bytes	2 bytes	2 bytes	2 bytes

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:

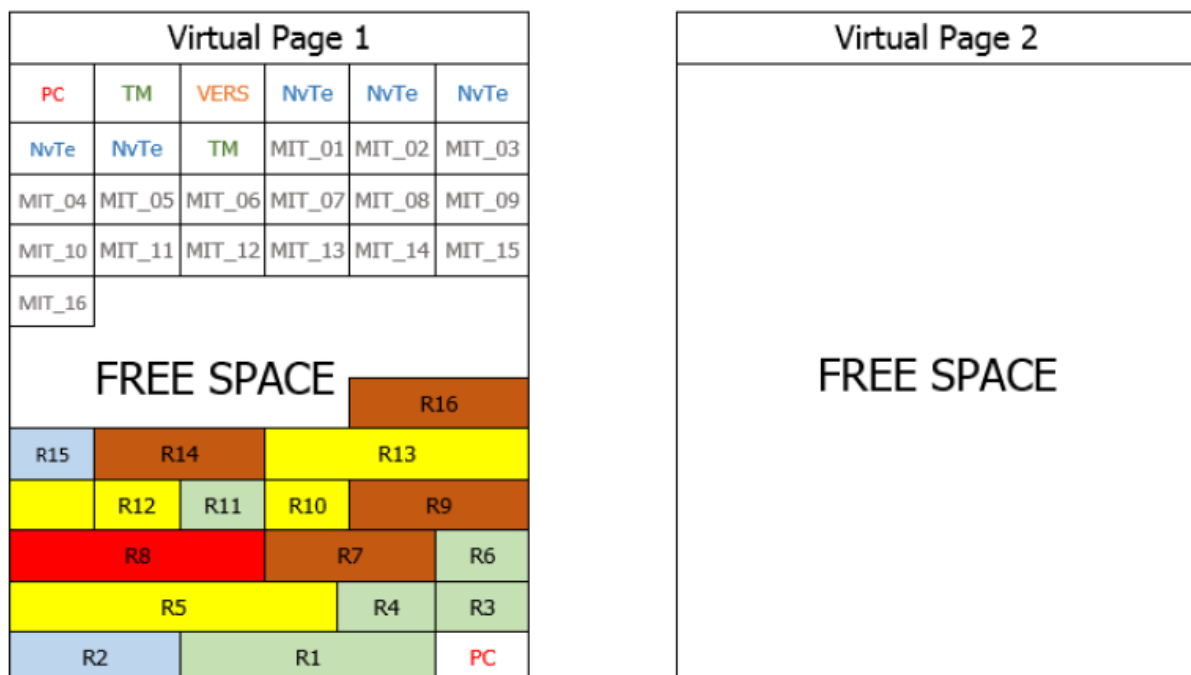
VSB	entryID	elemIdx	recordOffset	VEB
1 byte	2 bytes	2 bytes	2 bytes	

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.

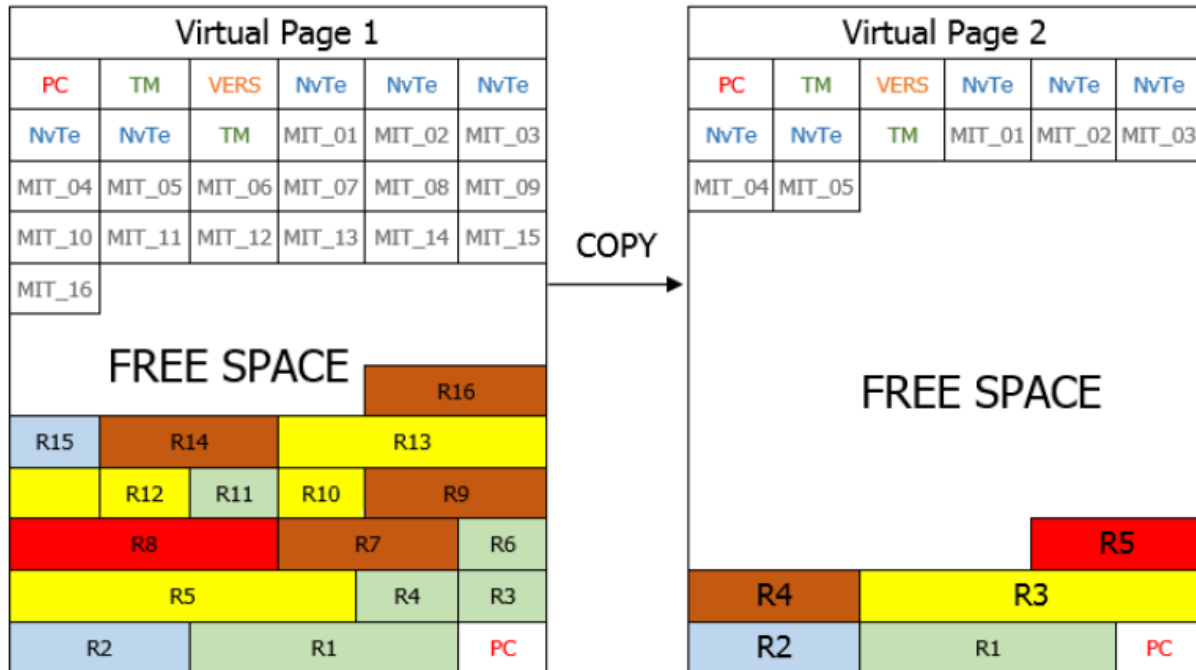


In this example, there are five datasets (one color for each dataset) with both 'full' and 'single' record types.

- 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 bus 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 alternatively 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 subsystem. 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 from 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_ecdh.c	Mbedtls SecLib_mbedtls.c	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	en-	x	x				
AES_128_OFB encryption	En-	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 preventing lowpower (only WFI) during its processing. Platform specific code can be found 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 obtained 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 synchronization 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 switch 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 than `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: $\text{new_estimation} = (\text{new_measurement} + ((1 \ll n) - 1) * \text{last_estimation}) \gg n$

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 FRO32K. 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.

3.3.2 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 MCS0~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 MCS0~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 MCS0~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_ieeeeps_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_ieeeeps_config::ps_null_interval`

The interval that STA sends null packet

`t_u32 wlan_ieeeeps_config::multiple_dtim_interval`

The count of listen interval

`t_u32 wlan_ieeeeps_config::listen_interval`

Periodic interval that STA listens to AP beacons

`t_u32 wlan_ieeeeps_config::adhoc_awake_period`

Periodic awake period for adhoc networks

`t_u32 wlan_ieeeeps_config::bcn_miss_timeout`

Beacon miss timeout in milliseconds

`t_s32 wlan_ieeeeps_config::delay_to_ps`

The delay of enabling IEEE-PS in milliseconds

`t_u32 wlan_ieeeeps_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 tw_tcapab
```

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 mfpcc
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 802.1x 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_INVALID 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_INVALID.

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 mfp and mfp 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_INVALID 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_INVALID 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_INVALID` 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_INVALID` 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_INVALID 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_INVALID` 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_INVALID` 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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_INVALID 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 disabled.

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_ieeeeps_cfg (struct wlan_ieeeeps_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*: timeout 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_ieeeeps_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_ieeeeps_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_deepsleeps_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_deepsleeps_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 (wlanflt_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 wlanfltcfg_t structure.

wlanfltcfg_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(wlanfltcfg_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_INVALID 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_INVALID 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 is 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_httxcf (unsigned short *httxcf*)

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 * *rss*, 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 * *rss*, 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 programming) memory

Parameters

Returns

WM_SUCCESS if user data read operation is successful.
-WM_E_INVALID 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_INVALID 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_preferch_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_rcv_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 adding 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 be 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_dtim*, 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 wififlt_cfg_t wlanflt_cfg_t

Configuration for memory efficient filters in Wi-Fi firmware from wififlt_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 4

RTOS

4.1 FreeRTOS

4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

[FreeRTOS kernel for MCUXpresso SDK Readme](#)

[FreeRTOS kernel for MCUXpresso SDK ChangeLog](#)

[FreeRTOS kernel Readme](#)

4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

4.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

[Readme](#)

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

4.1.5 corejson

JSON parser.

Readme

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

4.1.7 coremqtt-agent

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library.

Readme

4.1.8 corepkcs11

PKCS #11 key management library.

Readme

4.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme