



MCUXpresso SDK Documentation

Release 25.09.00-pvw2



NXP
Aug 12, 2025



Table of contents

1	LPCXpresso55S69	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with MCUXpresso SDK Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	55
1.3.1	Getting Started with MCUXpresso SDK Repository	55
1.4	Release Notes	68
1.4.1	MCUXpresso SDK Release Notes	68
1.5	ChangeLog	73
1.5.1	MCUXpresso SDK Changelog	73
1.6	Driver API Reference Manual	73
1.7	Middleware Documentation	73
1.7.1	Multicore	73
1.7.2	MCU Boot	74
1.7.3	Audio Voice components	74
1.7.4	Maestro Audio Framework for MCU	74
1.7.5	FreeMASTER	74
1.7.6	FreeRTOS	74
1.7.7	File systemFatfs	74
2	LPC55S69	75
2.1	ANACTRL: Analog Control Driver	75
2.2	CASPER: The Cryptographic Accelerator and Signal Processing Engine with RAM sharing	80
2.3	casper_driver	80
2.4	casper_driver_pkha	83
2.5	Clock Driver	86
2.6	CMP: Analog Comparator Driver	117
2.7	CRC: Cyclic Redundancy Check Driver	121
2.8	CTIMER: Standard counter/timers	123
2.9	DMA: Direct Memory Access Controller Driver	133
2.10	IAP: In Application Programming Driver	150
2.11	IAP_FFR Driver	159
2.12	FLEXCOMM: FLEXCOMM Driver	168
2.13	FLEXCOMM Driver	168
2.14	GINT: Group GPIO Input Interrupt Driver	169
2.15	Hashcrypt: The Cryptographic Accelerator	172
2.16	Hashcrypt Background HASH	172
2.17	Hashcrypt common functions	173
2.18	Hashcrypt AES	175
2.19	Hashcrypt HASH	179
2.20	I2C: Inter-Integrated Circuit Driver	181
2.21	I2C DMA Driver	181
2.22	I2C Driver	183
2.23	I2C Master Driver	186
2.24	I2C Slave Driver	196

2.25	I2S: I2S Driver	205
2.26	I2S DMA Driver	205
2.27	I2S Driver	208
2.28	INPUTMUX: Input Multiplexing Driver	217
2.29	IAP_KBP Driver	231
2.30	Common Driver	235
2.31	LPADC: 12-bit SAR Analog-to-Digital Converter Driver	247
2.32	GPIO: General Purpose I/O	266
2.33	IOCON: I/O pin configuration	268
2.34	Mailbox	270
2.35	MRT: Multi-Rate Timer	270
2.36	OSTIMER: OS Event Timer Driver	275
2.37	PINT: Pin Interrupt and Pattern Match Driver	278
2.38	PLU: Programmable Logic Unit	287
2.39	Power Driver	296
2.40	POWERQUAD: PowerQuad hardware accelerator	310
2.41	PRINCE: PRINCE bus crypto engine	340
2.42	PUF: Physical Unclonable Function	347
2.43	Reset Driver	349
2.44	RNG: Random Number Generator	354
2.45	RTC: Real Time Clock	356
2.46	SCTimer: SCTimer/PWM (SCT)	361
2.47	SDIF: SD/MMC/SDIO card interface	378
2.48	skboot_authenticate	395
2.49	SPI: Serial Peripheral Interface Driver	396
2.50	SPI DMA Driver	396
2.51	SPI Driver	400
2.52	SYSTCL: I2S bridging and signal sharing Configuration	408
2.53	USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver	411
2.54	USART DMA Driver	411
2.55	USART Driver	414
2.56	UTICK: MicroTick Timer Driver	429
2.57	WWDT: Windowed Watchdog Timer Driver	431
3	Middleware	435
3.1	Boot	435
3.1.1	MCUXpresso SDK : mcuxsdk-middleware-mcuboot_opensource	435
3.1.2	MCUboot	436
3.2	File System	437
3.2.1	FatFs	437
3.3	Motor Control	439
3.3.1	FreeMASTER	439
3.4	MultiCore	476
3.4.1	Multicore SDK	476
3.5	Multimedia	571
3.5.1	Audio Voice	571
4	RTOS	655
4.1	FreeRTOS	655
4.1.1	FreeRTOS kernel	655
4.1.2	FreeRTOS drivers	661
4.1.3	backoffalgorithm	661
4.1.4	corehttp	664
4.1.5	corejson	666
4.1.6	coremqtt	669
4.1.7	coremqtt-agent	672
4.1.8	corepkcs11	676
4.1.9	freertos-plus-tcp	679

This documentation contains information specific to the lpcxpresso55s69 board.

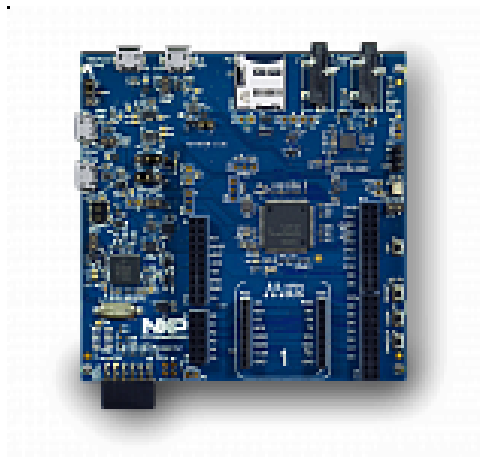
Chapter 1

LPCXpresso55S69

1.1 Overview

The LPCXpresso55S69 board is an LPCXpresso V3 style board, the latest generation of the original and highly successful LPCXpresso board family. “Arduino UNO” compatible shield connectors are included, with additional expansion ports around the Arduino footprint, along with a PMod/host interface port and Mikroelektronika Click module site. The board features an on-board LPC-Link2 debug probe based on the LPC4322 MCU for a performance debug experience over high speed USB, with easy firmware update options to support CMSIS-DSP or a special version of J-link LITE from SEGGER. The board can also be used with an external debug probe such as those from SEGGER and P&E.

The LPC5500 series is fully supported by NXP's [MCUXpresso suite](https://mcuxpresso.nxp.com) of free software and tools, which include an Eclipse-based IDE, configuration tools and extensive SDK drivers/examples available at <https://mcuxpresso.nxp.com>. MCUXpresso SDK includes project files for use with IDEs from lead partners Keil and IAR.



MCU device and part on board is shown below:

- Device: LPC55S69
- PartNumber: LPC55S69JBD100

1.2 Getting Started with MCUXpresso SDK Package

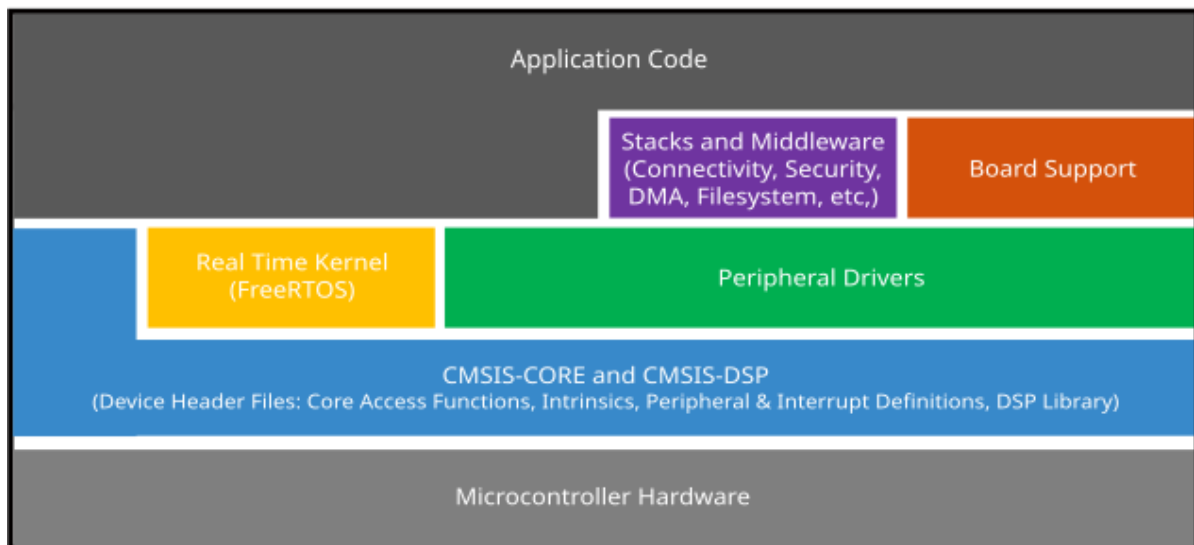
1.2.1 Getting Started with MCUXpresso SDK Package

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, and various other middleware to support rapid development.

For supported toolchain versions, see *MCUXpresso SDK Release Notes* (document MCUXSDKRN).

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



MCUXpresso SDK board support package folders

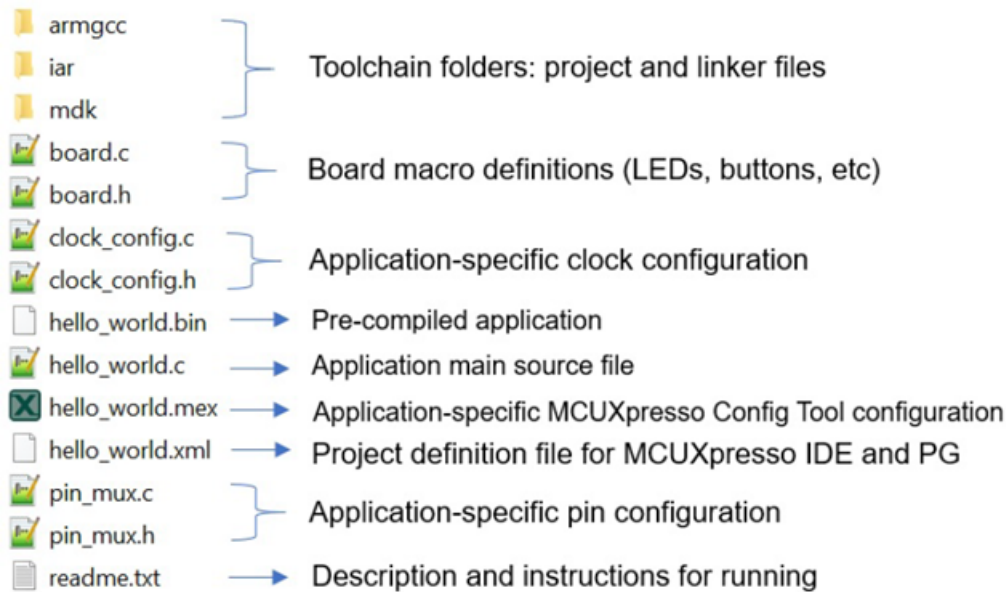
MCUXpresso SDK board support package provides example applications for NXP development and evaluation boards for Arm Cortex-M cores including Freedom, Tower System, and LPCXpresso boards. Board support packages are found inside the top-level boards folder and each supported board has its own folder (an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

- `cmsis_driver_examples`: Simple applications intended to show how to use CMSIS drivers.
- `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).
- `emwin_examples`: Applications that use the emWin GUI widgets.
- `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
- `usb_examples`: Applications that use the USB host/device/OTG stack.

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.

Locating example application source files When opening an example application in any of the supported IDEs, various source files are referenced. The MCUXpresso SDK devices folder is the central component to all example applications. It means that 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>/cmsis_drivers`: All the CMSIS drivers for your specific MCU
- `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
- `devices/<device_name>/project`: Project template used in CMSIS PACK new project creation

For examples containing middleware/stacks or an RTOS, there are references to the appropriate source code. Middleware source files are located in the `middleware` folder and 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.

Run a demo using MCUXpresso IDE

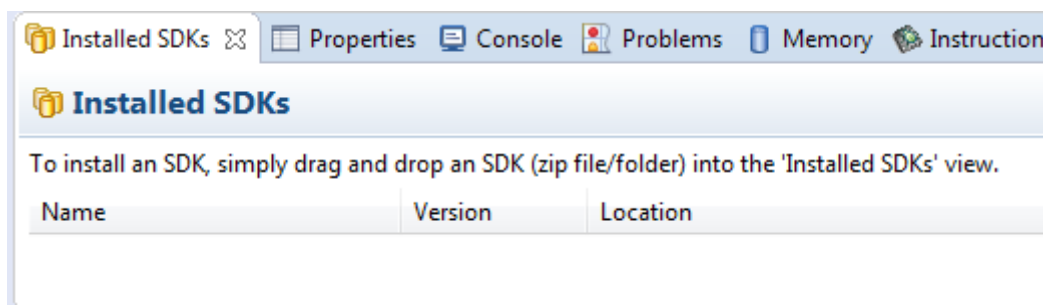
Note: Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

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 hardware platform is used as an example, though these steps can be applied to any example application in the MCUXpresso SDK.

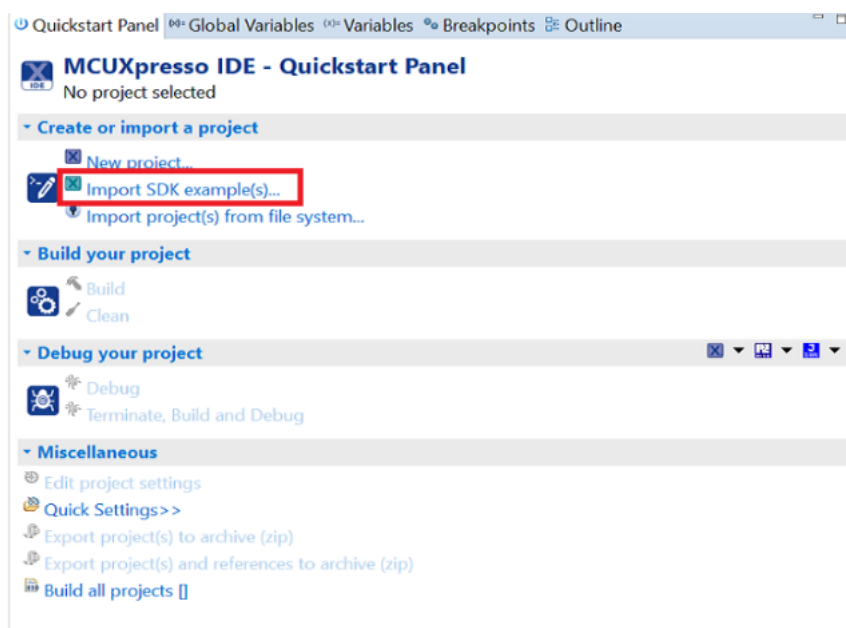
Select the workspace location Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the workspace.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

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

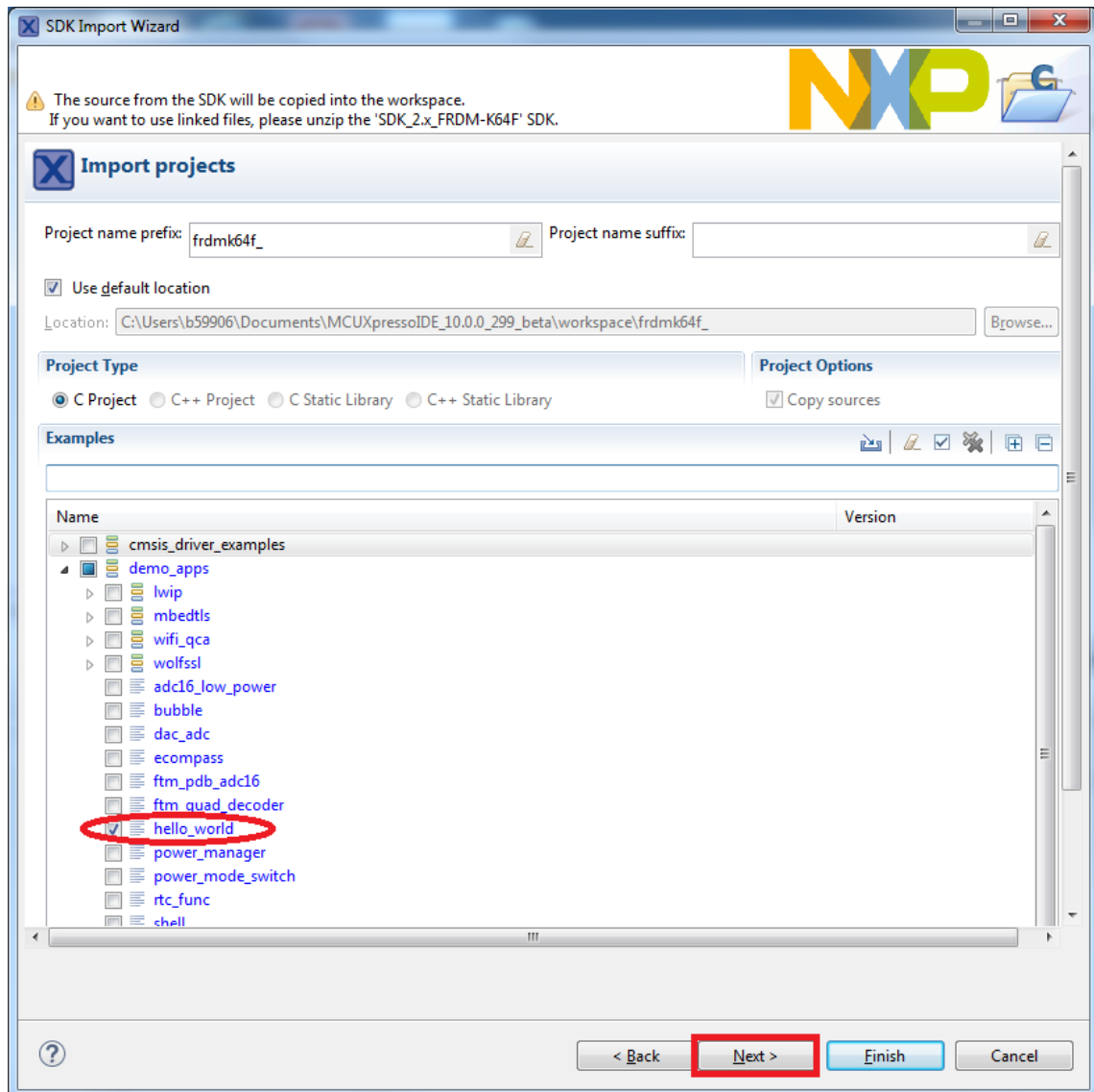
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



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



3. Expand the `demo_apps` folder and select `hello_world`.

4. Click **Next**.

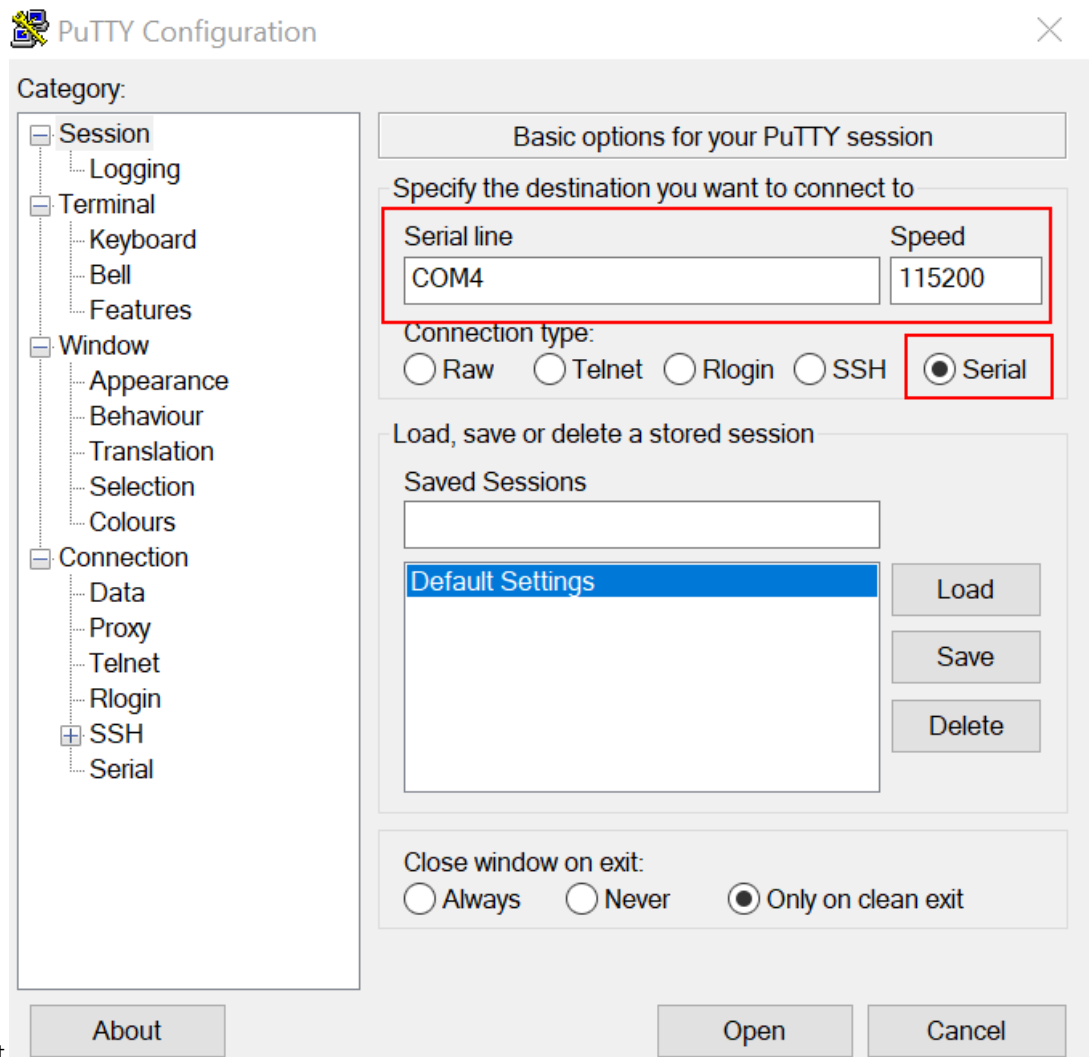
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminal for demo applications such as `adc_basic`, `adc_burst`, `adc_dma`, and `adc_interrupt`. Otherwise, it is not necessary to select this option. Then, click **Finish**.

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:

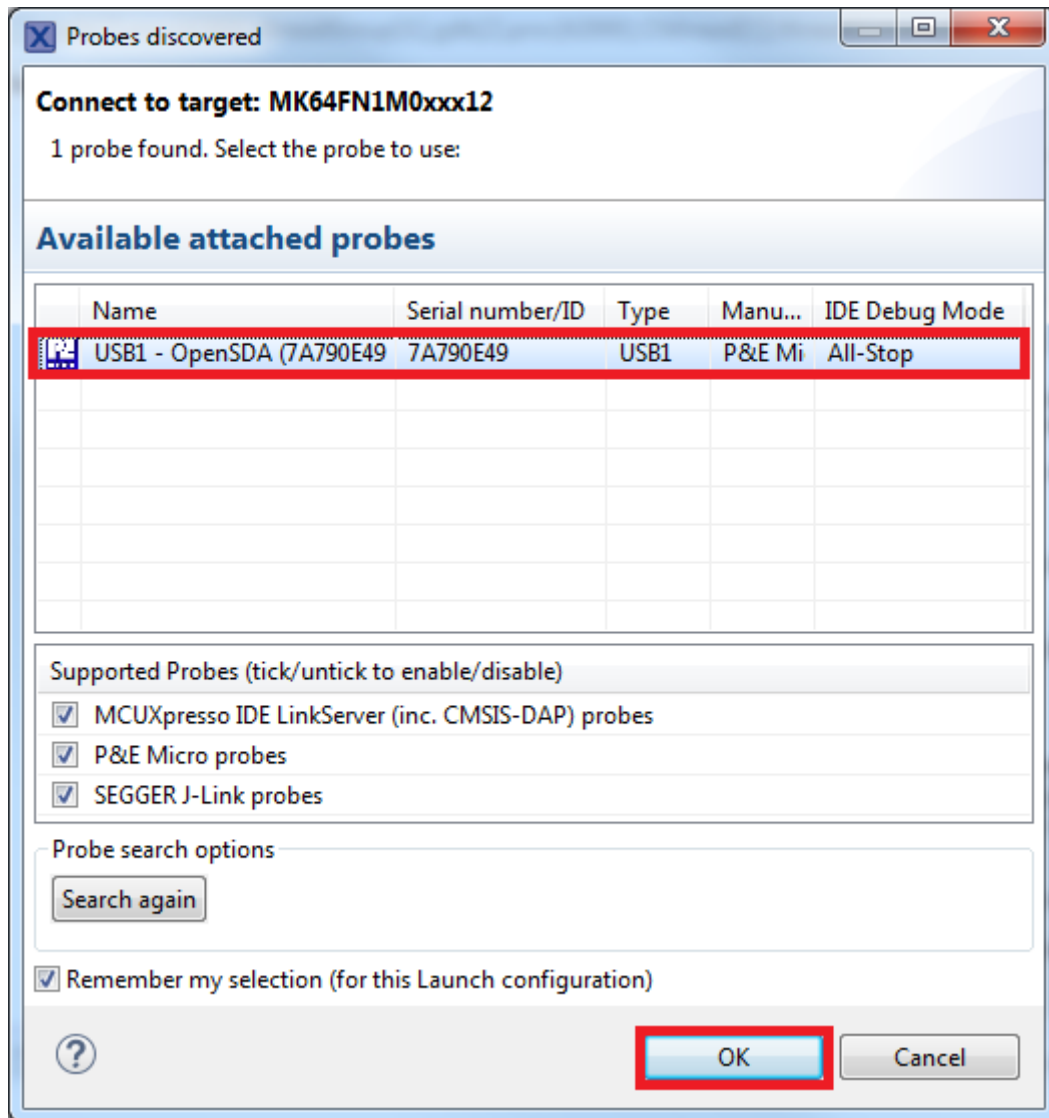
1. Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
2. Connect the development platform to your PC via a USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM 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

4. On the **Quickstart Panel**, click **Debug** to launch the debug session.
5. 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()`.
- Start the application by clicking **Resume**.

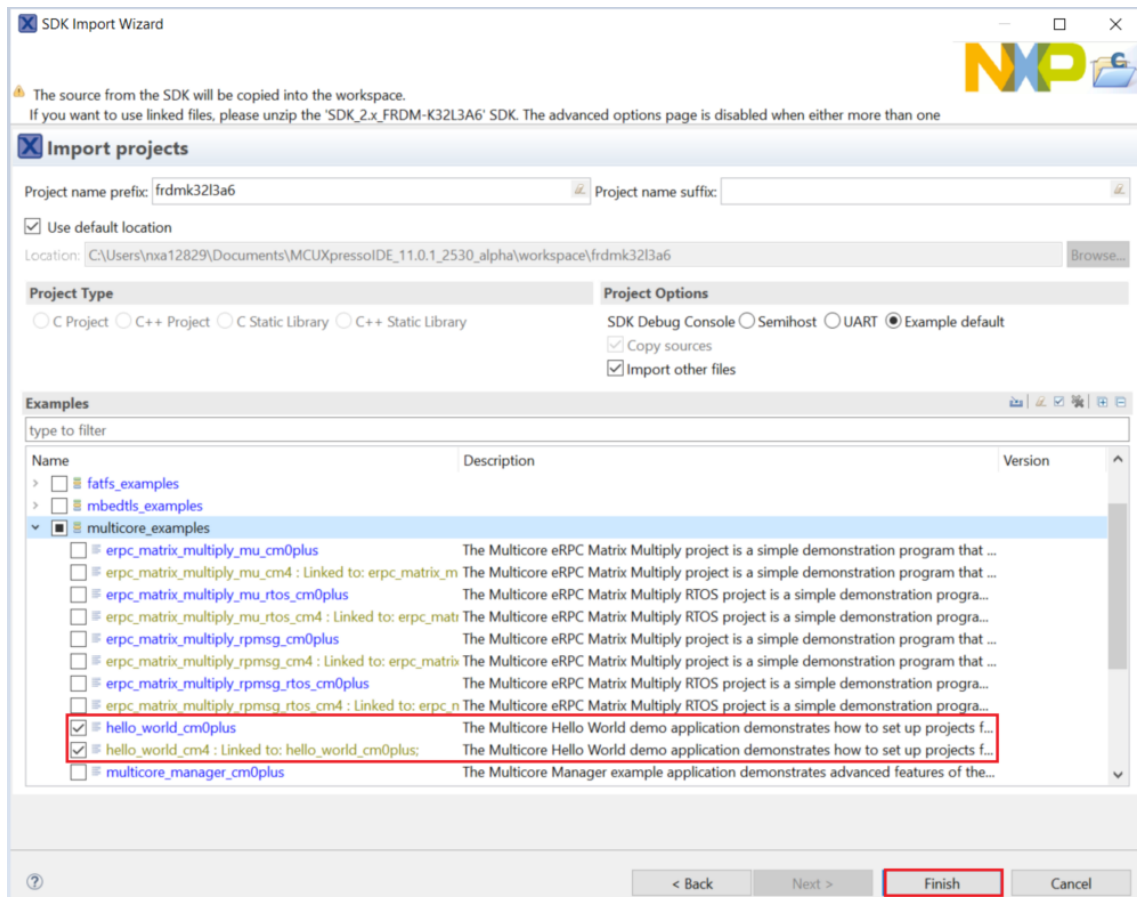


The `hello_world` application is now running and a banner is displayed on the terminal. If not, check your terminal settings and connections.

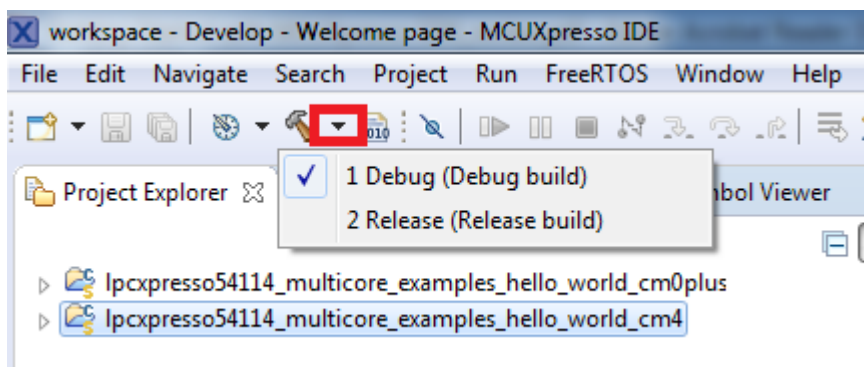


Build a multicore example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug multicore example applications. The following steps can be applied to any multicore example application in the MCUXpresso SDK. Here, the dual-core version of hello_world example application targeted for the LPCXpresso54114 hardware platform is used as an example.

1. Multicore examples are imported into the workspace in a similar way as single core applications, explained in **Build an example application**. When the SDK zip package for LPCXpresso54114 is installed and available in the **Installed SDKs** view, click **Import SDK example(s)...** on the Quickstart Panel. In the window that appears, expand the **LPCxx** folder and select **LPC54114J256**. Then, select **lpcxpresso54114** and click **Next**.
2. Expand the multicore_examples/hello_world folder and select **cm4**. The cm0plus counterpart project is automatically imported with the cm4 project, because the multicore examples are linked together and there is no need to select it explicitly. Click **Finish**.

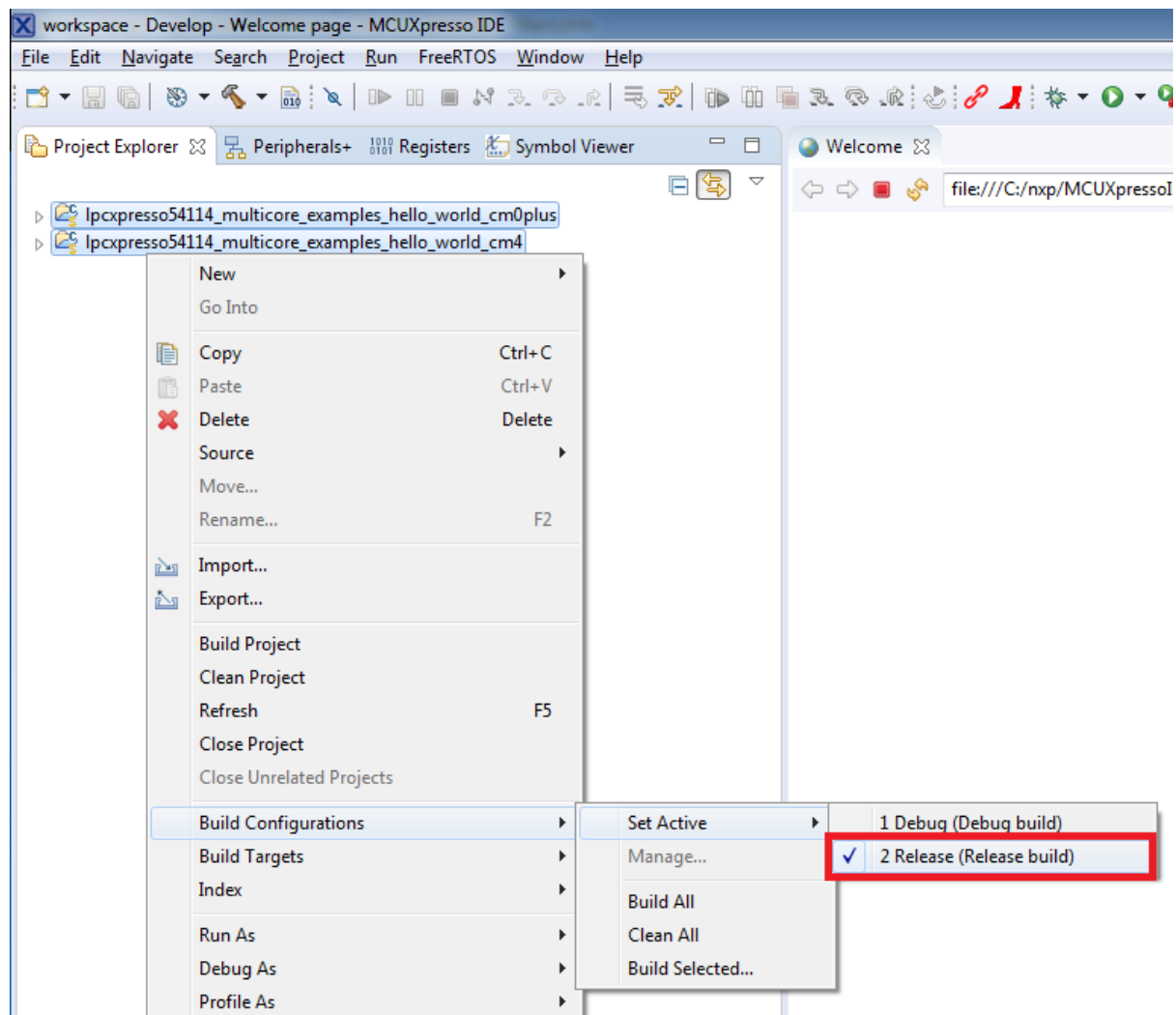


3. Now, two projects should be imported into the workspace. To start building the multicore application, highlight the `lpcxpresso54114_multicore_examples_hello_world_cm4` project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in the figure. For this example, select **Debug**.

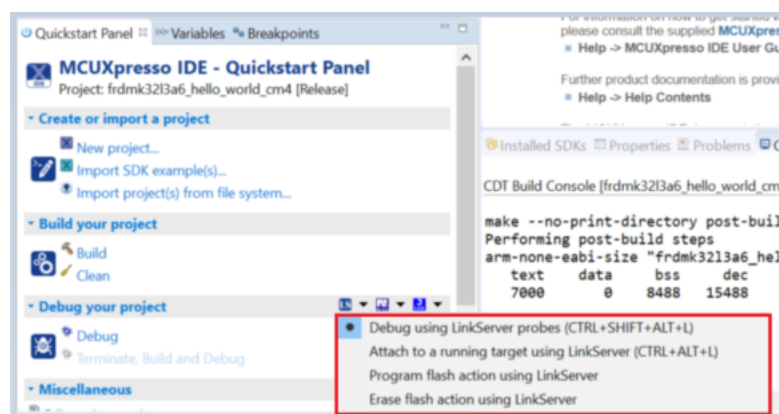


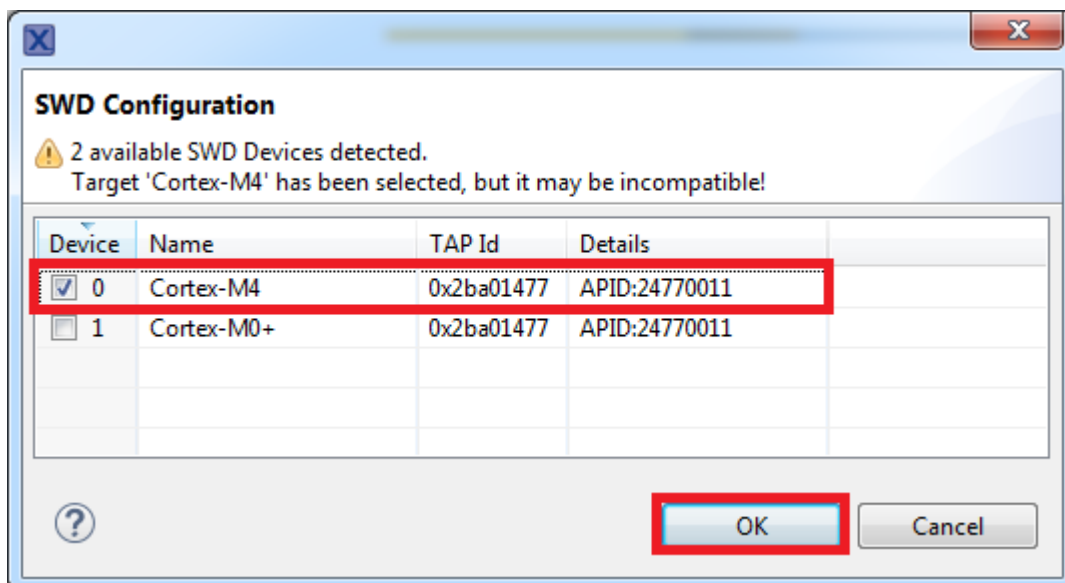
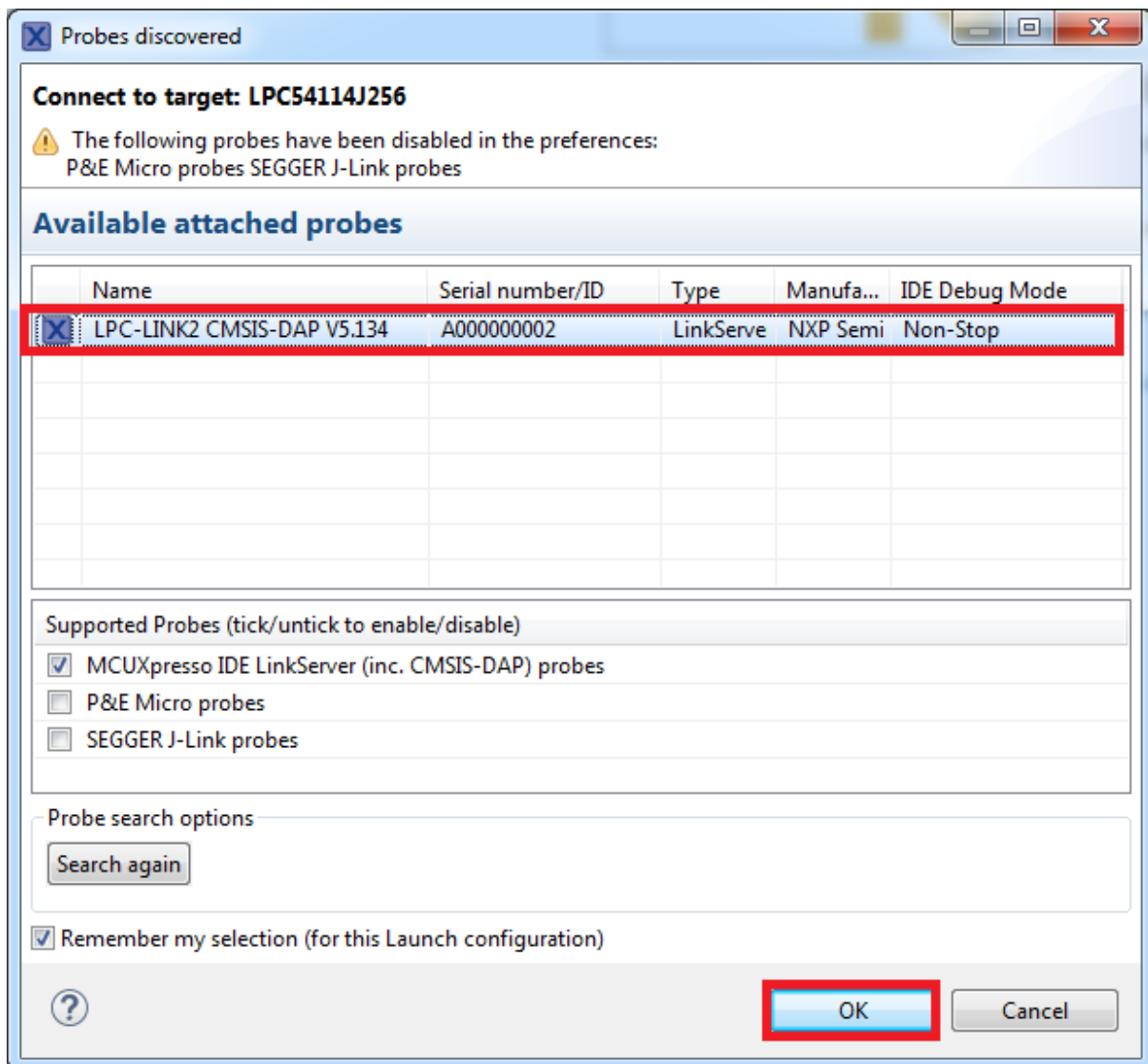
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

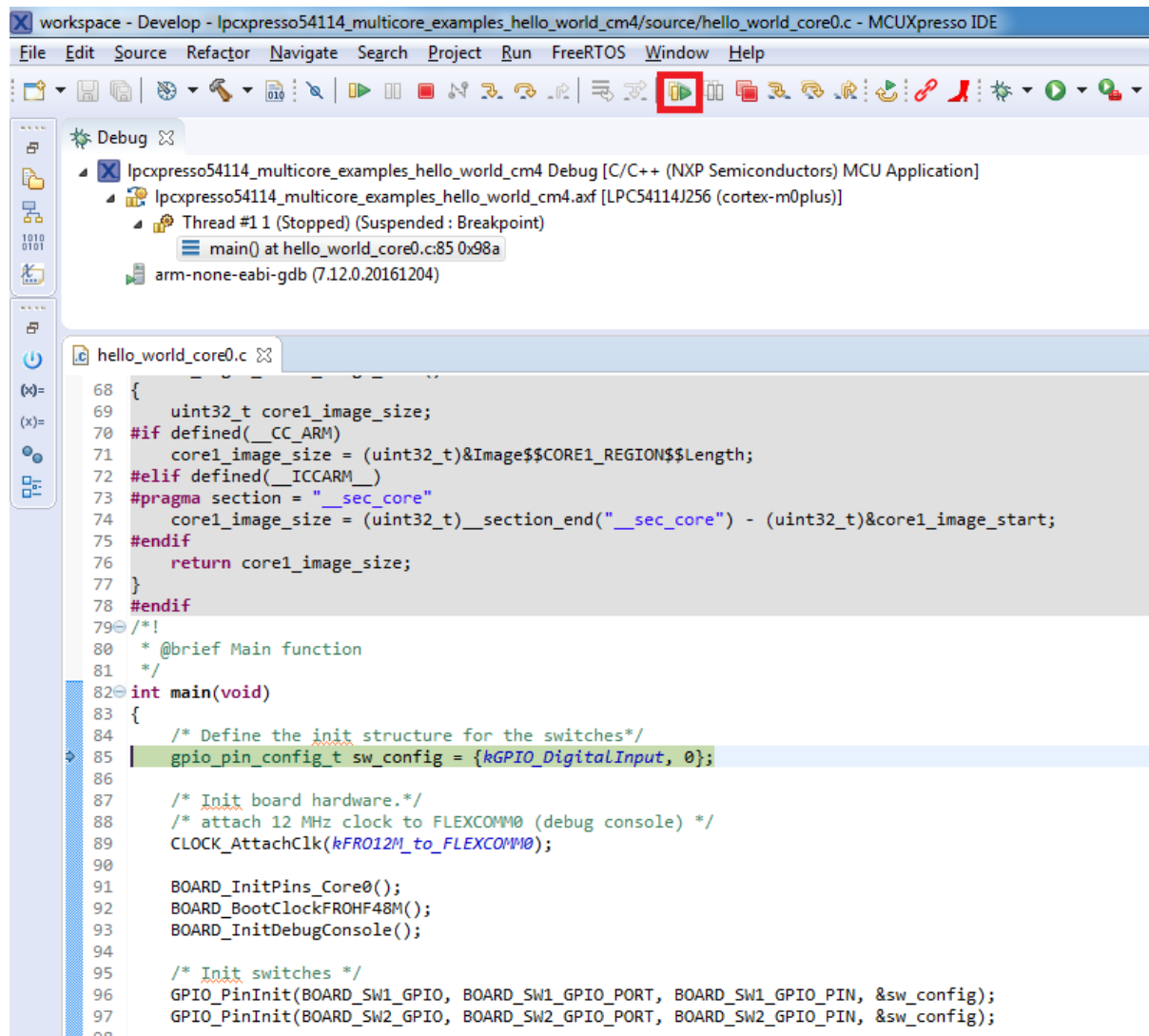
Note: When the **Release** build is requested, it is necessary to change the build configuration of both the primary and auxiliary core application projects first. To do this, select both projects in the Project Explorer view and then right click which displays the context-sensitive menu. Select **Build Configurations -> Set Active -> Release**. This alternate navigation using the menu item is **Project -> Build Configuration -> Set Active -> Release**. After switching to the **Release** build configuration, the build of the multicore example can be started by triggering the primary core application (cm4) build.



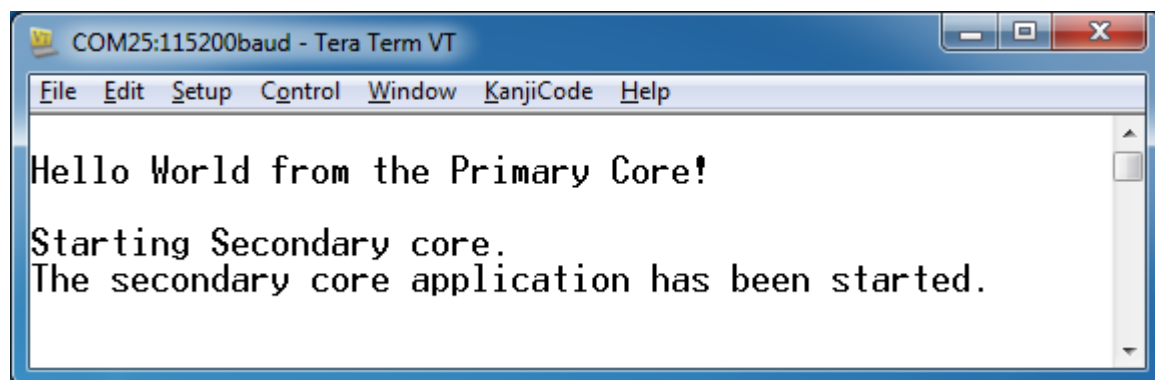
Run a multicore example application The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run an example application**. These steps are common for both single-core applications and the primary side of dual-core applications, ensuring both sides of the multicore application are properly loaded and started. However, there is one additional dialogue that is specific to multicore examples which requires selecting the target core. See the following figures as reference.





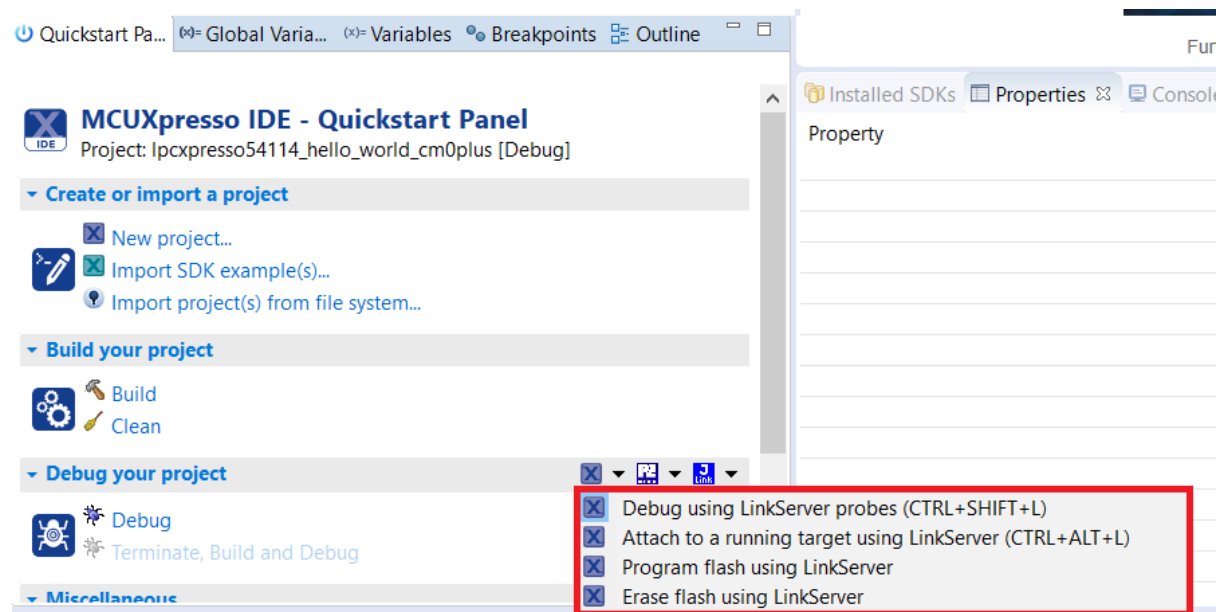


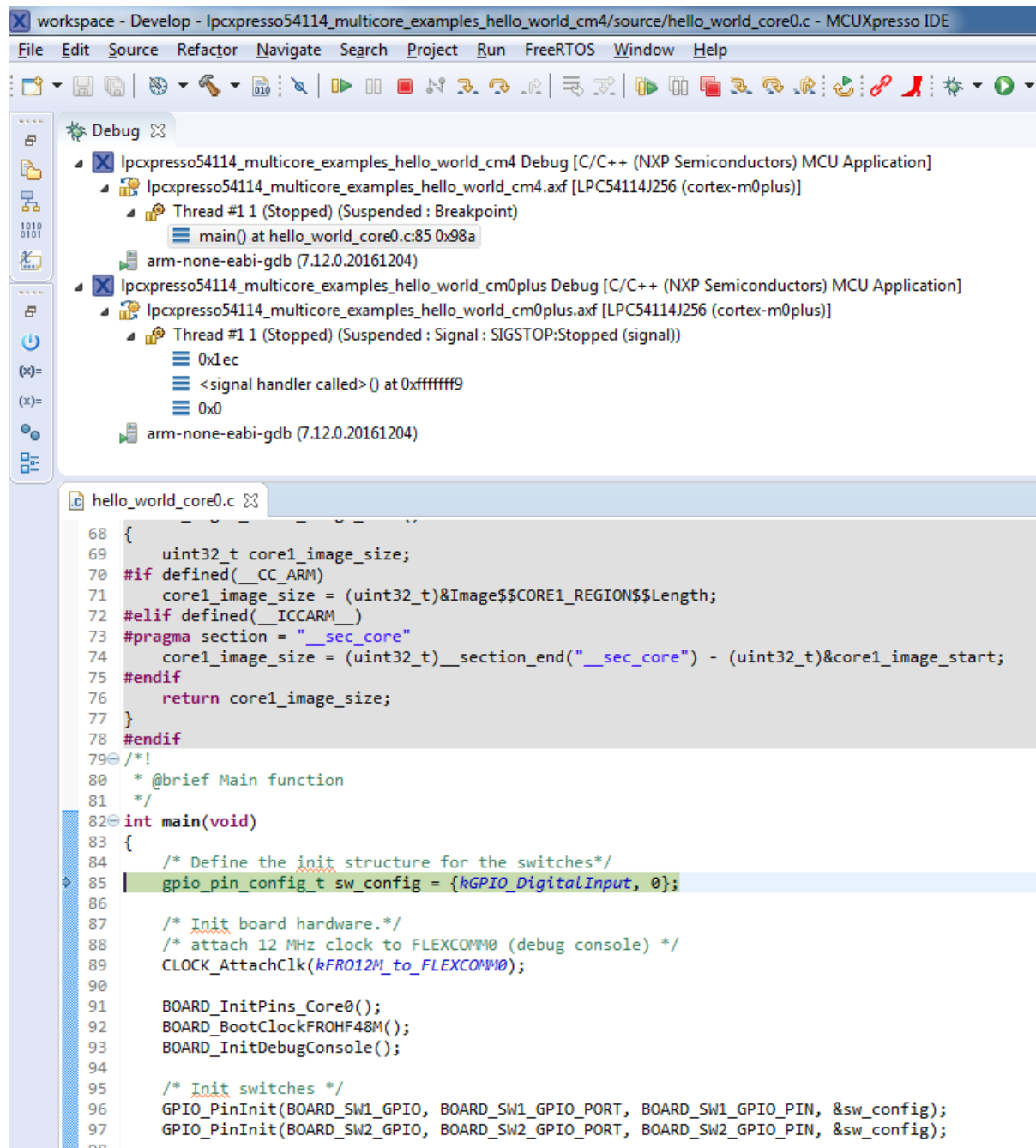
After clicking the “Resume All Debug sessions” button, the hello_world multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.



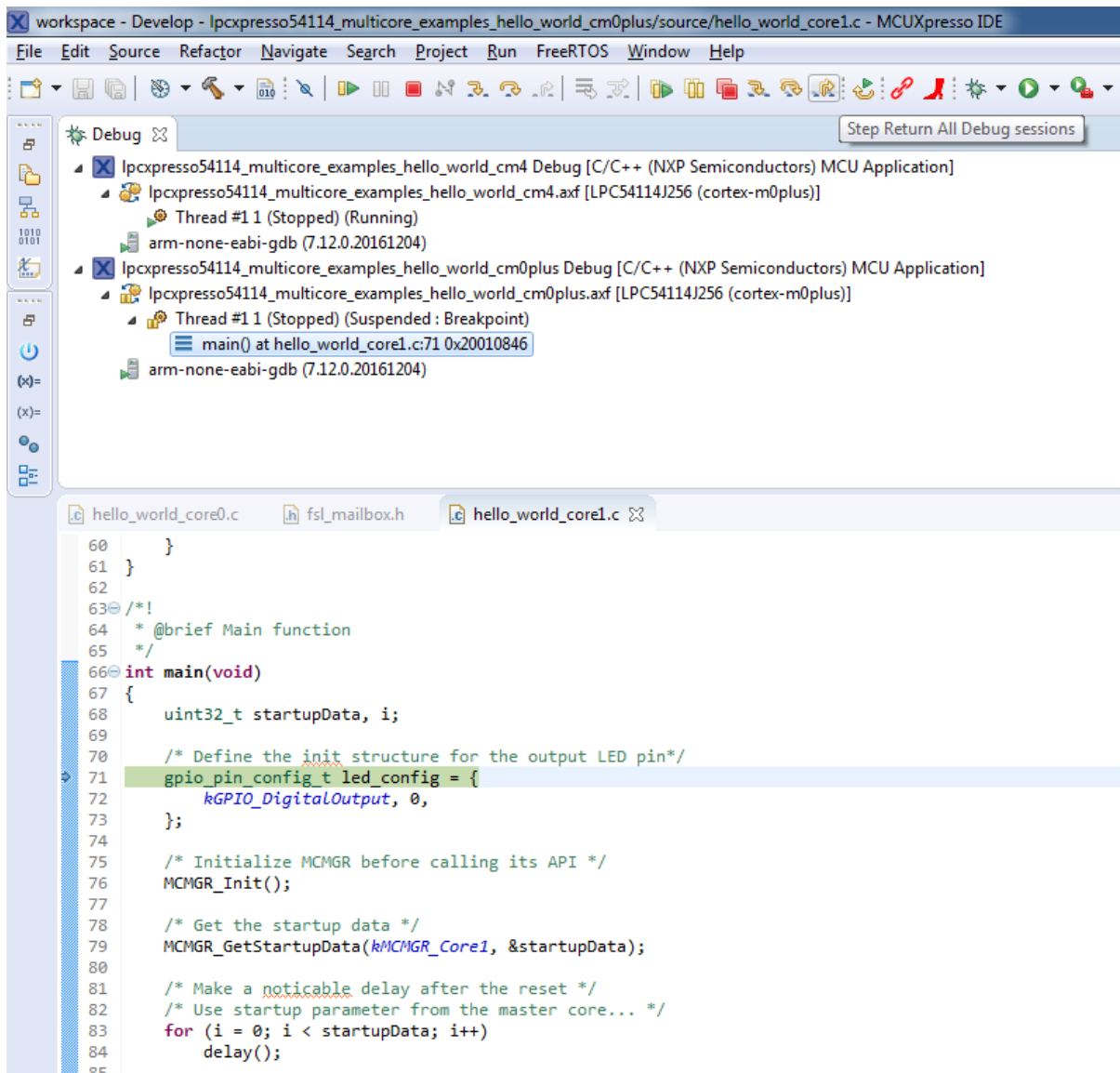
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps also for the auxiliary core application. Highlight the lpcxpresso54114_multicore_examples_hello_world_cm0plus project (multicore slave project) in the Project Explorer. On the Quickstart Panel, click “Debug ‘lpcxpresso54114_multicore_examples_hello_world_cm0plus’ [Debug]” to launch the second debug

session.

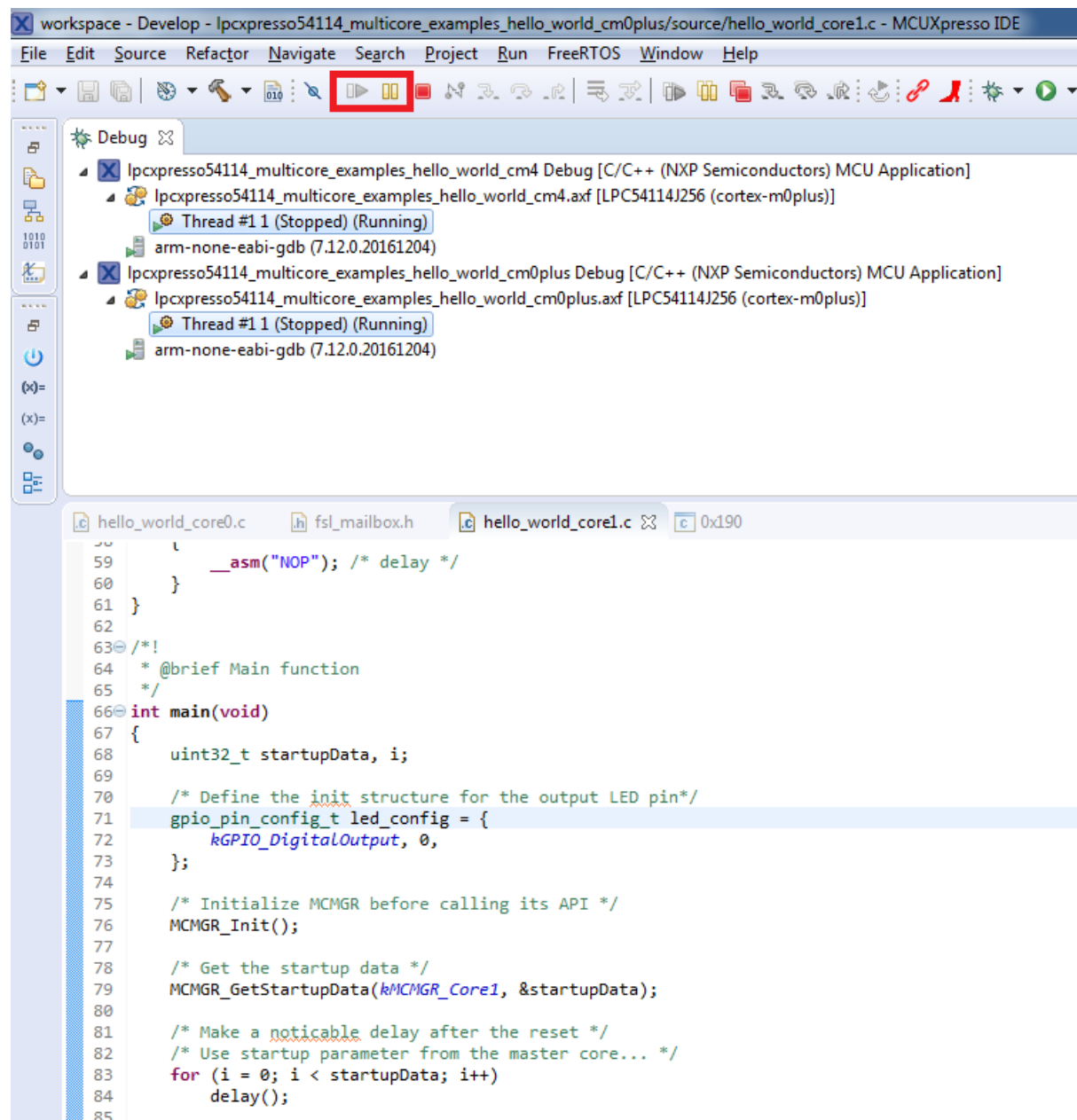


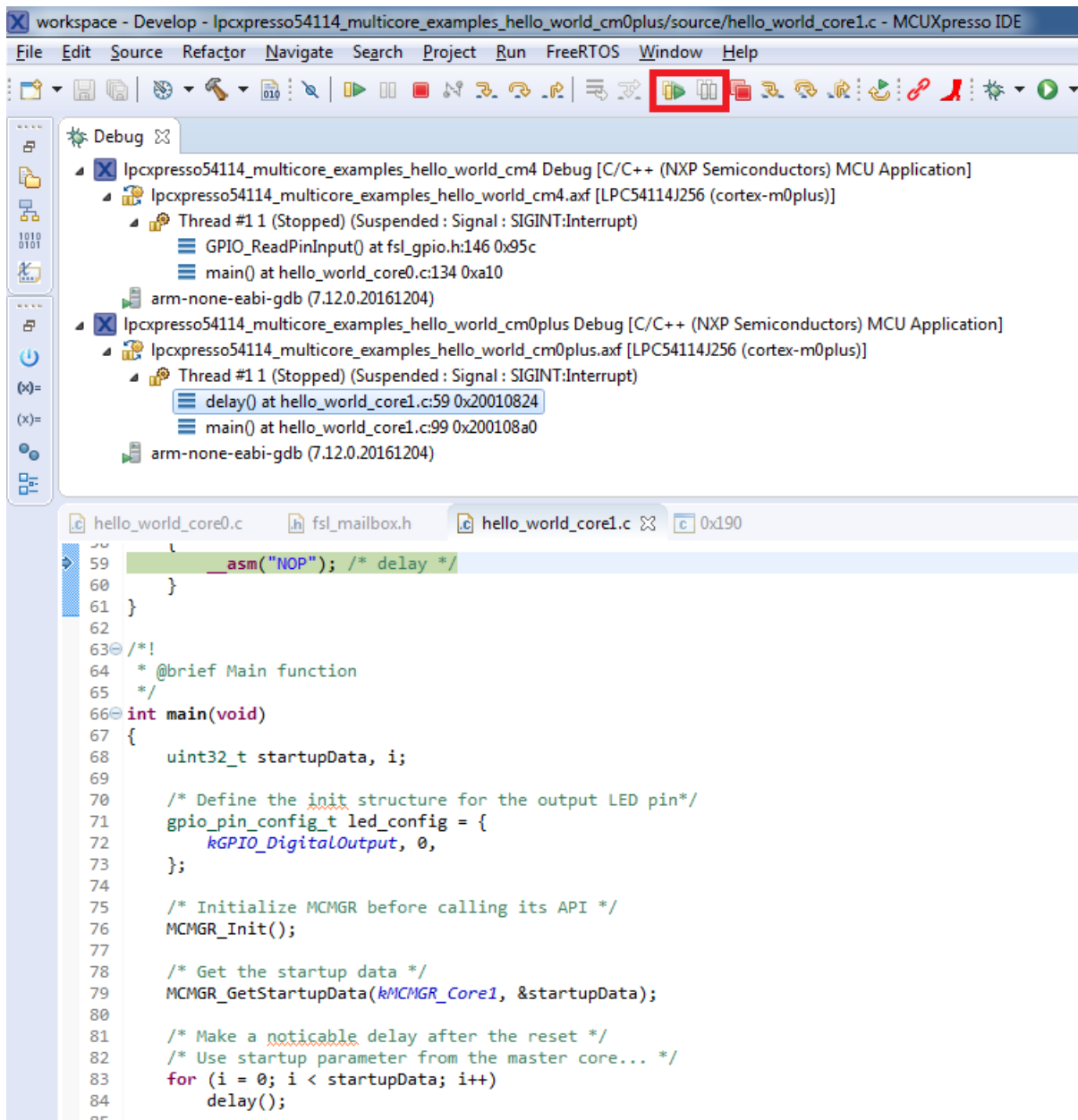


Now, the two debug sessions should be opened, and the debug controls can be used for both debug sessions depending on the debug session selection. Keep the primary core debug session selected by clicking the “Resume” button. The hello_world multicore application then starts running. The primary core application starts the auxiliary core application during runtime, and the auxiliary core application stops at the beginning of the main() function. The debug session of the auxiliary core application is highlighted. After clicking the “Resume” button, it is applied to the auxiliary core debug session. Therefore, the auxiliary core application continues its execution.



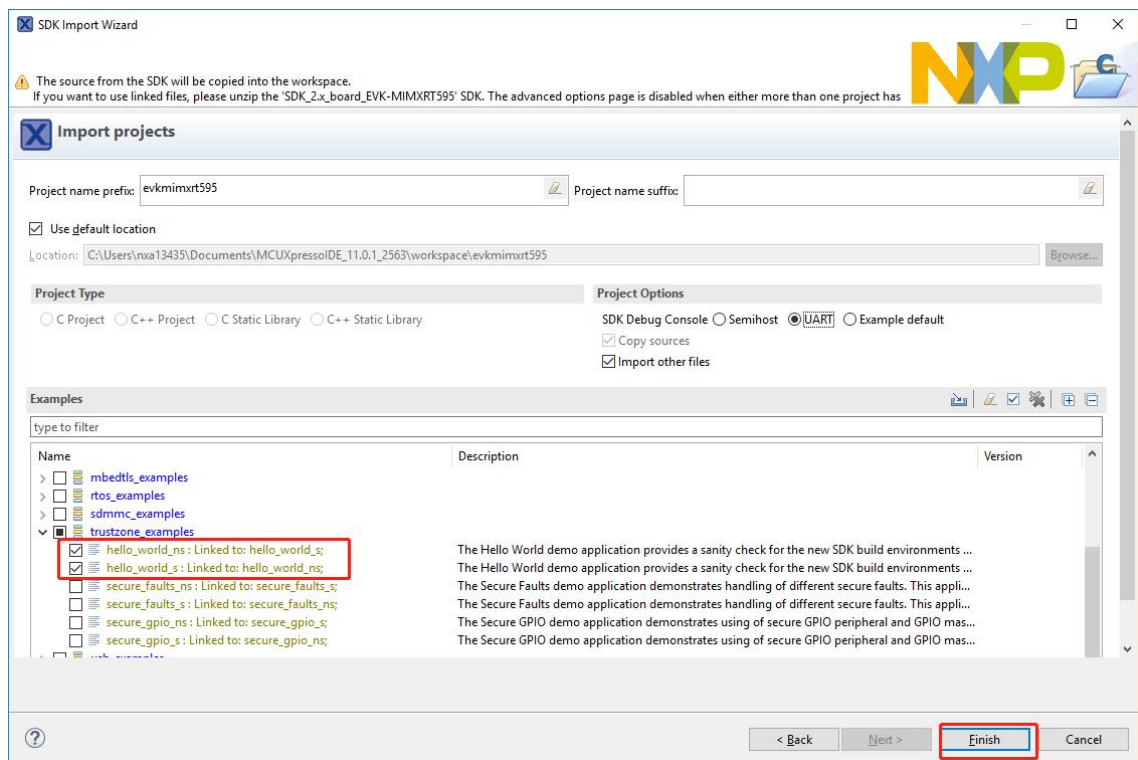
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the “Suspend” / “Resume” control button, or just using the “Suspend All Debug sessions” and the “Resume All Debug sessions” buttons.



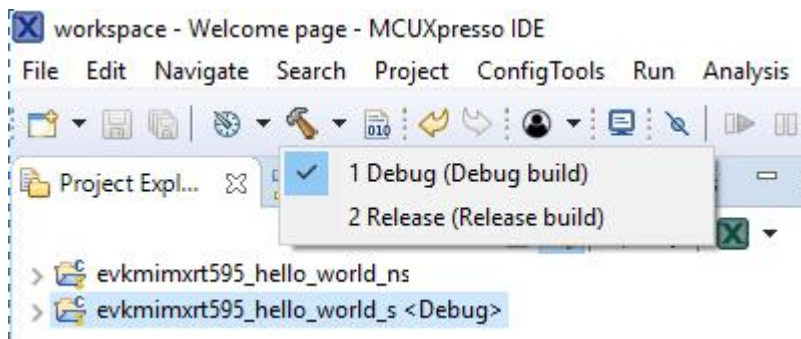


Build a TrustZone example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the `hello_world` example application targeted for the MIMXRT595-EVK hardware platform is used as an example, though these steps can be applied to any TrustZone example application in the MCUXpresso SDK.

1. TrustZone examples are imported into the workspace in a similar way as single core applications. When the SDK zip package for MIMXRT595-EVK is installed and available in the **Installed SDKs** view, click **Import SDK example(s)...** on the Quickstart Panel. In the window that appears, expand the **MIMXRT500** folder and select **MIMXRT595S**. Then, select **evkmimxrt595** and click **Next**.
2. Expand the `trustzone_examples/` folder and select `hello_world_s`. Because TrustZone examples are linked together, the non-secure project is automatically imported with the secure project, and there is no need to select it explicitly. Then, click **Finish**.

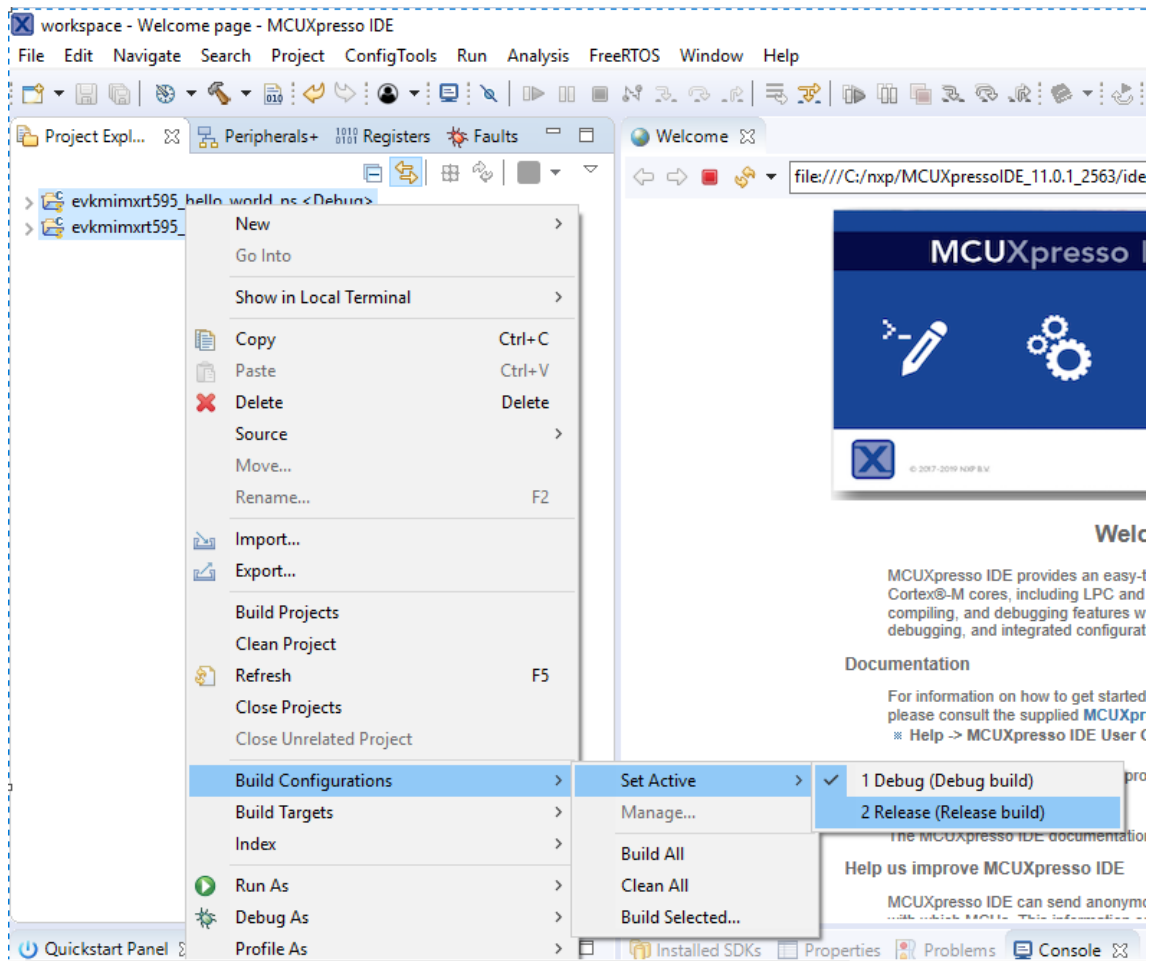


3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the `evkmimxrt595_hello_world_s` project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward facing arrow next to the hammer icon, as shown in following figure. For this example, select the **Debug** target.



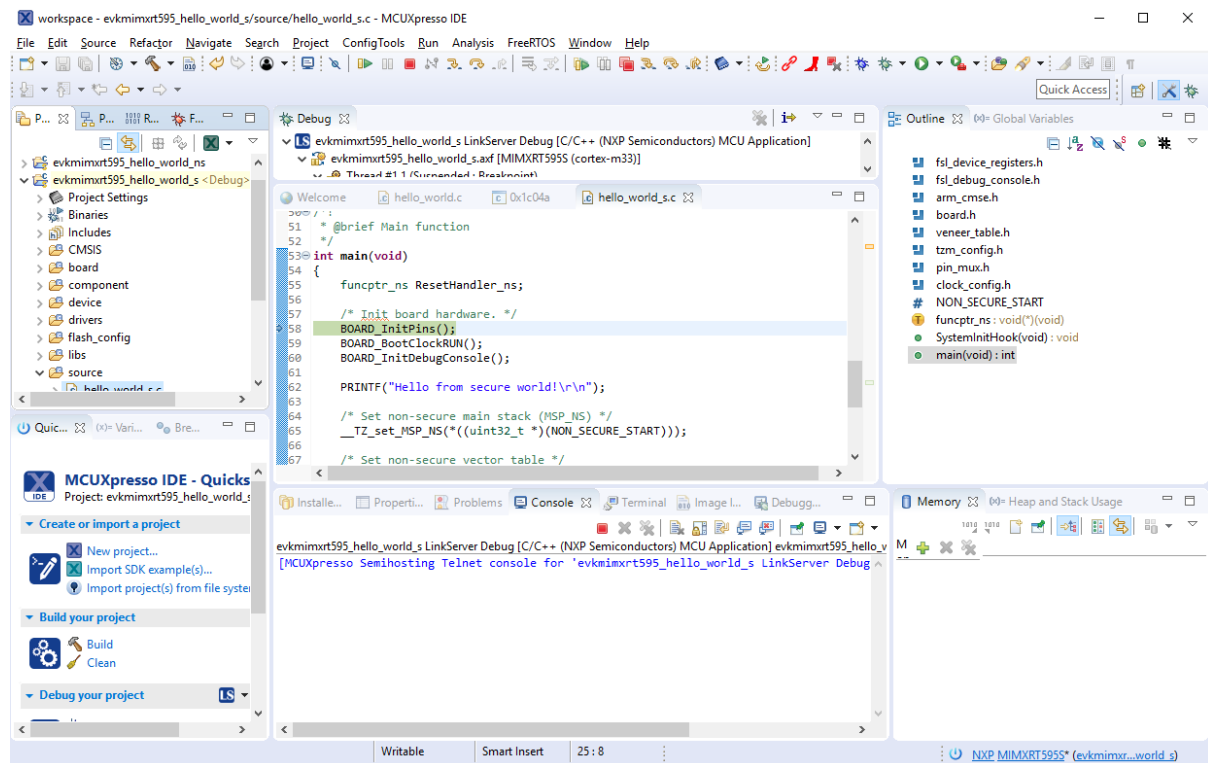
The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when the secure project since CMSE library is not ready.

Note: When the **Release** build is requested, it is necessary to change the build configuration of both the secure and non-secure application projects first. To do this, select both projects in the Project Explorer view by clicking to select the first project, then using shift-click or control-click to select the second project. Right click in the Project Explorer view to display the context-sensitive menu and select **Build Configurations > Set Active > Release**. This is also possible by using the menu item of **Project > Build Configuration > Set Active > Release**. After switching to the **Release** build configuration. Build the application for the secure project first.



Run a TrustZone example application To download and run the application, perform all steps as described in **Run an example application**. These steps are common for single core, and TrustZone applications, ensuring <board_name>_hello_world_s is selected for debugging.

In the Quickstart Panel, click **Debug** to launch the second debug session.



Now, the TrustZone sessions should be opened. Click **Resume**. The `hello_world` TrustZone application then starts running, and the secure application starts the non-secure application during runtime.

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 8.32.3 is used in the following example, and the IAR toolchain should correspond to the latest supported version, as described in the *MCUXpresso SDK Release Notes*.

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

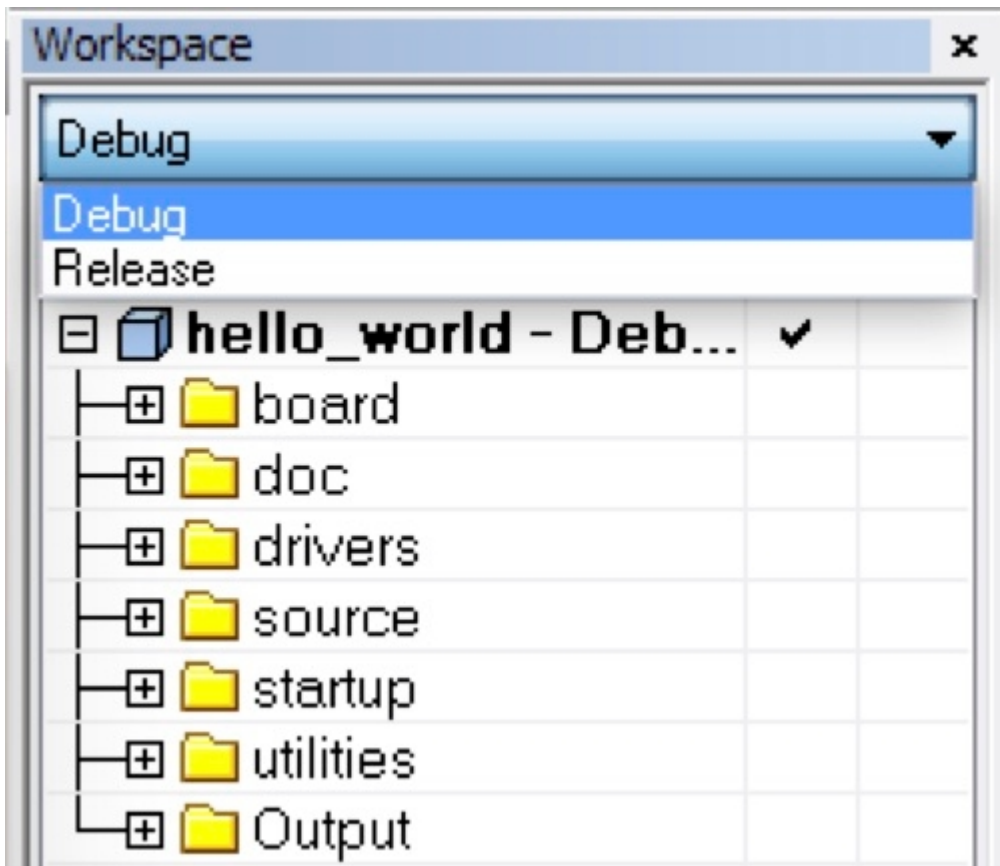
1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

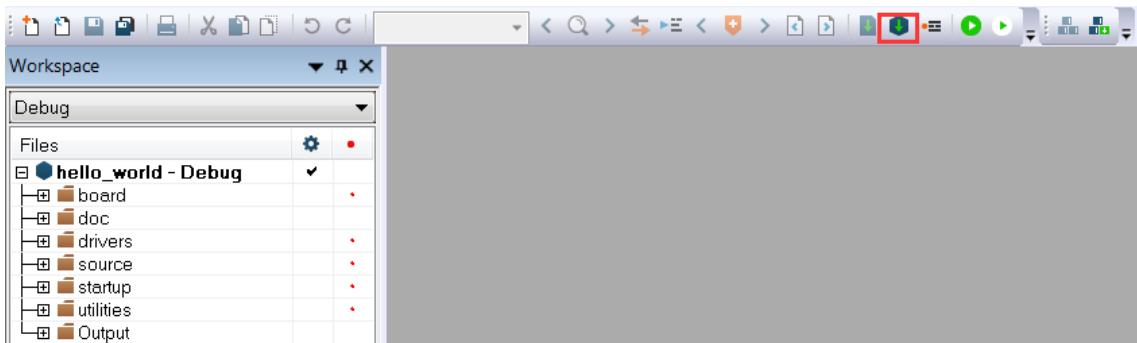
Other example applications may have additional folders in their path.

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

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



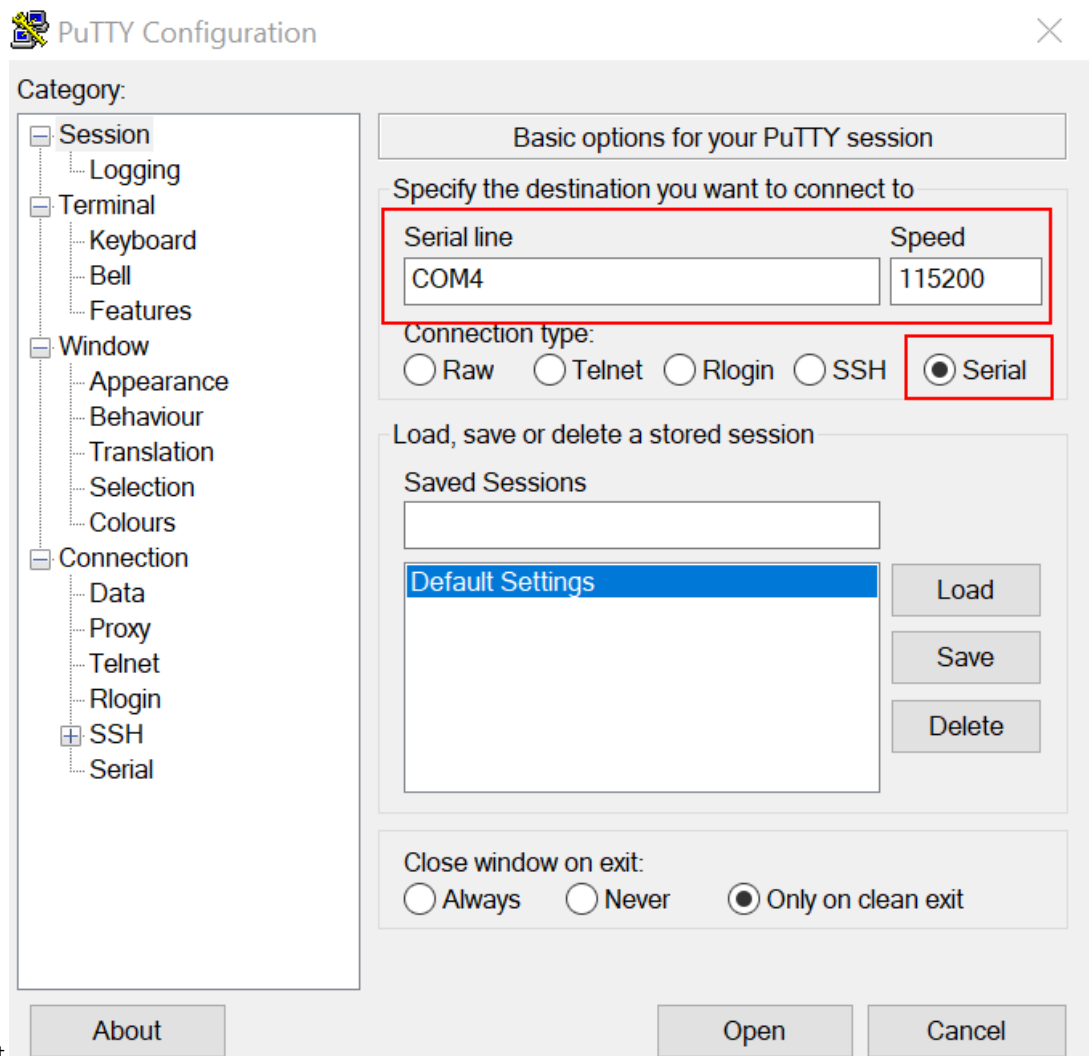
3. To build the demo application, click **Make**, highlighted in red in following figure.



4. The build completes without errors.

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

1. Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
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

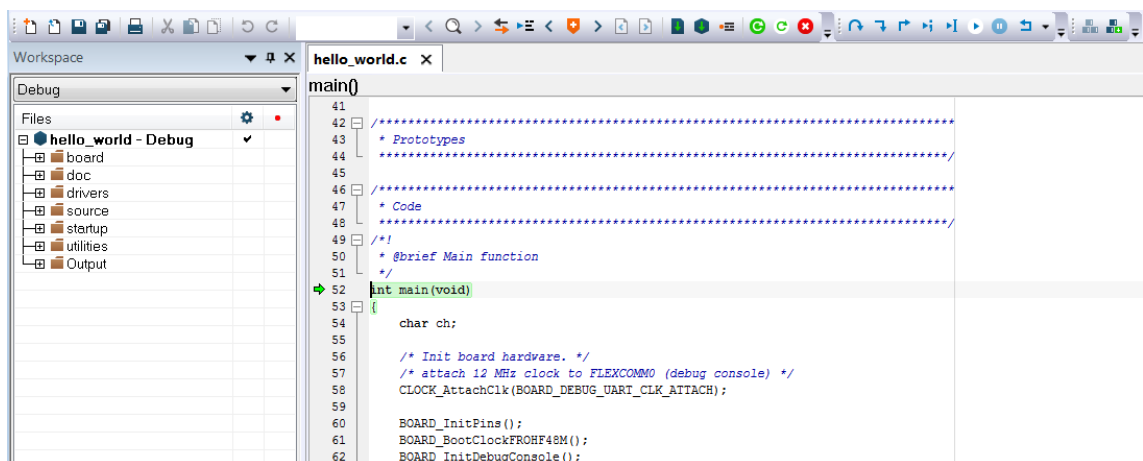


4. 1 stop bit

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.



Build a multicore example application This section describes the steps to build and run a dual-core application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/multicore_examples/<application_name>/<core_type>/iar
```

Begin with a simple dual-core version of the Hello World application. The multicore Hello World IAR workspaces are located in this folder:

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/iar/hello_world_cm0plus.  
↪ eww
```

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/iar/hello_world_cm4.eww
```

Build both applications separately by clicking the **Make** button. Build the application for the auxiliary core (cm0plus) first, because the primary core application project (cm4) must know the auxiliary core application binary when running the linker. It is not possible to finish the primary core linker when the auxiliary core application binary is not ready.

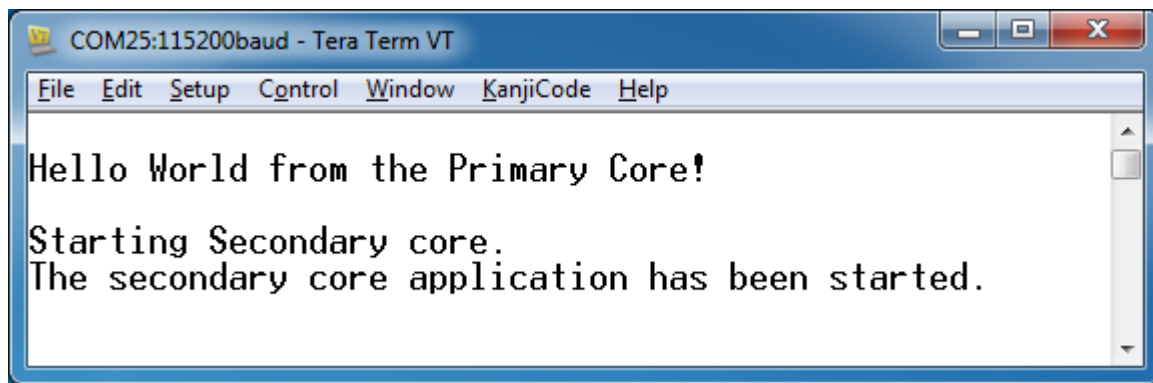
Run a multicore example application The primary core debugger handles flashing both primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 4 as described in **Run an example application**. These steps are common for both single core and dual-core applications in IAR.

After clicking the “Download and Debug” button, the auxiliary core project is opened in the separate EWARM instance. Both the primary and auxiliary images are loaded into the device flash memory and the primary core application is executed. It stops at the default C language entry point in the `*main()*function`.

Run both cores by clicking the “Start all cores” button to start the multicore application.



During the primary core code execution, the auxiliary core is released from the reset. The `hello_world` multicore application is now running and a banner is displayed on the terminal. If this does not appear, check the terminal settings and connections.



An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and is running correctly. When both cores are running, use the “Stop all cores”, and “Start all cores” control buttons to stop or run both cores simultaneously.

Build a TrustZone example application This section describes the particular steps that must be done in order to build and run a TrustZone application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/[<core_type>]/iar/  
↪<application_name>_ns/iar
```

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/[<core_type>]/iar/  
↪<application_name>_s/iar
```

Begin with a simple TrustZone version of the Hello World application. The TrustZone Hello World IAR workspaces are located in this folder:

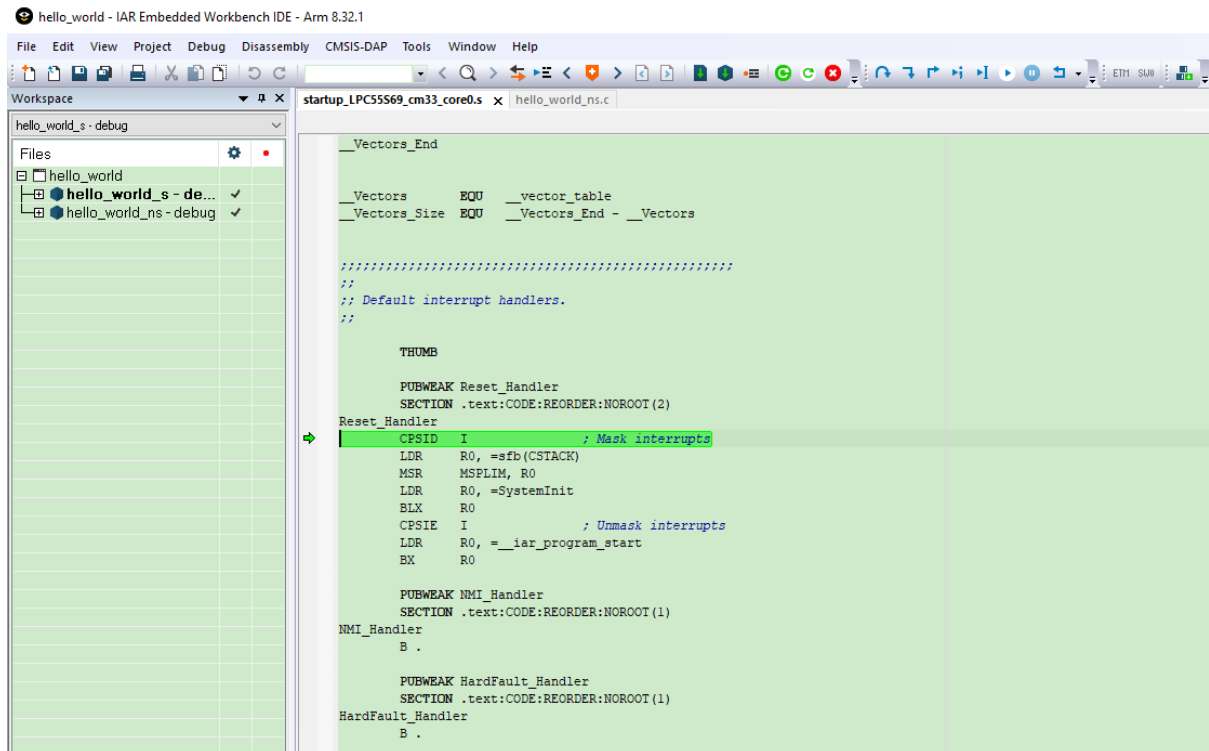
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/iar/hello_world_  
↪ns.eww
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/iar/hello_world_s.  
↪eww
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/iar/hello_world.eww
```

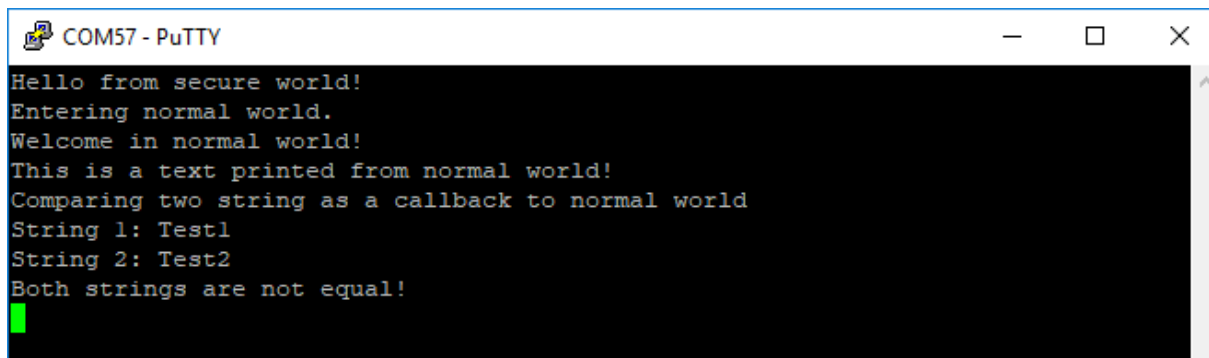
This project `hello_world.eww` contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another. Build both applications separately by clicking **Make**. It is requested to build the application for the secure project first, because the non-secure project must know the secure project, since the CMSE library is running the linker. It is not possible to finish the non-secure project linker with the secure project since CMSE library is not ready.

Run a TrustZone example application The secure project is configured to download both secure and non-secure output files, so debugging can be fully managed from the secure project. To download and run the TrustZone application, switch to the secure application project and perform steps 1 – 4 as described in **Run an example application**. These steps are common for both single core, and TrustZone applications in IAR. After clicking **Download and Debug**, both the secure and non-secure images are loaded into the device memory, and the secure application is executed. It stops at the `Reset_Handler` function.

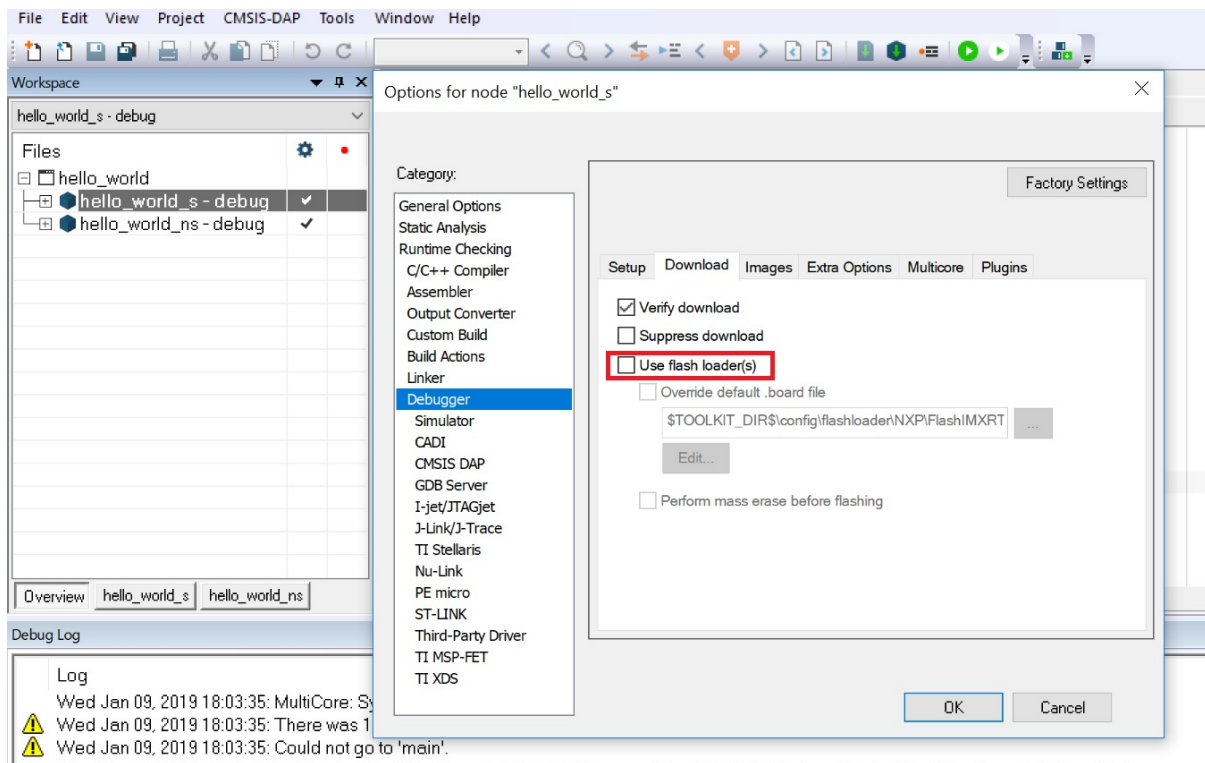


Run the code by clicking **Go** to start the application.

The TrustZone 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.



Note: If the application is running in RAM (debug/release build target), in **Options**>**Debugger > Download** tab, disable **Use flash loader(s)**. This can avoid the __ns download issue on i.MXRT500.

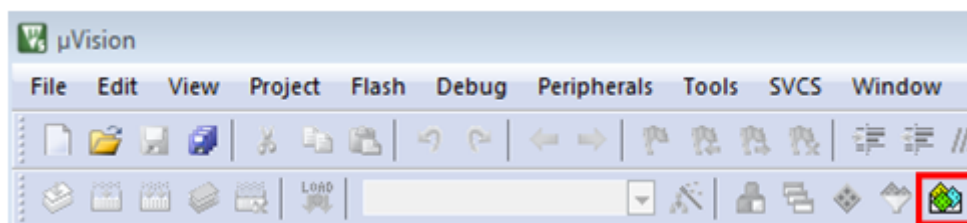


Run a demo using Keil MDK/μVision

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 appropriate CMSIS pack.

1. Open the MDK IDE, which is called μVision. In the IDE, select the **Pack Installer** icon.



2. After the installation finishes, close the Pack Installer window and return to the μVision IDE.

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:

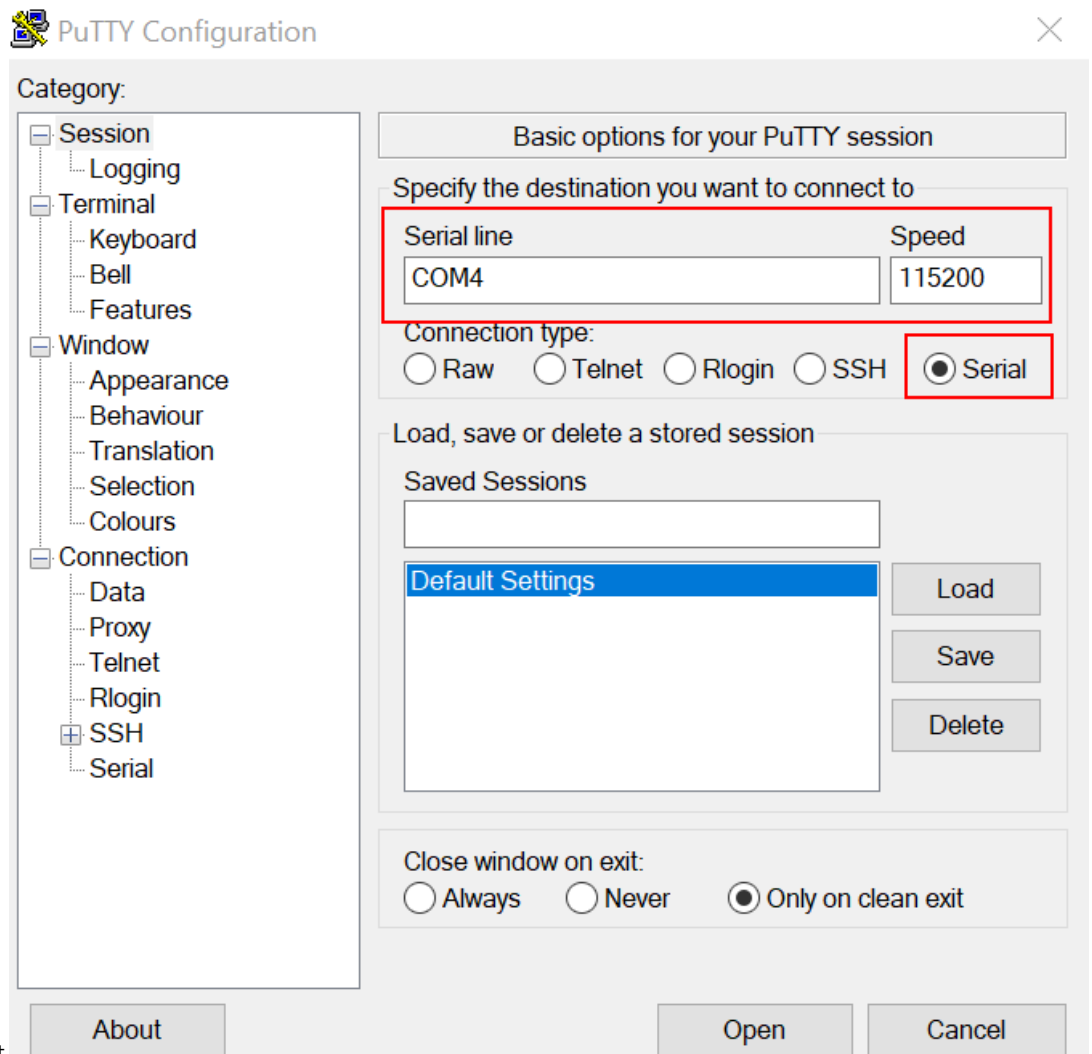
- To build the demo project, select **Rebuild**, highlighted in red.



- The build completes without errors.

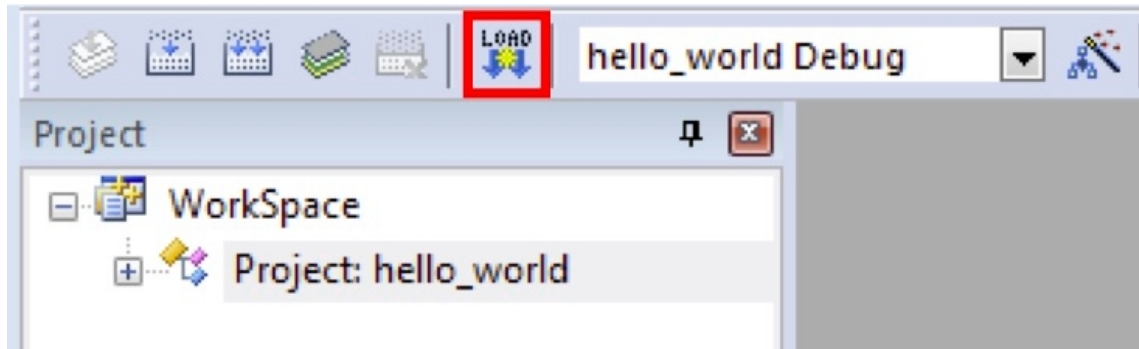
Run an example application To download and run the application, perform these steps:

- Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
- Connect the development platform to your PC via USB cable using USB connector.
- 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:
 - 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 - No parity
 - 8 data bits

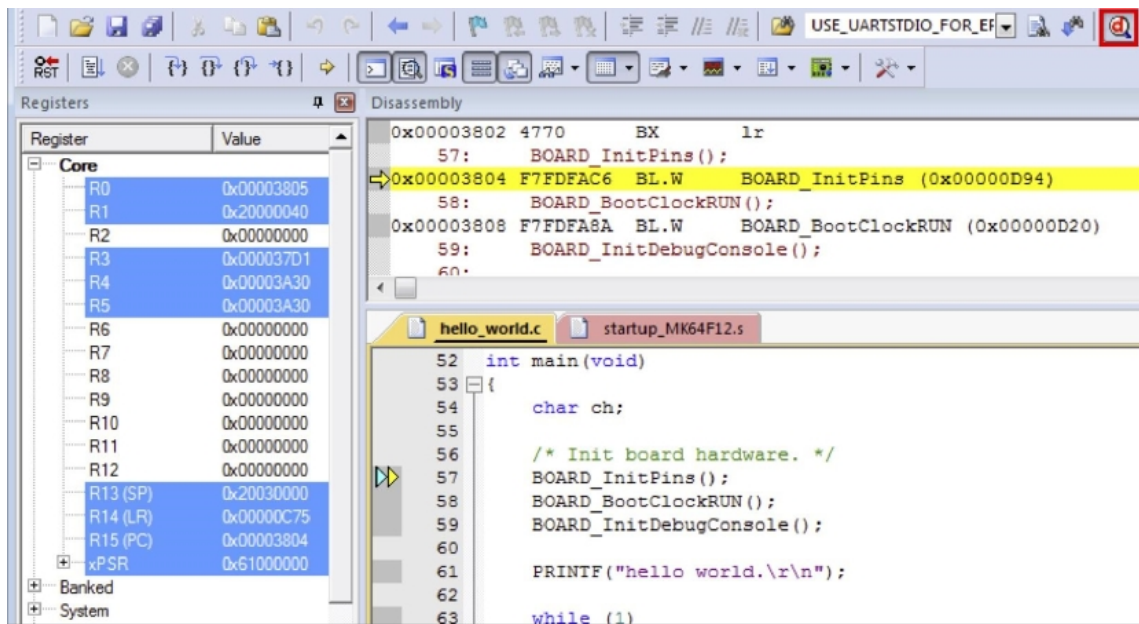


- 1 stop bit

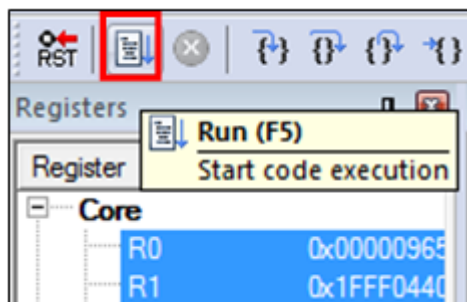
- In μ Vision, after the application is built, click the **Download** button to download the application to the target.



5. After clicking the **Download** button, the application downloads to the target and is running. To debug the application, click the **Start/Stop Debug Session** button, highlighted in red.



6. Run the code by clicking the **Run** button to start the 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.



Build a multicore example application This section describes the steps to build and run a dual-core application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/multicore_examples/<application_name>/<core_type>/mdk
```

Begin with a simple dual-core version of the Hello World application. The multicore Hello World Keil MSDK/μVision workspaces are located in this folder:

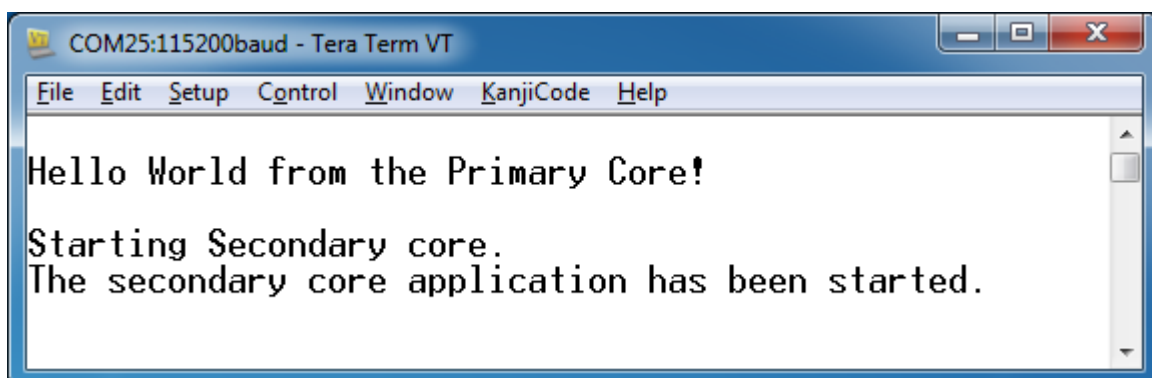
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/mdk/hello_world_
→cm0plus.uvmpw
```

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/mdk/hello_world_cm4.uvmpw
```

Build both applications separately by clicking the **Rebuild** button. Build the application for the auxiliary core (cm0plus) first because the primary core application project (cm4) must know the auxiliary core application binary when running the linker. It is not possible to finish the primary core linker when the auxiliary core application binary is not ready.

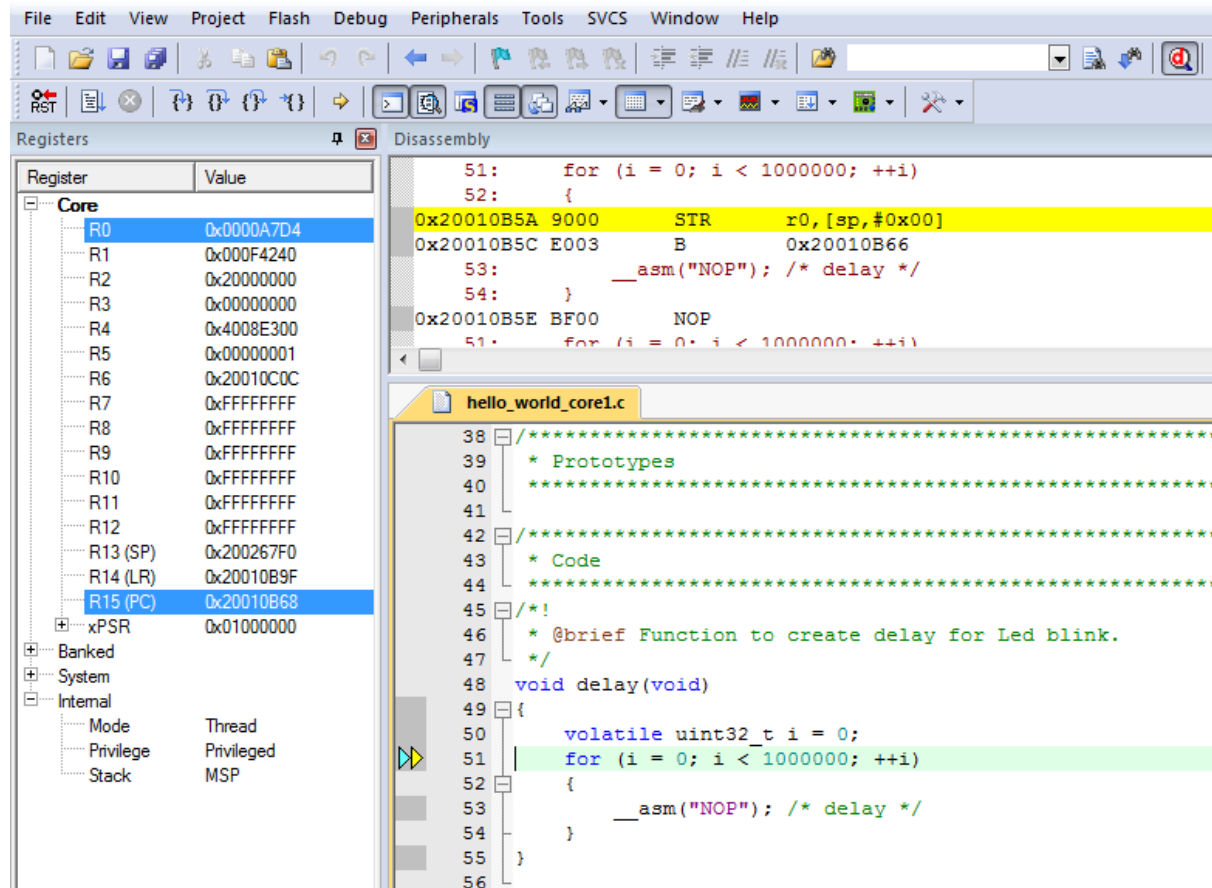
Run a multicore example application The primary core debugger flashes both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 5 as described in **Run an example application**. These steps are common for both single-core and dual-core applications in μVision.

Both the primary and the auxiliary image is loaded into the device flash memory. After clicking the “Run” button, the primary core application is executed. During the primary core code execution, the auxiliary core is released from the reset. The hello_world multicore application is now running and a banner is displayed on the terminal. If this does not appear, check your terminal settings and connections.



An LED controlled by the auxiliary core starts flashing indicating that the auxiliary core has been released from the reset and is running correctly.

Attach the running application of the auxiliary core by opening the auxiliary core project in the second μ Vision instance and clicking the “Start/Stop Debug Session” button. After this, the second debug session is opened and the auxiliary core application can be debugged.



Arm describes multicore debugging using the NXP LPC54114 Cortex-M4/M0+ dual-core processor and Keil uVision IDE in Application Note 318 at www.keil.com/appnotes/docs/apnt_318.asp. The associated video can be found [here](#).

Build a TrustZone example application This section describes the particular steps that must be done in order to build and run a TrustZone application. The demo applications workspace files are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/<application_name>_ns/
↪ mdk
```

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/<application_name>_s/
↪ mdk
```

Begin with a simple TrustZone version of the Hello World application. The TrustZone Hello World Keil MSDK/ μ Vision workspaces are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/mdk/hello_world_
↪ ns.uvmpw
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world_s.
↪ uvmpw
```



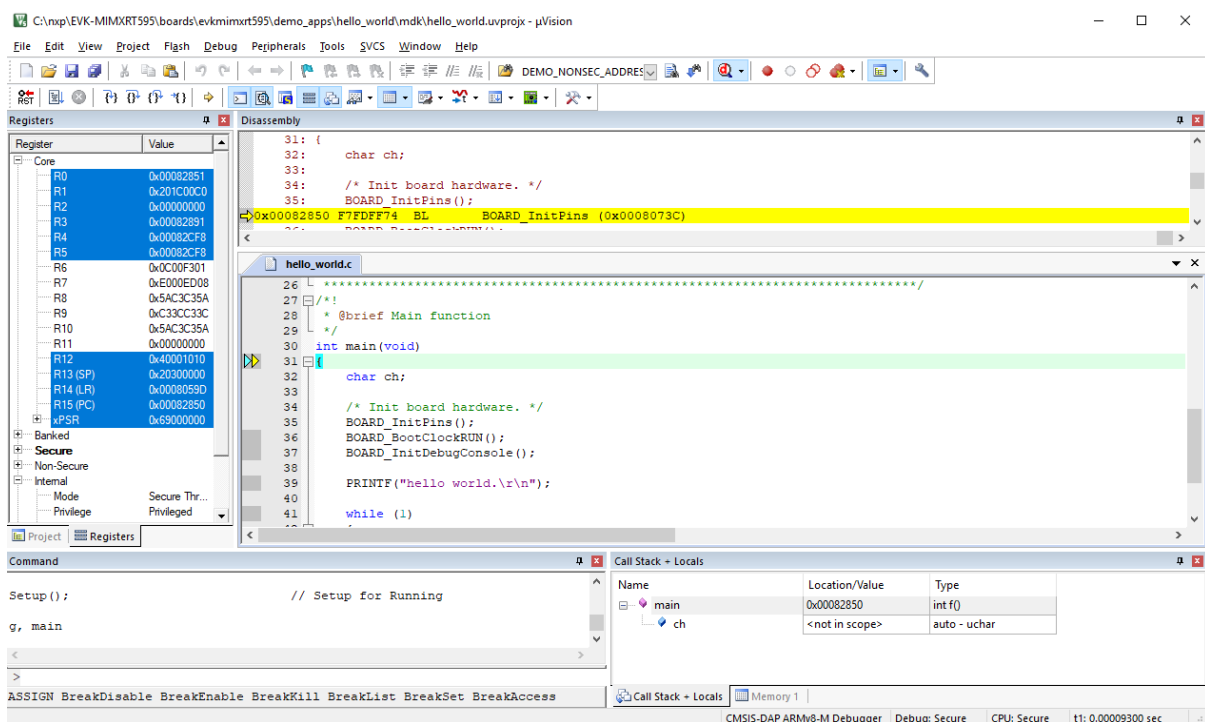
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world.  
→ uvmpw
```

This project `hello_world.uvmpw` contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another.

Build both applications separately by clicking **Rebuild**. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library is running the linker. It is not possible to finish the non-secure project linker with the secure project because CMSE library is not ready.

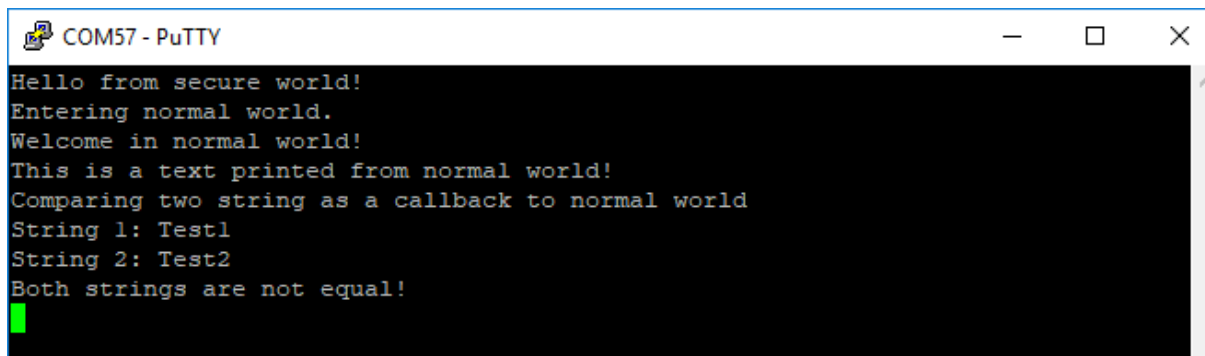
Run a TrustZone example application The secure project is configured to download both secure and non-secure output files so debugging can be fully managed from the secure project.

To download and run the TrustZone application, switch to the secure application project and perform steps as described in **Run an example application**. These steps are common for single core, dual-core, and TrustZone applications in μ Vision. After clicking **Download and Debug**, both the secure and non-secure images are loaded into the device flash memory, and the secure application is executed. It stops at the `main()` function.



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 not, check your terminal settings and connections.



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 which is used as an example.

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.

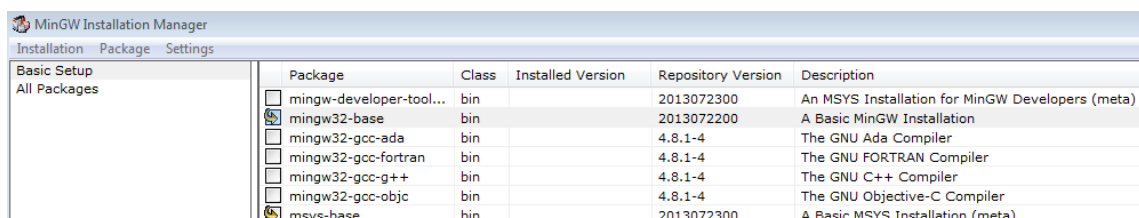
Install GCC Arm Embedded tool chain Download and run the installer from GNU Arm Embedded Toolchain. This is the actual toolset (in other words, compiler, linker, and so on). The GCC toolchain should correspond to the latest supported version, as described in **MCUXpresso SDK Release Notes**.

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.

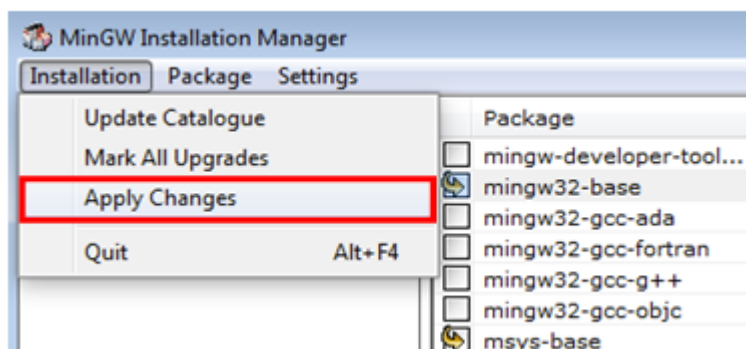
1. Download the latest MinGW mingw-get-setup installer from [MinGW](#).
2. Run the installer. The recommended installation path is `C:\MinGW`, however, you may install to any location.

Note: The installation path cannot contain any spaces.

3. 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.

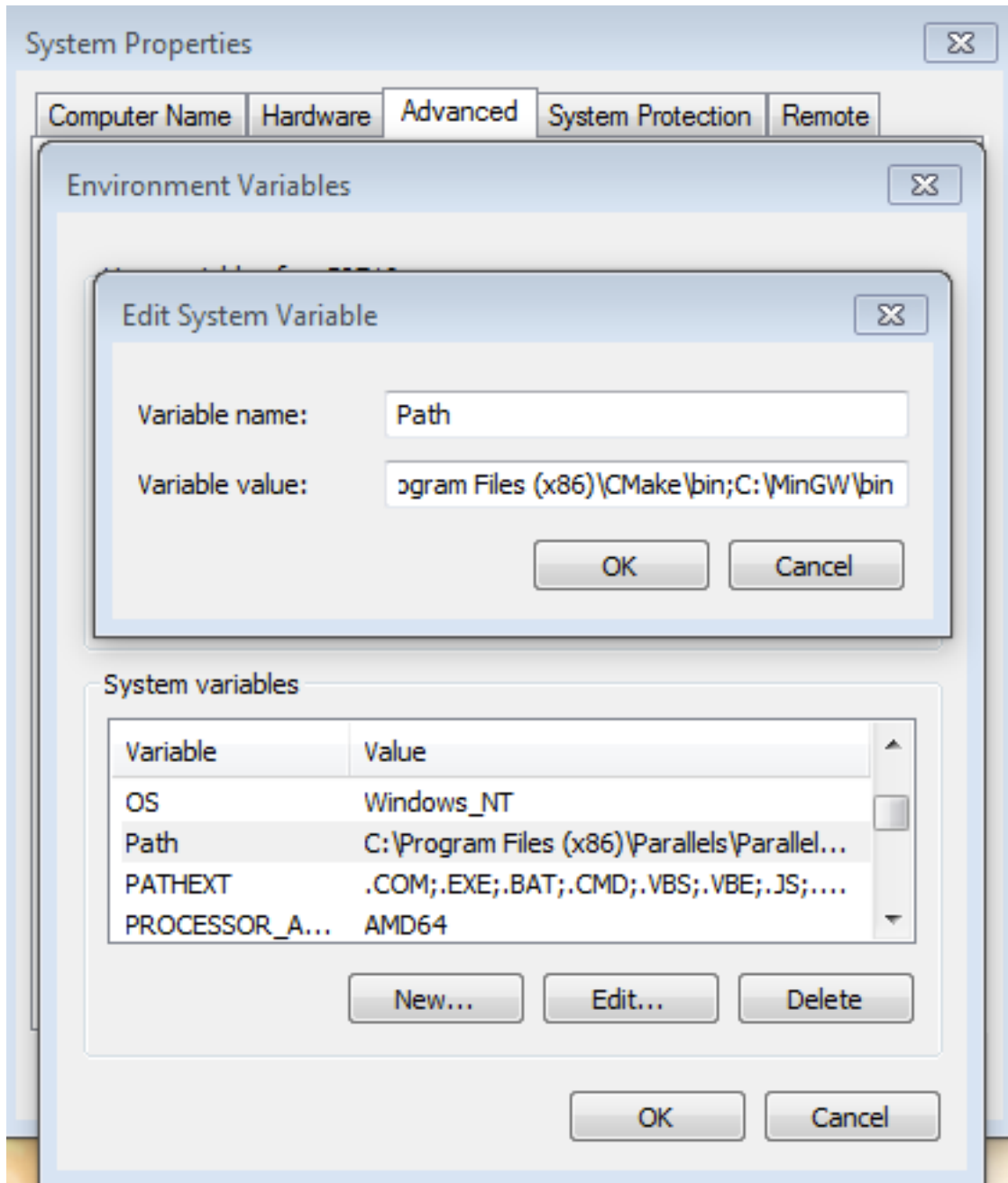


5. Add the appropriate item to the Windows operating system path environment variable. It can be found under **Control Panel->System and Security->System->Advanced System Settings** in the **Environment Variables...** section. The path is:

```
<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.



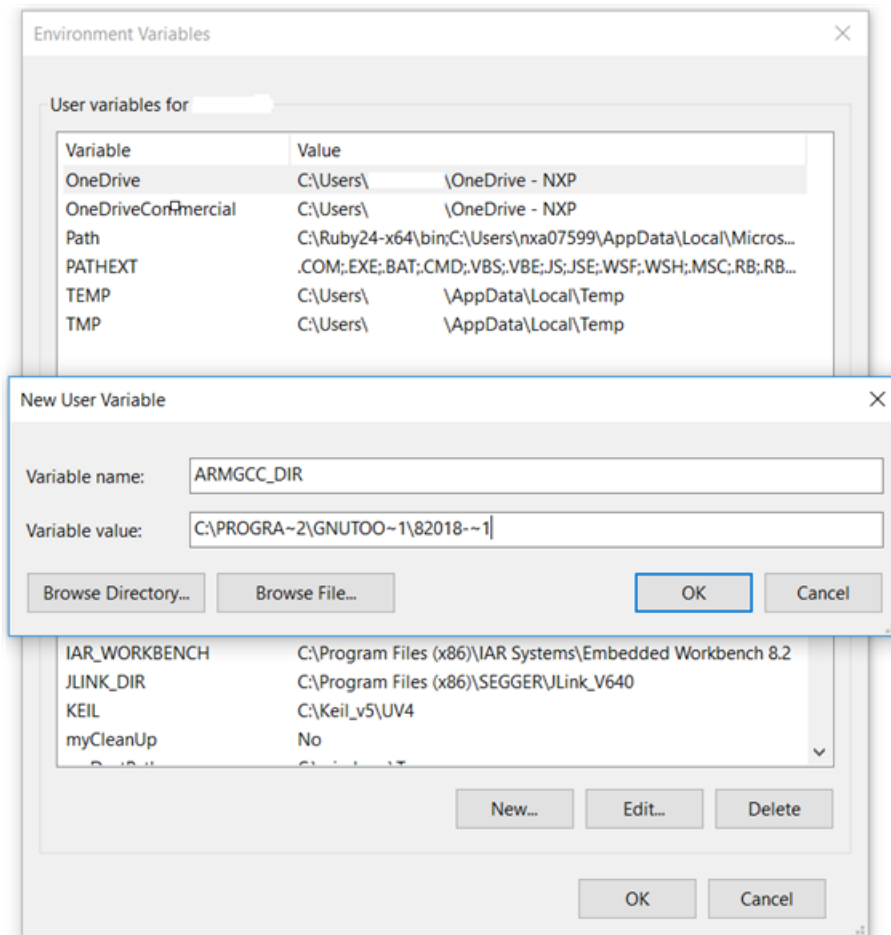
Add a new system environment variable for ARMGCC_DIR Create a new *system* environment variable and name it as ARMGCC_DIR. The value of this variable should point to the Arm GCC Embedded tool chain installation path. For this example, the path is:

```
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major
```

See the installation folder of the GNU Arm GCC Embedded tools for the exact pathname 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

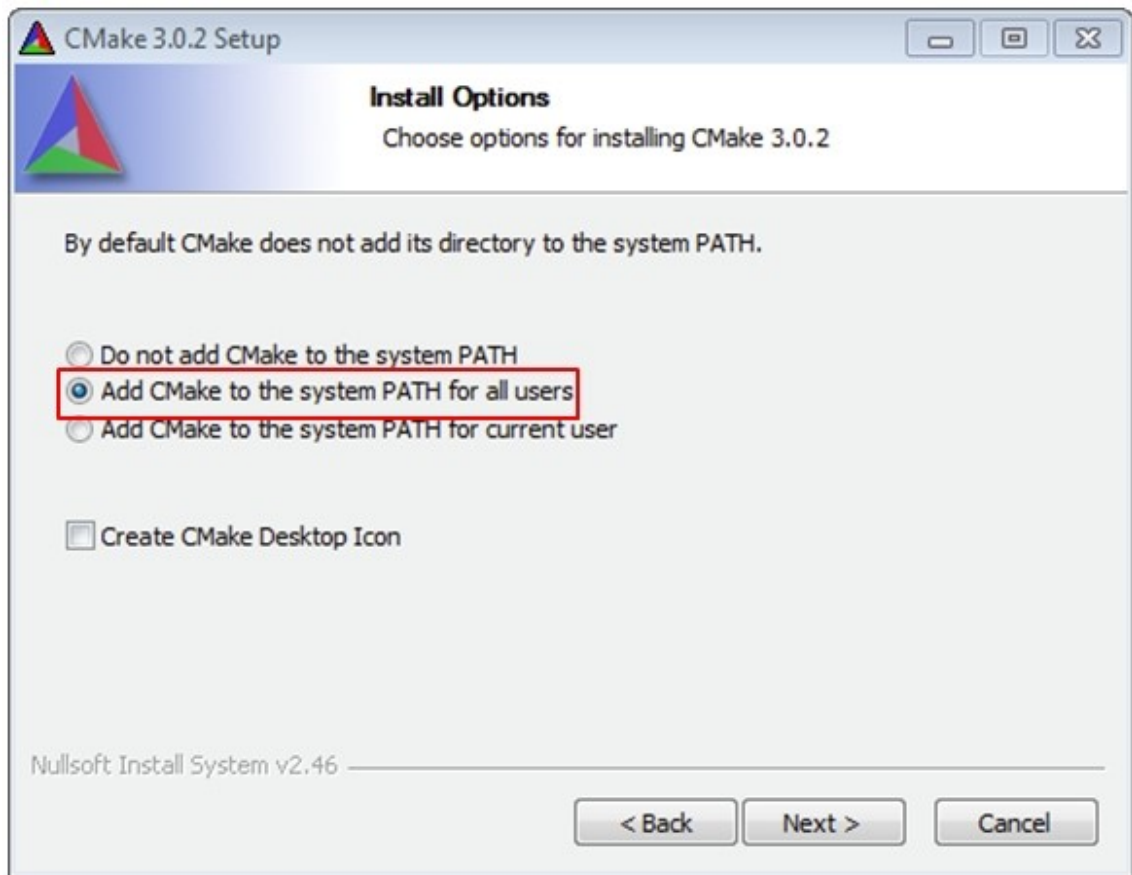
```
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>for %I in (.) do echo %~sI
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major>echo C:\PROGRA~2\GNUTOO~1\82018~1
C:\PROGRA~2\GNUTOO~1\82018~1
```



Install CMake

Windows OS

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.

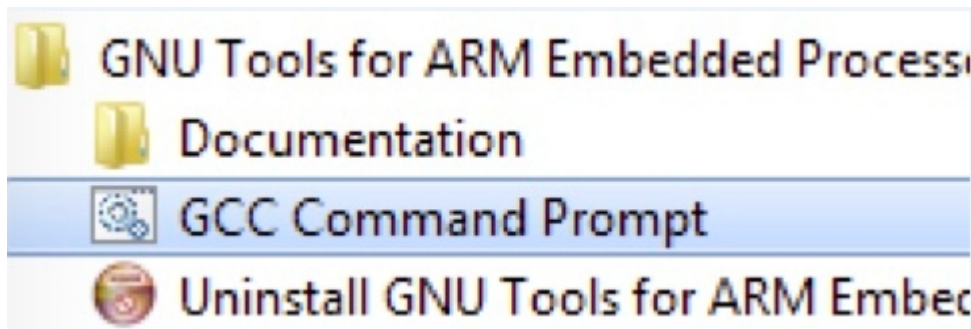
Linux OS It depends on the distributions of Linux Operation System. Here we use Ubuntu as an example.

Open shell and use following commands to install cmake and its version. Ensure the cmake version is above 3.0.x.

```
$ sudo apt-get install cmake
$ cmake --version
```

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:

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 following figure.

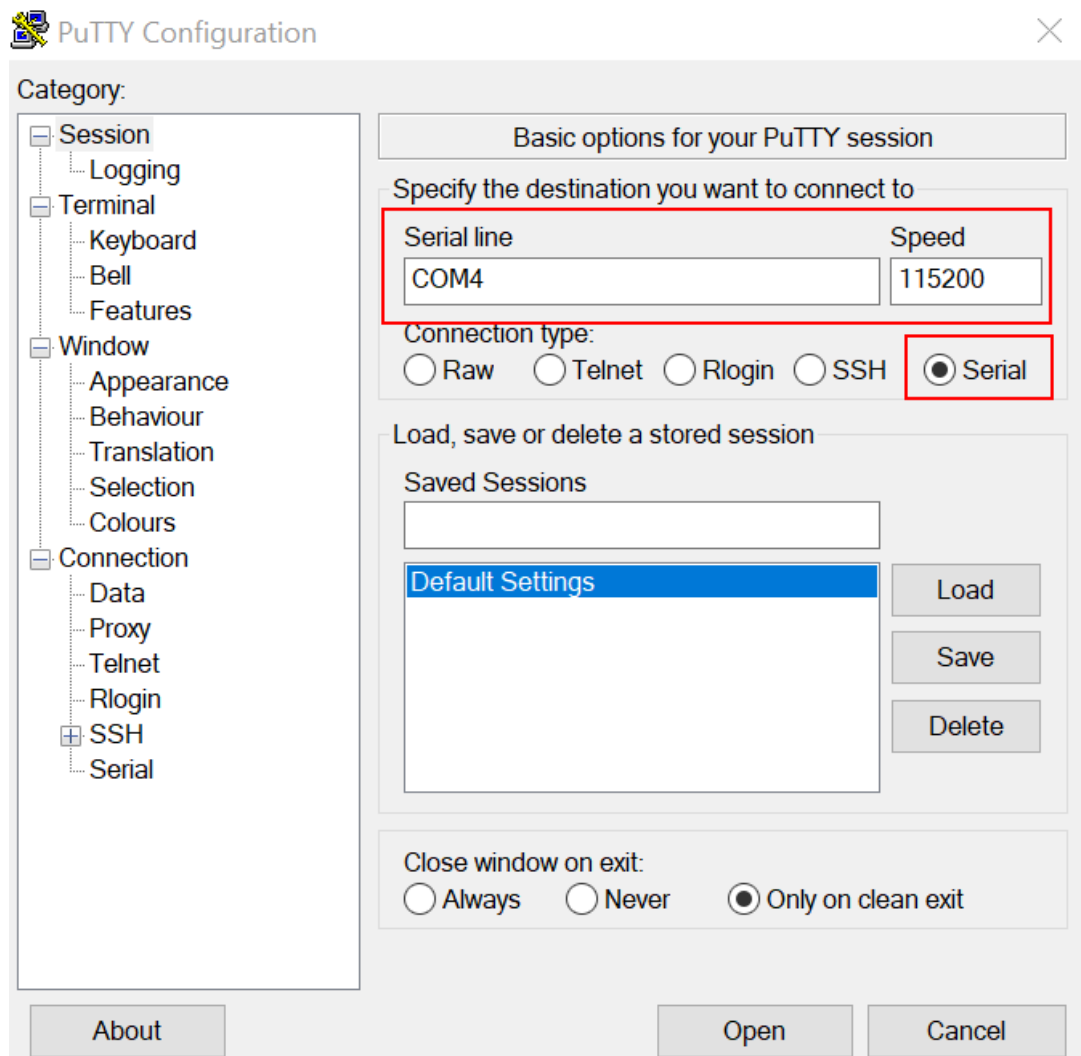
```
[ 84%] Building C object CMakeFiles/hello_world.elf.dir/C:/nxp/SDK_2.0_FRDM-K64F
/devices/MK64F12/drivers/fsl_smc.c.obj
[ 92%] Building C object CMakeFiles/hello_world.elf.dir/C:/nxp/SDK_2.0_FRDM-K64F
/devices/MK64F12/drivers/fsl_clock.c.obj
[100%] Linking C executable debug\hello_world.elf
[100%] Built target hello_world.elf

C:\nxp\SDK_2.0_FRDM-K64F\boards\frdmk64f\demo_apps\hello_world\armgcc>IF "" == "
" <pause >
Press any key to continue . . .
```

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

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 on-board debugger 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

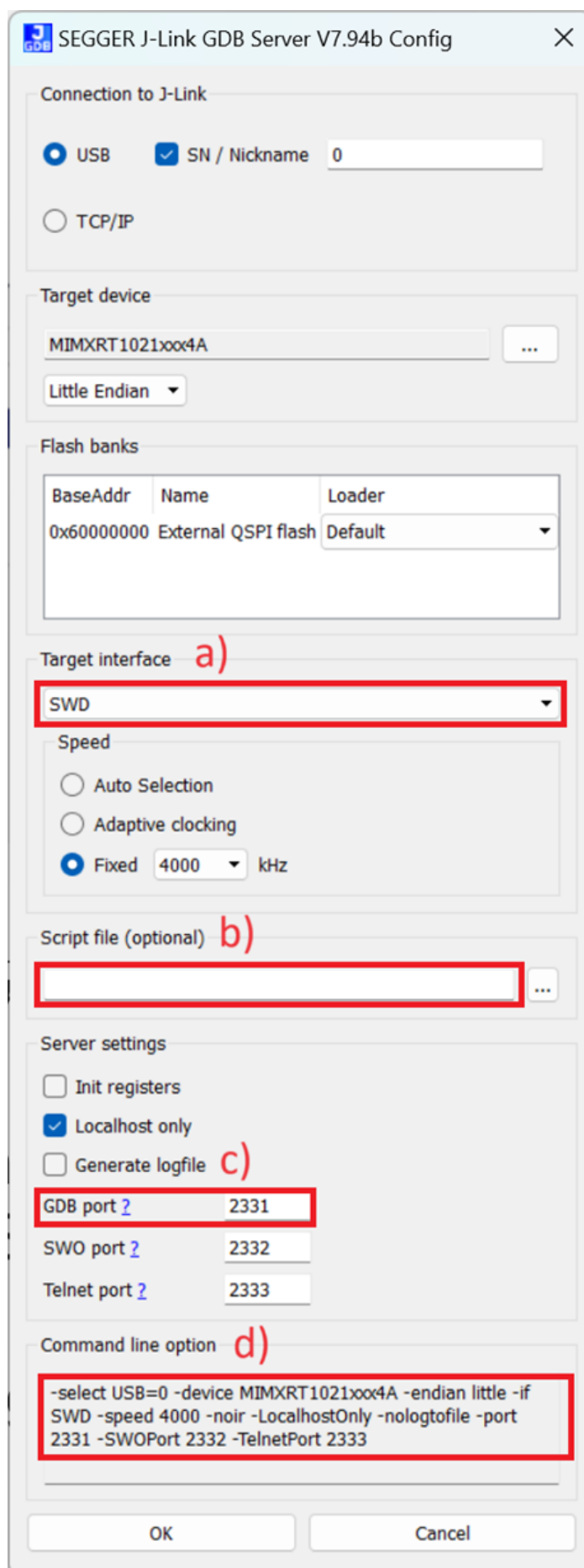


3. To launch the application, open the Windows **Start** menu and select **Programs > SEGGER > J-Link <version> J-Link GDB Server**.

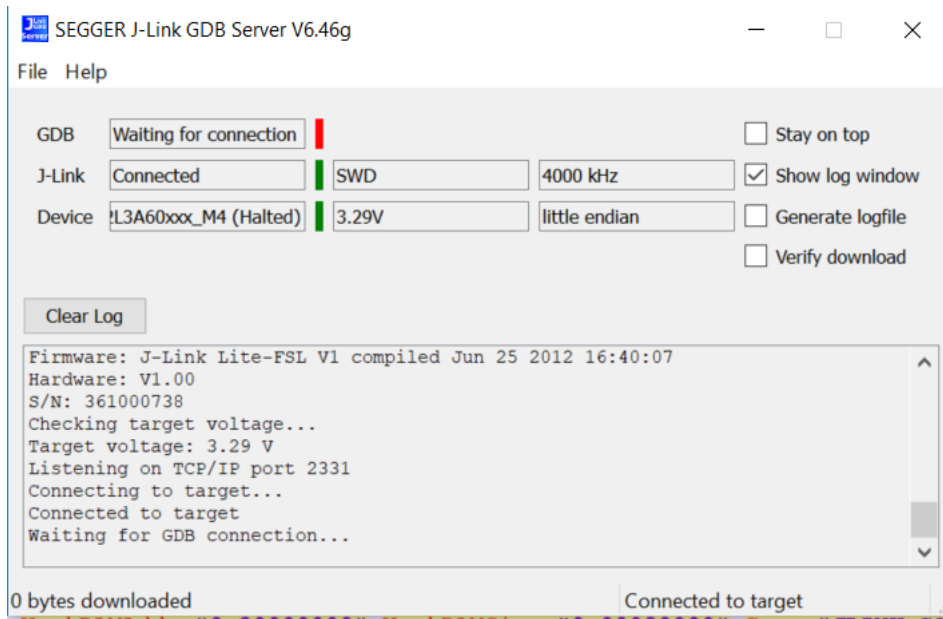
Note: It is assumed that the J-Link software is already installed.

The **SEGGER J-Link GDB Server Config** settings dialog appears.

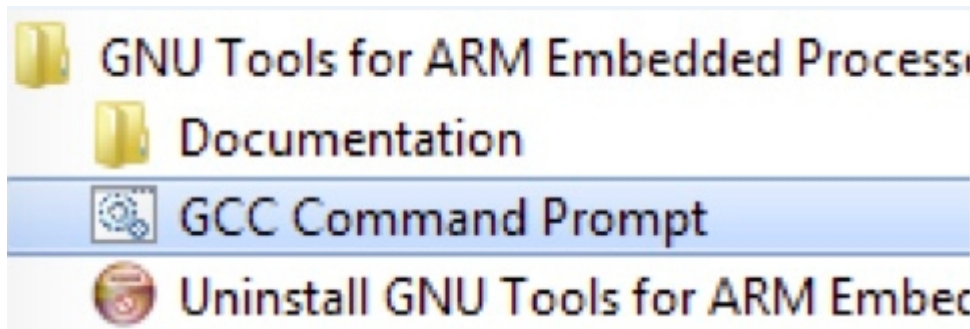
4. Make sure to check the following options.
 1. **Target interface:** The debug connection on board uses internal SWD signaling. In case of a wrong setting J-Link is unable to communicate with device under test.
 2. **Script file:** If required, a J-Link init script file can be used for board initialization. The file with the ".jlinkscript" file extension is located in the <install_dir>/boards/<board_name>/ directory.
 3. Under the **Server settings**, check the GDB port for connection with the gdb target remote command. For more information, see step 9.
 4. There is a command line version of J-Link GDB server "JLinkGDBServerCL.exe". Typical path is C:\Program Files\SEGGER\JLink\. To start the J-Link GDB server with the same settings as are selected in the UI, you can use these command line options.



5. After it is connected, the screen should look like this figure:



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
```

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`.

```

GCC Command Prompt - arm-none-eabi-gdb.exe C:\Users\nxa12829\Desktop\k3213\boards\frdmk3213a6\demo_apps\hello_world\cm4\armgcc\debug\hello_world_demo_cm4.elf
C:\Program Files (x86)\GNU Tools ARM Embedded\8 2018-q4-major>arm-none-eabi-gdb.exe C:\Users\nxa12829\Desktop\k3213\boards\frdmk3213a6\demo_apps\hello_world\cm4\armgcc\debug\hello_world_demo_cm4.elf
C:\Program Files (x86)\GNU Tools ARM Embedded\8 2018-q4-major\bin>arm-none-eabi-gdb.exe: warning: Couldn't determine a path for the index cache directory.
GNU gdb (GNU Tools for Arm Embedded Processors 8-2018-q4-major) 8.2.50.20181213-git
Copyright (C) 2018 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:
<http://www.gnu.org/software/gdb/bugs/>.
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 C:\Users\nxa12829\Desktop\k3213\boards\frdmk3213a6\demo_apps\hello_world\cm4\armgcc\debug\hello_world_demo_cm4.elf...
(gdb)

```

9. Run these commands:

1. target remote localhost:2331
2. monitor reset
3. monitor halt
4. load
5. monitor reset

10. The application is now downloaded and halted. 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.

```

Build a multicore example application This section describes the steps to build and run a dual-core application. The demo application build scripts are located in this folder:

```
<install_dir>/boards/<board_name>/multicore_examples/<application_name>/<core_type>/armgcc
```

Begin with a simple dual-core version of the Hello World application. The multicore Hello World GCC build scripts are located in this folder:

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/armgcc/build_debug.bat
```

```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/armgcc/build_debug.bat
```

Build both applications separately following steps for single core examples as described in **Build an example application**.

```

GCC Command Prompt - build_debug.bat

[ 47%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/drivers/fsl_common.c.obj
[ 52%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/drivers/fsl_msmc.c.obj
[ 56%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/utilities/debug_console/fsl_debug_console.c.obj
[ 60%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/utilities/fsl_assert.c.obj
[ 65%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/utilities/str/fsl_str.c.obj
[ 69%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/uart/lpuart_adapter.c.obj
[ 73%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/serial_manager/serial_manager.c.obj
[ 78%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/serial_manager/serial_port_uart.c.obj
[ 82%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/lists/generic_list.c.obj
[ 86%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/system_K32L3A60_cm0plus.c.obj
[ 91%] Building ASM object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/gcc/startup_K32L3A60_cm0plus.S.obj
[ 95%] Building C object CMakeFiles/hello_world_cm0plus.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/middleware/multicore/mcmgr/src/mcmgr.c.obj
[100%] Linking C executable debug\hello_world_cm0plus.elf
[100%] Built target hello_world_cm0plus.elf

c:\packages\SDK_2.6.0_FRDM-K32L3A6_RC1\boards\frdmk32l3a6\multicore_examples\hello_world\cm0plus\armgcc>IF "" == "" (pause)
Press any key to continue . . .

GCC Command Prompt - build_debug.bat

[ 50%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/drivers/fsl_lpuart.c.obj
[ 54%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/drivers/fsl_common.c.obj
[ 58%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/drivers/fsl_msmc.c.obj
[ 62%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/utilities/str/fsl_str.c.obj
[ 66%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/utilities/fsl_assert.c.obj
[ 70%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/utilities/debug_console/fsl_debug_console.c.obj
[ 75%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/uart/lpuart_adapter.c.obj
[ 79%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/serial_manager/serial_port_uart.c.obj
[ 83%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/serial_manager/serial_manager.c.obj
[ 87%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/components/lists/generic_list.c.obj
[ 91%] Building C object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/system_K32L3A60_cm4.c.obj
[ 95%] Building ASM object CMakeFiles/hello_world_cm4.elf.dir/C:/packages/SDK_2.6.0_FRDM-K32L3A6_RC1/devices/K32L3A60/gcc/startup_K32L3A60_cm4.S.obj
[100%] Linking C executable debug\hello_world_cm4.elf
[100%] Built target hello_world_cm4.elf

c:\packages\SDK_2.6.0_FRDM-K32L3A6_RC1\boards\frdmk32l3a6\multicore_examples\hello_world\cm4\armgcc>IF "" == "" (pause)
Press any key to continue . . .

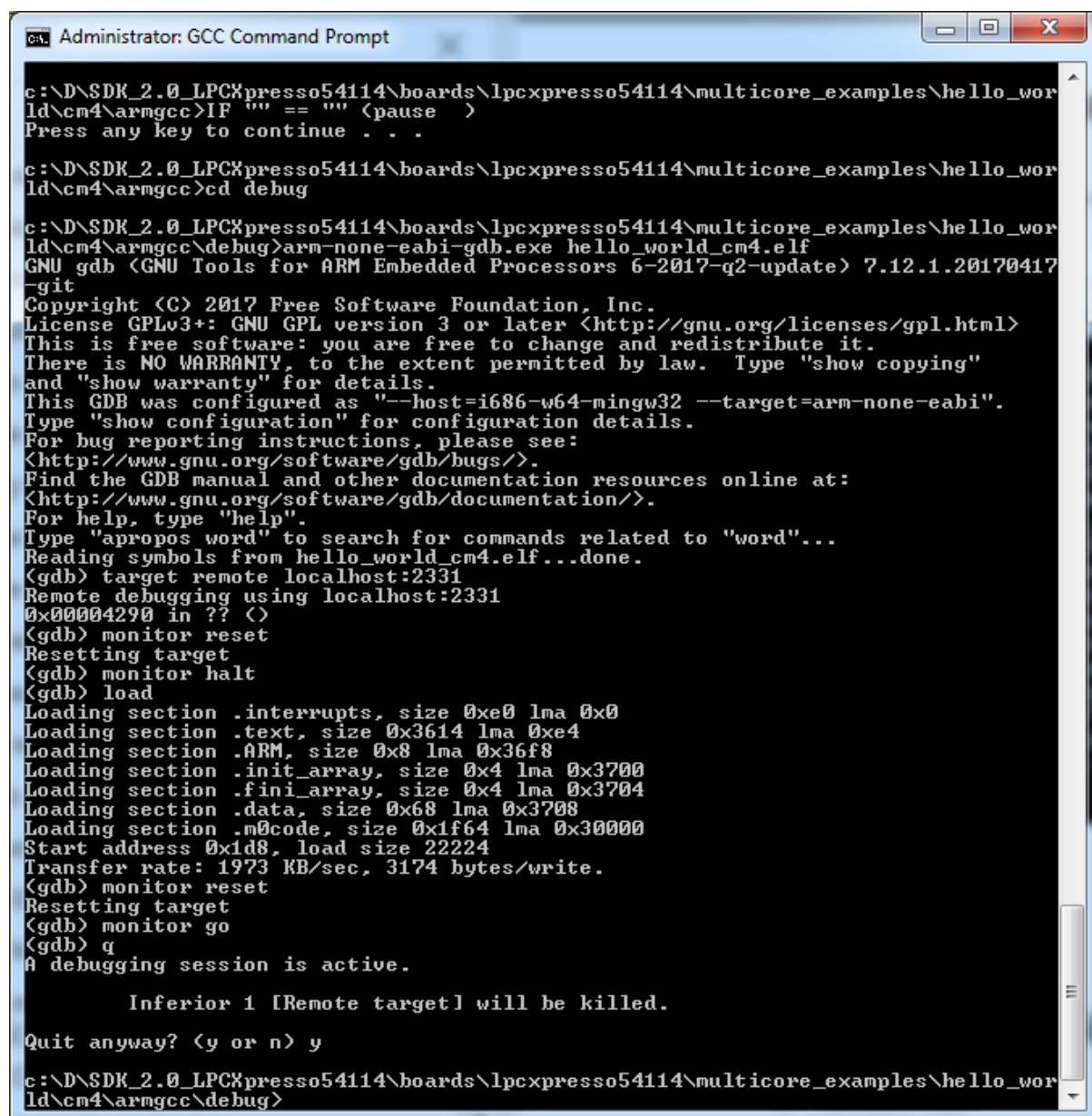
```

Run a multicore example application When running a multicore application, the same prerequisites for J-Link/J-Link OpenSDA firmware, and the serial console as for the single-core application, applies, as described in **Run an example application**.

The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 to 10, as described in **Run an example application**. These steps are common for both single-core and dual-core applications in Arm GCC.

Both the primary and the auxiliary image is loaded into the SPI flash memory. After execution of the monitor go command, the primary core application is executed. During the primary core code execution, the auxiliary core code is reallocated from the flash memory to the RAM, and the auxiliary core is released from the reset. The hello_world multicore application is now running

and a banner is displayed on the terminal. If this is not true, check your terminal settings and connections.



```

Administrator: GCC Command Prompt

c:\D\SDK_2.0_LPCXpresso54114\boards\lpcxpresso54114\multicore_examples\hello_world_cm4\armgcc>IF "" == "" <pause >
Press any key to continue . . .

c:\D\SDK_2.0_LPCXpresso54114\boards\lpcxpresso54114\multicore_examples\hello_world_cm4\armgcc>cd debug

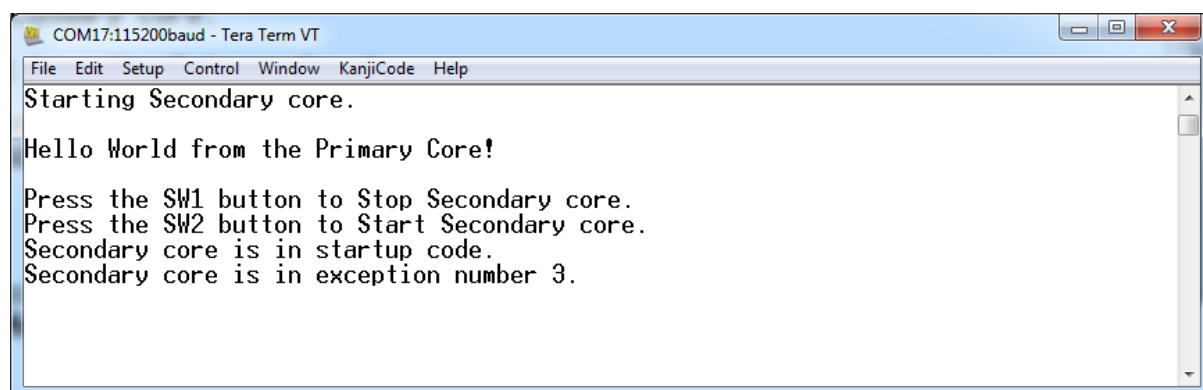
c:\D\SDK_2.0_LPCXpresso54114\boards\lpcxpresso54114\multicore_examples\hello_world_cm4\armgcc\debug>arm-none-eabi-gdb.exe hello_world_cm4.elf
GNU gdb (GNU Tools for ARM Embedded Processors 6-2017-q2-update) 7.12.1.20170417-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL 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:
<http://www.gnu.org/software/gdb/bugs/>.
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_cm4.elf...done.
(gdb) target remote localhost:2331
Remote debugging using localhost:2331
0x00004290 in ?? ()
(gdb) monitor reset
Resetting target
(gdb) monitor halt
(gdb) load
Loading section .interrupts, size 0xe0 lma 0x0
Loading section .text, size 0x3614 lma 0xe4
Loading section .ARM, size 0x8 lma 0x36f8
Loading section .init_array, size 0x4 lma 0x3700
Loading section .fini_array, size 0x4 lma 0x3704
Loading section .data, size 0x68 lma 0x3708
Loading section .m0code, size 0x1f64 lma 0x30000
Start address 0x1d8, load size 22224
Transfer rate: 1973 KB/sec, 3174 bytes/write.
(gdb) monitor reset
Resetting target
(gdb) monitor go
(gdb) q
A debugging session is active.

    Inferior 1 [Remote target] will be killed.

Quit anyway? (y or n) y

c:\D\SDK_2.0_LPCXpresso54114\boards\lpcxpresso54114\multicore_examples\hello_world_cm4\armgcc\debug>

```



```

COM17:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help
Starting Secondary core.

Hello World from the Primary Core!

Press the SW1 button to Stop Secondary core.
Press the SW2 button to Start Secondary core.
Secondary core is in startup code.
Secondary core is in exception number 3.

```

Build a TrustZone example application This section describes the steps to build and run a TrustZone application. The demo application build scripts are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/[<core_type>]/  
↪<application_name>_ns/armgcc
```

```
<install_dir>/boards/<board_name>/trustzone_examples/<application_name>/[<core_type>]/  
↪<application_name>_s/armgcc
```

Begin with a simple TrustZone version of the Hello World application. The TrustZone Hello World GCC build scripts are located in this folder:

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/armgcc/build_  
↪debug.bat
```

```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/armgcc/build_  
↪debug.bat
```

Build both applications separately, following steps for single core examples as described in **Build an example application**. It is requested to build the application for the secure project first, because the non-secure project must know the secure project, since CMSE library is running the linker. It is not possible to finish the non-secure project linker with the secure project because the CMSE library is not ready.

```
C:\WINDOWS\system32\cmd.exe
[ 55%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/utilities/fsl_
assert.c.obj
[ 59%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/uart/usart_adapter.c
.obj
[ 62%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_
flexspi.c.obj
[ 66%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_
cache.c.obj
[ 70%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/serial_manager/seria
l_manager.c.obj
[ 74%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/serial_manager/seria
l_port_uart.c.obj
[ 77%] Building ASM object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/gcc/startu
p_MIMXRT595S_cm33.S.obj
[ 81%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/lists/generic_list.c
.obj
[ 85%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_
usart.c.obj
[ 88%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_
flexcomm.c.obj
[ 92%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_
gpio.c.obj
[ 96%] Building C object CMakeFiles/hello_world_s.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_
iap.c.obj
[100%] Linking C executable debug\hello_world_s.elf
[100%] Built target hello_world_s.elf
C:\npx\SDK_2.6.0_EVK-MIMXRT595\boards\evkmimxrt595\trustzone_examples\hello_world\hello_world_s\armgcc>IF "" == "" (paus
e)
Press any key to continue . . .
```

```

C:\WINDOWS\system32\cmd.exe
[ 52%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/uart/usart_adapter.c.obj
[ 56%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/utilities/fsl_assert.c.obj
[ 60%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_flexspi.c.obj
[ 64%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_cache.c.obj
[ 68%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/serial_manager/serial_manager.c.obj
[ 72%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/serial_manager/serial_port_uart.c.obj
[ 76%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_usart.c.obj
[ 80%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/components/lists/generic_list.c.obj
[ 84%] Building ASM object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/gcc/startup_MIMXRT595S_cm33.S.obj
[ 88%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_flexcomm.c.obj
[ 92%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_gpio.c.obj
[ 96%] Building C object CMakeFiles/hello_world_ns.elf.dir/C:/npx/SDK_2.6.0_EVK-MIMXRT595/devices/MIMXRT595S/drivers/fsl_iap.c.obj
[100%] Linking C executable debug\hello_world_ns.elf
[100%] Built target hello_world_ns.elf

C:\npx\SDK_2.6.0_EVK-MIMXRT595\boards\evkmimxrt595\trustzone_examples\hello_world\hello_world_ns\armgcc>IF "" == "" (pause)
Press any key to continue . . .

```

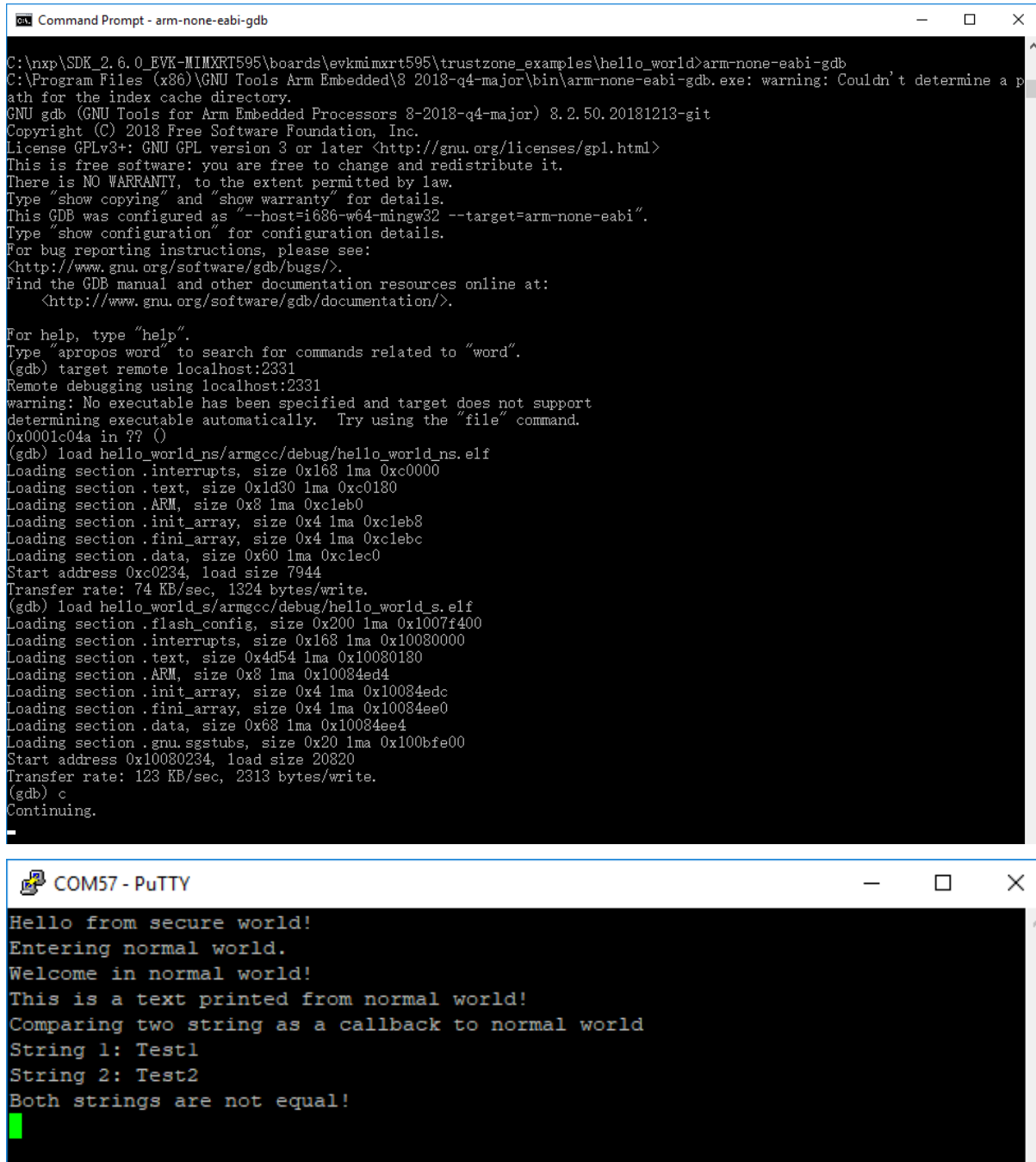
Run a TrustZone example application When running a TrustZone application, the same prerequisites for J-Link/J-Link OpenSDA firmware, and the serial console as for the single core application, apply, as described in **Run an example application**.

To download and run the TrustZone application, perform steps 1 to 10, as described in **Run an example application**. These steps are common for both single core and TrustZone applications in Arm GCC.

Then, run these commands:

1. arm-none-eabi-gdb.exe
2. target remote localhost:2331
3. monitor reset
4. monitor halt
5. monitor exec SetFlashDLNoRMWThreshold = 0x20000
6. load <install_dir>/boards/evkmimxrt595/trustzone_examples/hello_world/hello_world_ns/armgcc/debug/hello_world_ns.elf
7. load <install_dir>/boards/evkmimxrt595/trustzone_examples/hello_world/hello_world_s/armgcc/debug/hello_world_s.elf
8. monitor reset

The application is now downloaded and halted. Execute the `c` command to start the demo application.



```

C:\nxp\SDK_2.6.0_EVK-MIMXRT595\boards\evkmimxrt595\trustzone_examples\hello_world>arm-none-eabi-gdb
C:\Program Files (x86)\GNU Tools Arm Embedded\8 2018-q4-major\bin\arm-none-eabi-gdb.exe: warning: Couldn't determine a p
ath for the index cache directory.
GNU gdb (GNU Tools for Arm Embedded Processors 8-2018-q4-major) 8.2.50.20181213-git
Copyright (C) 2018 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:
<http://www.gnu.org/software/gdb/bugs/>.
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".
(gdb) target remote localhost:2331
Remote debugging using localhost:2331
warning: No executable has been specified and target does not support
determining executable automatically. Try using the "file" command.
0x0001c04a in ?? ()
(gdb) load hello_world_ns/armgcc/debug/hello_world_ns.elf
Loading section .interrupts, size 0x168 lma 0xc0000
Loading section .text, size 0x1d30 lma 0xc0180
Loading section .ARM, size 0x8 lma 0xc1eb0
Loading section .init_array, size 0x4 lma 0xc1eb8
Loading section .fini_array, size 0x4 lma 0xc1ebc
Loading section .data, size 0x60 lma 0xc1ec0
Start address 0xc0234, load size 7944
Transfer rate: 74 KB/sec, 1324 bytes/write.
(gdb) load hello_world_s/armgcc/debug/hello_world_s.elf
Loading section .flash_config, size 0x200 lma 0x1007f400
Loading section .interrupts, size 0x168 lma 0x10080000
Loading section .text, size 0x4d54 lma 0x10080180
Loading section .ARM, size 0x8 lma 0x10084ed4
Loading section .init_array, size 0x4 lma 0x10084edc
Loading section .fini_array, size 0x4 lma 0x10084ee0
Loading section .data, size 0x68 lma 0x10084ee4
Loading section .gnu.sgstubs, size 0x20 lma 0x100bfe00
Start address 0x10080234, load size 20820
Transfer rate: 123 KB/sec, 2313 bytes/write.
(gdb) c
Continuing.

```

```

COM57 - PuTTY
Hello from secure world!
Entering normal world.
Welcome in normal world!
This is a text printed from normal world!
Comparing two string as a callback to normal world
String 1: Test1
String 2: Test2
Both strings are not equal!

```

MCUXpresso Config Tools

MCUXpresso Config Tools can help configure the processor and generate initialization code for the on chip peripherals. The tools are able to modify any existing example project, or create a new configuration for the selected board or processor. The generated code is designed to be used with MCUXpresso SDK version 24.12.00 or later.

Following table describes the tools included in the MCUXpresso Config Tools.

Config Tool	Description	Image
Pins tool	For configuration of pin routing and pin electrical properties.	
Clock tool	For system clock configuration	
Peripheral tools	For configuration of other peripherals	
TEE tool	Configures access policies for memory area and peripherals helping to protect and isolate sensitive parts of the application.	
Device Configuration tool	Configures Device Configuration Data (DCD) contained in the program image that the Boot ROM code interprets to set up various on-chip peripherals prior to the program launch.	

MCUXpresso Config Tools can be accessed in the following products:

- **Integrated** in the MCUXpresso IDE. Config tools are integrated with both compiler and debugger which makes it the easiest way to begin the development.
- **Standalone version** available for download from www.nxp.com/mcuxpresso. Recommended for customers using IAR Embedded Workbench, Keil MDK μ Vision, or Arm GCC.
- **Online version** available on mcuxpresso.nxp.com. Recommended doing a quick evaluation of the processor or use the tool without installation.

Each version of the product contains a specific *Quick Start Guide* document MCUXpresso IDE Config Tools installation folder that can help start your work.

How to determine COM port

This section describes the steps necessary to determine the debug COM port number of your NXP hardware development platform. All NXP boards ship with a factory programmed, onboard debug interface, whether it is based on MCU-Link or the legacy OpenSDA, LPC-Link2, P&E Micro OSJTAG interface. To determine what your specific board ships with, see [Default debug interfaces](#).

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 for core0 debug console and the other is for core1.

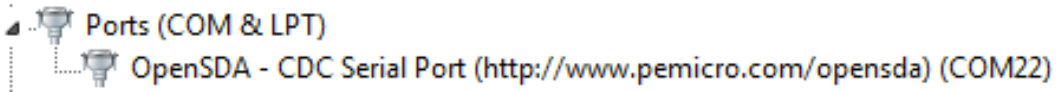
2. **Windows:** To determine the COM port open Device Manager in the Windows operating system. Click the **Start** menu and type **Device Manager** in the search bar.

In the Device Manager, expand the **Ports (COM & LPT)** section to view the available ports. The COM port names are different for all the NXP boards.

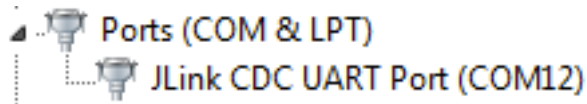
1. **CMSIS-DAP/mbd/DAPLink** interface:



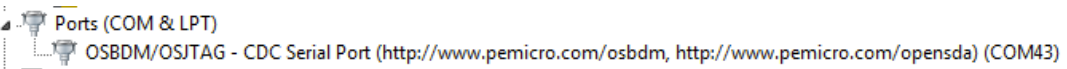
2. P&E Micro:



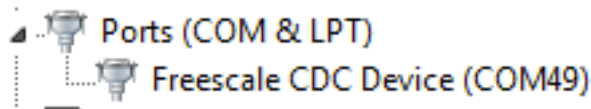
3. J-Link:



4. P&E Micro OSJTAG:



5. MRB-KW01:



On-board Debugger

This section describes the on-board debuggers used on NXP development boards.

On-board debugger MCU-Link MCU-Link is a powerful and cost effective debug probe that can be used seamlessly with MCUXpresso IDE, and is also compatible with 3rd party IDEs that support CMSIS-DAP protocol. MCU-Link also includes a USB to UART bridge feature (VCOM) that can be used to provide a serial connection between the target MCU and a host computer. MCU-Link features a high-speed USB interface for high performance debug. MCU-Link is compatible with Windows, MacOS and Linux. A free utility from NXP provides an easy way to install firmware updates.

On-board MCU-Link debugger supports CMSIS-DAP and J-Link firmware. See the table in [Default debug interfaces](#) to determine the default debug interface that comes loaded on your specific hardware platform.

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, 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.
- If using J-Link with either a standalone debug pod or MCU-Link, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.

Updating MCU-Link firmware 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 reprogram 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), MCU-Link 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 that 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 MCU-Link or NXP boards. The utility can be downloaded from [MCU-Link](#).

These steps show how to update the debugger firmware on your board for Windows operating system.

1. Install the MCU-Link utility.
2. Unplug the board's USB cable.
3. Make the DFU link (install the jumper labeled DFUlink).
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: <MCU-Link install dir>/scripts/program_CMSIS
 2. To program J-Link debug firmware: <MCU-Link install dir>/scripts/program_JLINK
6. Remove DFU link (remove the jumper installed in Step 3).
7. Repower the board by removing the USB cable and plugging it in again.

On-board debugger LPC-Link LPC-Link 2 is an extensible debug probe that can be used seamlessly with MCUXpresso IDE, and is also compatible with 3rd party IDEs that support CMSIS-DAP protocol. MCU-Link also includes a USB to UART bridge feature (VCOM) that can be used to provide a serial connection between the target MCU and a host computer. LPC-Link 2 is compatible with Windows, MacOS and Linux. A free utility from NXP provides an easy way to install firmware updates.

On-board LPC-Link 2 debugger supports CMSIS-DAP and J-Link firmware. See the table in [Default debug interfaces](#) to determine the default debug interface that comes loaded on your specific hardware platform.

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, 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.
- If using J-Link with either a standalone debug pod or MCU-Link, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.

Updating LPC-Link firmware The LPCXpresso 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 LPCScript. 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 reprogram 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 that most up-to-date/compatible firmware is used with MCUXpresso IDE.

NXP provides the LPCScript utility, which is the recommended tool for programming the latest versions of CMSIS-DAP and J-Link firmware onto LPC-Link2 or LPCXpresso boards. The utility can be downloaded from [LPCScript](#).

These steps show how to update the debugger firmware on your board for Windows operating system. For Linux OS, follow the instructions described in LPCScript user guide ([LPCScript](#), select **LPCScript**, and then the documentation tab).

1. Install the LPCScript utility.
2. Unplug the board's USB cable.
3. Make the DFU link (install the jumper labeled DFUlink).
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 LPCScript installation directory (<LPCScript 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. Repower the board by removing the USB cable and plugging it in again.

On-board debugger OpenSDA OpenSDA/OpenSDAv2 is a serial and debug adapter that is built into several NXP evaluation boards. It provides a bridge between your computer (or other USB host) and the embedded target processor, which can be used for debugging, flash programming, and serial communication, all over a simple USB cable.

The difference is the firmware implementation: OpenSDA: Programmed with the proprietary P&E Micro developed bootloader. P&E Micro is the default debug interface app. OpenSDAv2: Programmed with the open-sourced CMSIS-DAP/mbd bootloader. CMSIS-DAP is the default debug interface app.

See the table in [Default debug interfaces](#) to determine the default debug interface that comes loaded on your specific hardware platform.

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, 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.

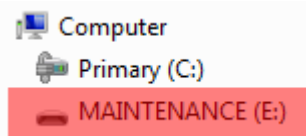
Updating OpenSDA firmware Any NXP hardware platform that comes with an OpenSDA-compatible debug interface has the ability to update the OpenSDA firmware. This typically means to switch from the default application (either CMSIS-DAP or P&E Micro) to a SEGGER J-Link. This section contains the steps to switch the OpenSDA firmware to a J-Link interface. However, the steps can be applied to restoring the original image also. 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 [Default debug interfaces](#). 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.
- CMSIS-DAP: CMSIS-DAP OpenSDA firmware is available at www.nxp.com/opensda.

- 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 reenter 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 reenter bootloader mode by holding down the **Reset** button and power cycling.

On-board debugger Multilink An on-board Multilink debug circuit provides a JTAG interface and a power supply input through a single micro-USB connector. It is a hardware interface that allows PC software to debug and program a target processor through its debug port.

The host driver must be installed before debugging.

- See [PE micro](#) to download and install the P&E Micro Hardware Interface Drivers package.

On-board debugger OSJTAG An on-board OSJTAG debug circuit provides a JTAG interface and a power supply input through a single micro-USB connector. It is a hardware interface that allows PC software to debug and program a target processor through its debug port.

The host driver must be installed before debugging.

- See [PE micro](#) to download and install the P&E Micro Hardware Interface Drivers package.

Default debug interfaces

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

Hardware platform	Default debugger firmware	On-board debugger probe
EVK-MCIMX7ULP	N/A	N/A
EVK-MIMX8MM	N/A	N/A
EVK-MIMX8MN	N/A	N/A
EVK-MIMX8MNDDR3L	N/A	N/A
EVK-MIMX8MP	N/A	N/A
EVK-MIMX8MQ	N/A	N/A
EVK-MIMX8ULP	N/A	N/A
EVK-MIMXRT1010	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1015	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1020	CMSIS-DAP	LPC-Link2
EVK-MIMXRT1064	CMSIS-DAP	LPC-Link2
EVK-MIMXRT595	CMSIS-DAP	LPC-Link2
EVK-MIMXRT685	CMSIS-DAP	LPC-Link2
EVK9-MIMX8ULP	N/A	N/A
EVKB-IMXRT1050	CMSIS-DAP	LPC-Link2
FRDM-K22F	CMSIS-DAP	OpenSDA v2
FRDM-K32L2A4S	CMSIS-DAP	OpenSDA v2
FRDM-K32L2B	CMSIS-DAP	OpenSDA v2
FRDM-K32L3A6	CMSIS-DAP	OpenSDA v2
FRDM-KE02Z40M	P&E Micro	OpenSDA v1
FRDM-KE15Z	CMSIS-DAP	OpenSDA v2
FRDM-KE16Z	CMSIS-DAP	OpenSDA v2
FRDM-KE17Z	CMSIS-DAP	OpenSDA v2
FRDM-KE17Z512	CMSIS-DAP	MCU-Link
FRDM-MCXA153	CMSIS-DAP	MCU-Link
FRDM-MCXA156	CMSIS-DAP	MCU-Link
FRDM-MCXA346	CMSIS-DAP	MCU-Link
FRDM-MCXC041	CMSIS-DAP	MCU-Link
FRDM-MCXC242	CMSIS-DAP	MCU-Link
FRDM-MCXC444	CMSIS-DAP	MCU-Link
FRDM-MCXE247	CMSIS-DAP	MCU-Link
FRDM-MCXN236	CMSIS-DAP	MCU-Link
FRDM-MCXN947	CMSIS-DAP	MCU-Link
FRDM-MCXW23	CMSIS-DAP	MCU-Link
FRDM-MCXW71	CMSIS-DAP	MCU-Link
FRDM-MCXW72	CMSIS-DAP	MCU-Link
FRDM-RW612	CMSIS-DAP	MCU-Link
IMX943-EVK	N/A	N/A
IMX95LP4XEVK-15	N/A	N/A
IMX95LPD5EVK-19	N/A	N/A
IMX95VERDINEVK	N/A	N/A
KW45B41Z-EVK	CMSIS-DAP	MCU-Link
KW45B41Z-LOC	CMSIS-DAP	MCU-Link
KW47-EVK	CMSIS-DAP	MCU-Link
KW47-LOC	CMSIS-DAP	MCU-Link
LPC845BREAKOUT	CMSIS-DAP	LPC-Link2
LPCXpresso51U68	CMSIS-DAP	LPC-Link2
LPCXpresso54628	CMSIS-DAP	LPC-Link2

continues on next page

Table 1 – continued from previous page

Hardware platform	Default debugger firmware	On-board debugger probe
LPCXpresso54S018	CMSIS-DAP	LPC-Link2
LPCXpresso54S018M	CMSIS-DAP	LPC-Link2
LPCXpresso55S06	CMSIS-DAP	LPC-Link2
LPCXpresso55S16	CMSIS-DAP	LPC-Link2
LPCXpresso55S28	CMSIS-DAP	LPC-Link2
LPCXpresso55S36	CMSIS-DAP	MCU-Link
LPCXpresso55S69	CMSIS-DAP	LPC-Link2
LPCXpresso802	CMSIS-DAP	LPC-Link2
LPCXpresso804	CMSIS-DAP	LPC-Link2
LPCXpresso824MAX	CMSIS-DAP	LPC-Link2
LPCXpresso845MAX	CMSIS-DAP	LPC-Link2
LPCXpresso860MAX	CMSIS-DAP	LPC-Link2
MC56F80000-EVK	P&E Micro	Multilink
MC56F81000-EVK	P&E Micro	Multilink
MC56F83000-EVK	P&E Micro	OSJTAG
MCIMX93-EVK	N/A	N/A
MCIMX93-QSB	N/A	N/A
MCIMX93AUTO-EVK	N/A	N/A
MCX-N5XX-EVK	CMSIS-DAP	MCU-Link
MCX-N9XX-EVK	CMSIS-DAP	MCU-Link
MCX-W71-EVK	CMSIS-DAP	MCU-Link
MCX-W72-EVK	CMSIS-DAP	MCU-Link
MIMXRT1024-EVK	CMSIS-DAP	LPC-Link2
MIMXRT1040-EVK	CMSIS-DAP	LPC-Link2
MIMXRT1060-EVKB	CMSIS-DAP	LPC-Link2
MIMXRT1060-EVKC	CMSIS-DAP	MCU-Link
MIMXRT1160-EVK	CMSIS-DAP	LPC-Link2
MIMXRT1170-EVKB	CMSIS-DAP	MCU-Link
MIMXRT1180-EVK	CMSIS-DAP	MCU-Link
MIMXRT685-AUD-EVK	CMSIS-DAP	LPC-Link2
MIMXRT700-EVK	CMSIS-DAP	MCU-Link
RD-RW612-BGA	CMSIS-DAP	MCU-Link
TWR-KM34Z50MV3	P&E Micro	OpenSDA v1
TWR-KM34Z75M	P&E Micro	OpenSDA v1
TWR-KM35Z75M	CMSIS-DAP	OpenSDA v2
TWR-MC56F8200	P&E Micro	OSJTAG
TWR-MC56F8400	P&E Micro	OSJTAG

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)
{
    // 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
}
```

1.3 Getting Started with MCUXpresso SDK GitHub

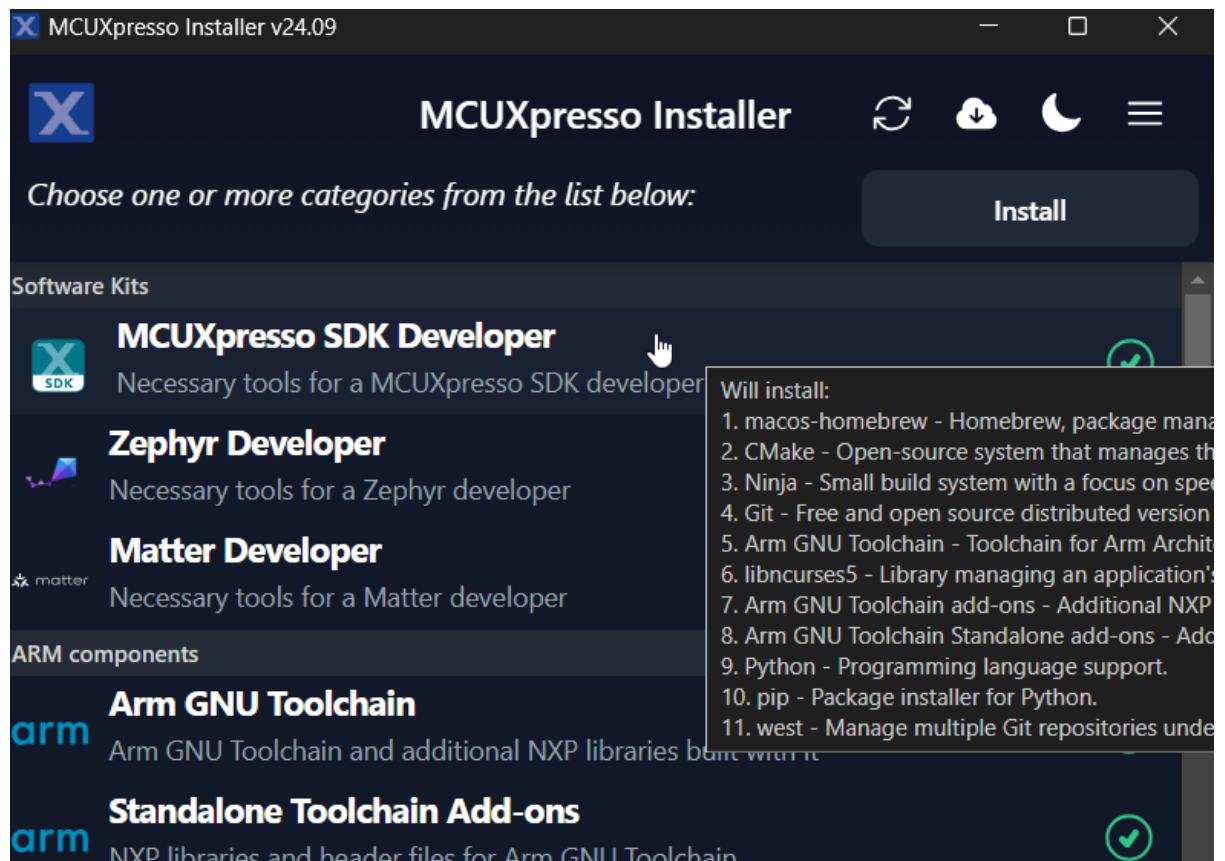
1.3.1 Getting Started with MCUXpresso SDK Repository

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:\Freescall\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:

```

└─ arm
└─ common
└─ install-info

```

- 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

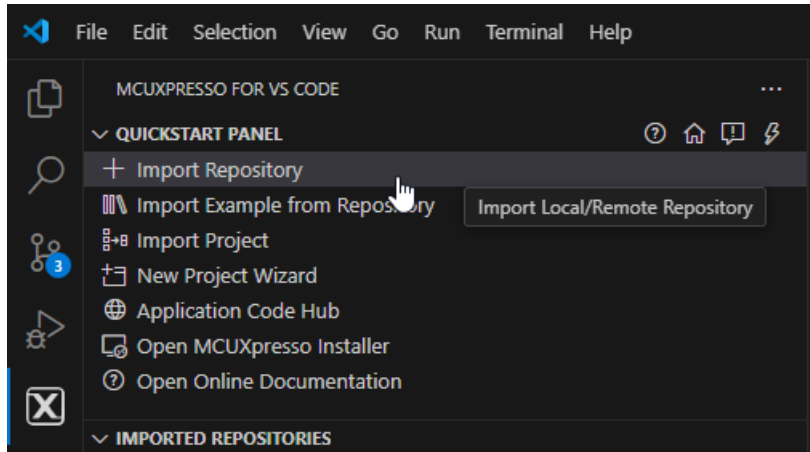
This section explains how to configure MCUXpresso for VS Code to build, run, and debug example applications. This guide uses the `hello_world` demo application as an example. However, these

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](#).

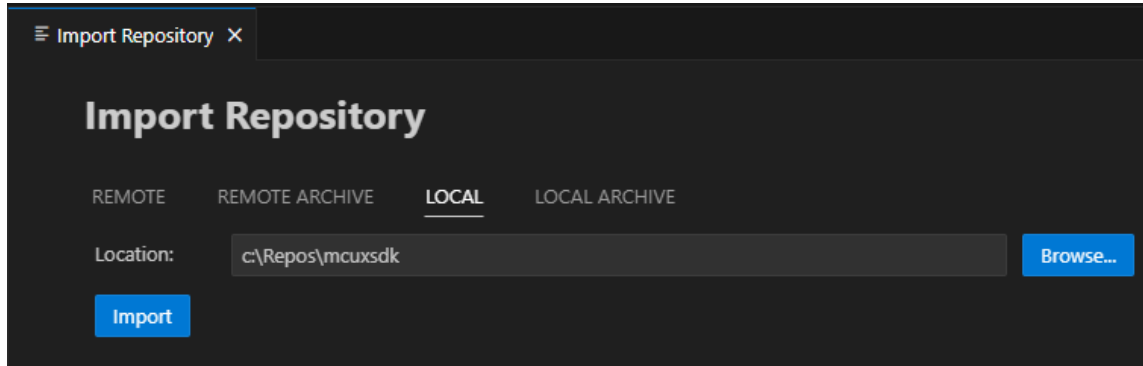
To build an example application:

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

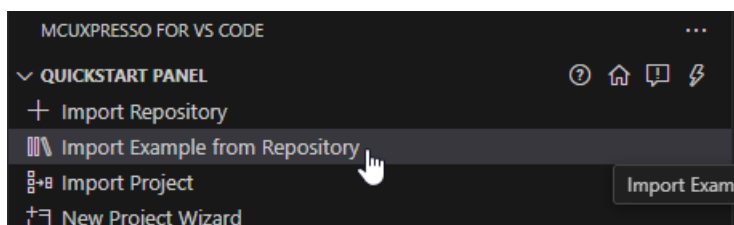


Note: You can import the SDK in several ways. Refer to [MCUXpresso for VS Code Wiki](#) for details.

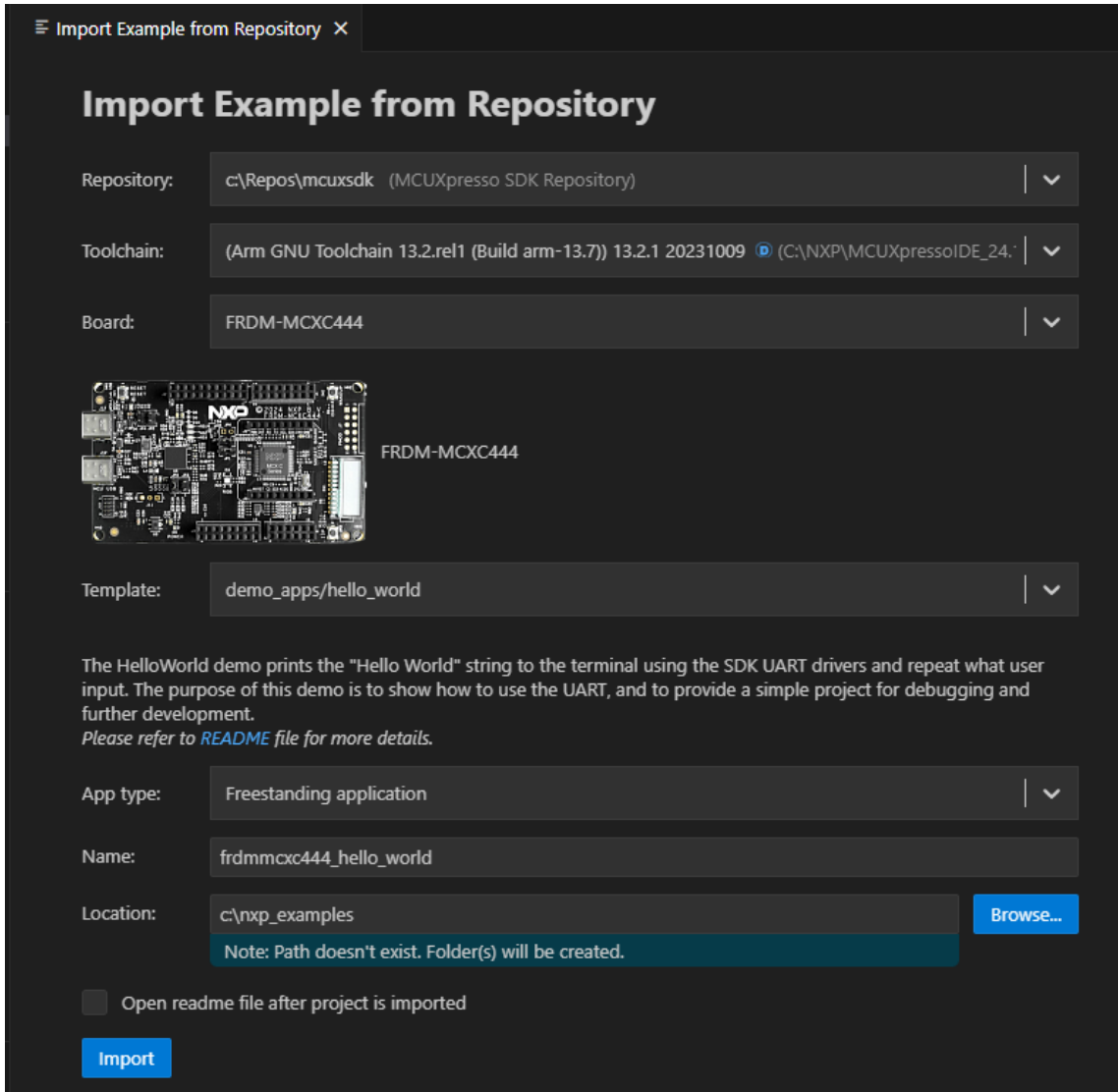
Select **Local** if you've already obtained the SDK as seen in [Get MCUXpresso SDK Repo](#). Select your location and click **Import**.



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) | ▾

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

Board: FRDM-MCXC444 | ▾

 FRDM-MCXC444

Template: demo_apps/hello_world | ▾

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 | ▾

Name: frdmmcxc444_hello_world

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

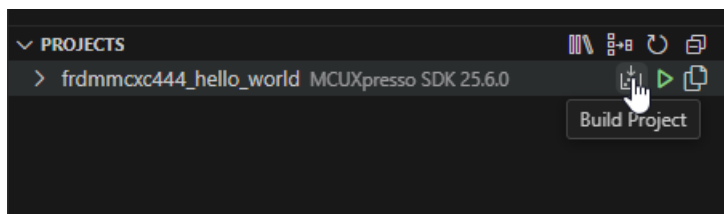
Note: Path doesn't exist. Folder(s) will be created.

☐ 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.

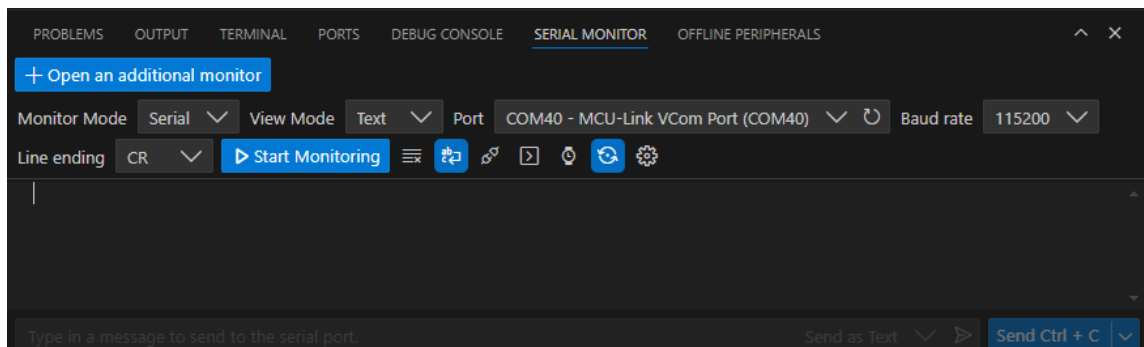
```

[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.

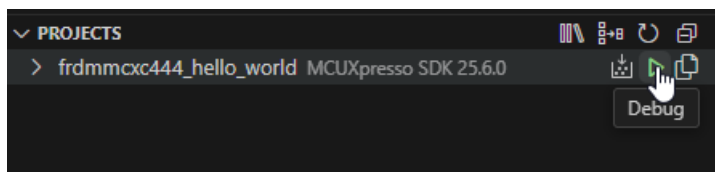
```

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
---- 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

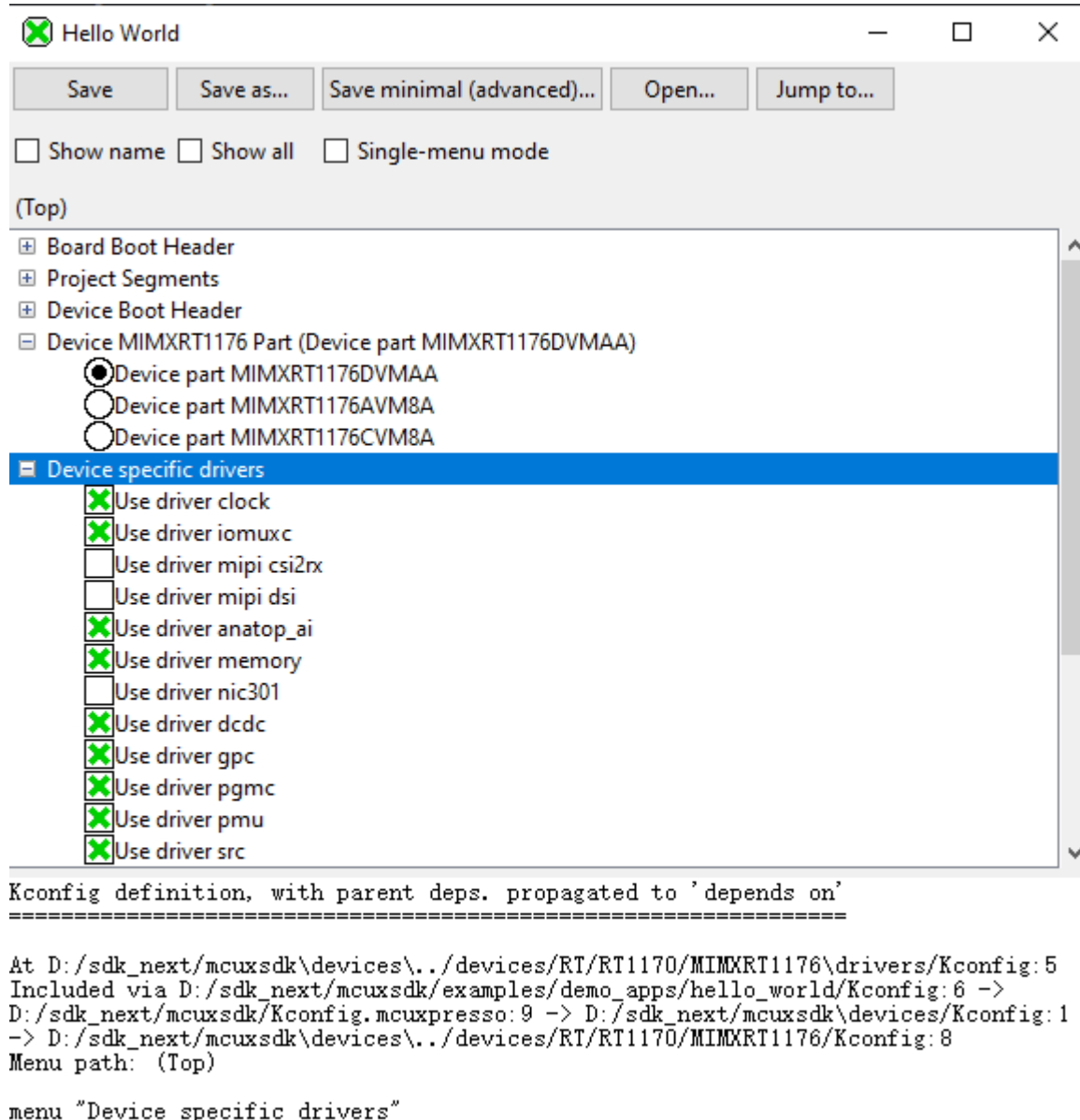
```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world -Dcore_id=cm7 --cmake-only -p
```

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:

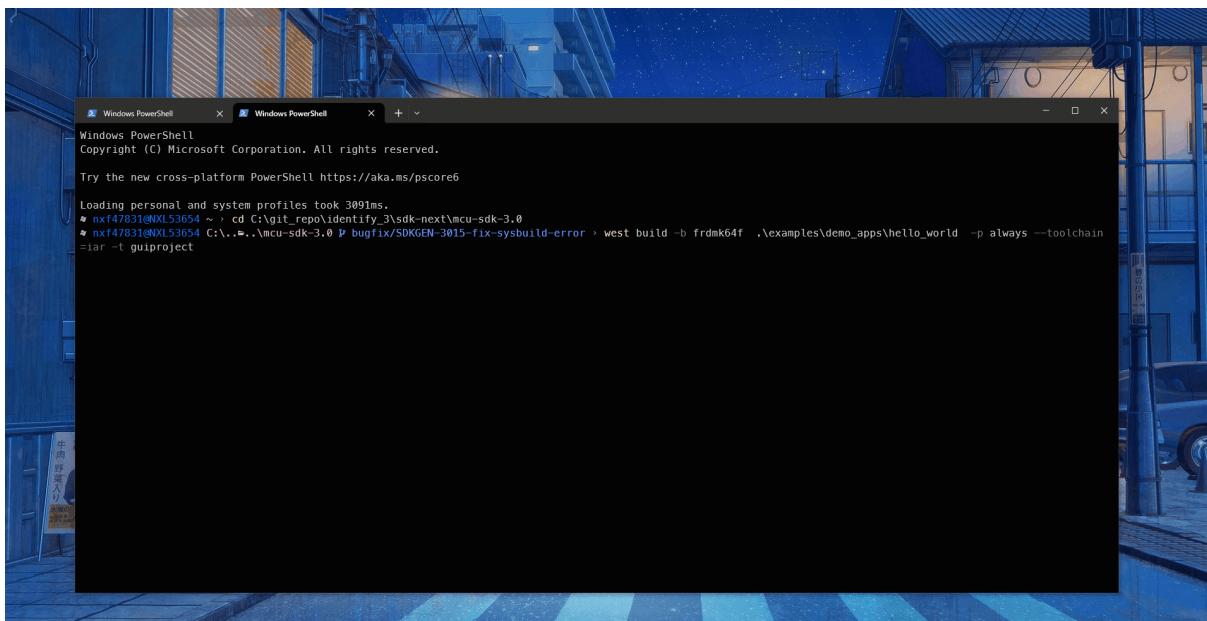
```
west debug -r linkserver
```

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
LPCXpresso55S69	LPC55S69JBD100 , LPC55S69JBD64, LPC55S69JEV59, LPC55S69JEV98, LPC55S66JBD100, LPC55S66JBD64, LPC55S66JEV98

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.

coreHTTP coreHTTP

PSA Test Suite Arm Platform Security Architecture Test Suite

mbedTLS mbedtls SSL/TLS library v3.x

USB Type-C PD Stack See the *MCUXpresso SDK USB Type-C PD Stack User's Guide* (document MCUXSDKUSBPUG) 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

SDMMC stack The SDMMC software is integrated with MCUXpresso SDK to support SD/MMC/SDIO standard specification. This also includes a host adapter layer for bare-metal/RTOS applications.

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”.

Motor Control Software (ACIM, BLDC, PMSM) Motor control examples.

Multicore Multicore Software Development Kit

MCU Boot Open source MCU Bootloader.

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).

Voice intelligent technology library Voice Intelligent Technology (VIT) Library provides wake word and voice command engine for voice control

Audio Voice components Audio Voice components for MCU

Maestro Audio Framework for MCU Maestro Audio Framework library for MCU

LVGL LVGL Open Source Graphics Library

llhttp HTTP parser llhttp

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.

emWin The MCUXpresso SDK is pre-integrated with the SEGGER emWin GUI middleware. The AppWizard provides developers and designers with a flexible tool to create stunning user interface applications, without writing any code.

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

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.

Cannot add SDK components into FreeRTOS projects

It is not possible to add any SDK components into FreeRTOS project using the MCUXpresso IDE New Project wizard.

The i2c_read_accel_value_transfer doesn't work

The example has problem with identification of accelerometer.

Affected toolchains: mcux, iar, mdk, armgcc **Affected platforms:** lpcxpresso55s69

Examples hello_world_ns, secure_faults_ns, and secure_faults_trdc_ns have incorrect library path in GUI projects

When the affected examples are generated as GUI projects, the library linking the secure and non-secure worlds has an incorrect path set. This causes linking errors during project compilation.

Examples: hello_world_ns, hello_world_s, secure_faults_ns, secure_faults_s, secure_faults_trdc_ns, secure_faults_trdc_s

Affected toolchains: mdk, iar

Workaround: In the IDE project settings for the non-secure (_ns) project, find the linked library (named `hello_world_s_CMSE_lib.o`, or similar, depending on the example project) and replace the path to the library with `<build_directory>/<secure_world_project_folder>/<IDE>/`, replacing the subdirectory names with the build directory, the secure world project name, and IDE name.

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
-

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.

[LPC55S69](#)

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 Multicore

[Multicore SDK](#)

1.7.2 MCU Boot

[mcuboot_opensource](#)

1.7.3 Audio Voice components

[Audio Voice Components](#)

1.7.4 Maestro Audio Framework for MCU

[Maestro Audio Framework](#)

1.7.5 FreeMASTER

[freemaster](#)

1.7.6 FreeRTOS

[FreeRTOS](#)

1.7.7 File systemFatfs

[FatFs](#)

Chapter 2

LPC55S69

2.1 ANACTRL: Analog Control Driver

`void ANACTRL_Init(ANACTRL_Type *base)`

Initializes the ANACTRL mode, the module's clock will be enabled by invoking this function.

Parameters

- `base` – ANACTRL peripheral base address.

`void ANACTRL_Deinit(ANACTRL_Type *base)`

De-initializes ANACTRL module, the module's clock will be disabled by invoking this function.

Parameters

- `base` – ANACTRL peripheral base address.

`void ANACTRL_SetFro192M(ANACTRL_Type *base, const anactrl_fro192M_config_t *config)`

Configures the on-chip high-speed Free Running Oscillator(FRO192M), such as enabling/disabling 12 MHz clock output and enable/disable 96MHz clock output.

Parameters

- `base` – ANACTRL peripheral base address.
- `config` – Pointer to FRO192M configuration structure. Refer to `anactrl_fro192M_config_t` structure.

`void ANACTRL_GetDefaultFro192MConfig(anactrl_fro192M_config_t *config)`

Gets the default configuration of FRO192M. The default values are:

```
config->enable12MHzClk = true;
config->enable96MHzClk = false;
```

Parameters

- `config` – Pointer to FRO192M configuration structure. Refer to `anactrl_fro192M_config_t` structure.

`void ANACTRL_SetXo32M(ANACTRL_Type *base, const anactrl_xo32M_config_t *config)`

Configures the 32 MHz Crystal oscillator(High-speed crystal oscillator), such as enable/disable output to CPU system, and so on.

Parameters

- `base` – ANACTRL peripheral base address.
- `config` – Pointer to XO32M configuration structure. Refer to `anactrl_xo32M_config_t` structure.

`void ANACTRL_GetDefaultXo32MConfig(anactrl_xo32M_config_t *config)`

Gets the default configuration of XO32M. The default values are:

```
config->enableSysClkOutput = false;  
config->enableACBufferBypass = false;
```

Parameters

- `config` – Pointer to XO32M configuration structure. Refer to `anactrl_xo32M_config_t` structure.

`uint32_t ANACTRL_MeasureFrequency(ANACTRL_Type *base, uint8_t scale, uint32_t refClkFreq)`

Measures the frequency of the target clock source.

This function measures target frequency according to a accurate reference frequency. The formula is: $F_{target} = (CAPVAL * F_{reference}) / ((1 \ll SCALE) - 1)$

Note: Both target and reference clocks are selectable by programming the target clock select `FREQMEAS_TARGET` register in `INPUTMUX` and reference clock select `FREQMEAS_REF` register in `INPUTMUX`.

Parameters

- `base` – ANACTRL peripheral base address.
- `scale` – Define the power of 2 count that ref counter counts to during measurement, ranges from 2 to 31.
- `refClkFreq` – frequency of the reference clock.

Returns

frequency of the target clock.

`static inline void ANACTRL_EnableInterrupts(ANACTRL_Type *base, uint32_t mask)`

Enables the ANACTRL interrupts.

Parameters

- `base` – ANACTRL peripheral base address.
- `mask` – The interrupt mask. Refer to “`_anactrl_interrupt`” enumeration.

`static inline void ANACTRL_DisableInterrupts(ANACTRL_Type *base, uint32_t mask)`

Disables the ANACTRL interrupts.

Parameters

- `base` – ANACTRL peripheral base address.
- `mask` – The interrupt mask. Refer to “`_anactrl_interrupt`” enumeration.

`static inline void ANACTRL_ClearInterrupts(ANACTRL_Type *base, uint32_t mask)`

Clears the ANACTRL interrupts.

Parameters

- `base` – ANACTRL peripheral base address.
- `mask` – The interrupt mask. Refer to “`_anactrl_interrupt`” enumeration.

static inline uint32_t ANACTRL_GetStatusFlags(ANACTRL_Type *base)

Gets ANACTRL status flags.

This function gets Analog control status flags. The flags are returned as the logical OR value of the enumerators `_anactrl_flags`. To check for a specific status, compare the return value with enumerators in the `_anactrl_flags`. For example, to check whether the flash is in power down mode:

```
if (kANACTRL_FlashPowerDownFlag & ANACTRL_ANACTRL_GetStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- `base` – ANACTRL peripheral base address.

Returns

ANACTRL status flags which are given in the enumerators in the `_anactrl_flags`.

static inline uint32_t ANACTRL_GetOscStatusFlags(ANACTRL_Type *base)

Gets ANACTRL oscillators status flags.

This function gets Anactrl oscillators status flags. The flags are returned as the logical OR value of the enumerators `_anactrl_osc_flags`. To check for a specific status, compare the return value with enumerators in the `_anactrl_osc_flags`. For example, to check whether the FRO192M clock output is valid:

```
if (kANACTRL_OutputClkValidFlag & ANACTRL_ANACTRL_GetOscStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- `base` – ANACTRL peripheral base address.

Returns

ANACTRL oscillators status flags which are given in the enumerators in the `_anactrl_osc_flags`.

static inline uint32_t ANACTRL_GetInterruptStatusFlags(ANACTRL_Type *base)

Gets ANACTRL interrupt status flags.

This function gets Anactrl interrupt status flags. The flags are returned as the logical OR value of the enumerators `_anactrl_interrupt_flags`. To check for a specific status, compare the return value with enumerators in the `_anactrl_interrupt_flags`. For example, to check whether the VBAT voltage level is above the threshold:

```
if (kANACTRL_BodVbatPowerFlag & ANACTRL_ANACTRL_GetInterruptStatusFlags(ANACTRL))
{
    ...
}
```

Parameters

- `base` – ANACTRL peripheral base address.

Returns

ANACTRL oscillators status flags which are given in the enumerators in the `_anactrl_osc_flags`.

static inline void ANACTRL_EnableVref1V(ANACTRL_Type *base, bool enable)

Aux_Bias Control Interfaces.

Enables/disables 1V reference voltage buffer.

Parameters

- base – ANACTRL peripheral base address.
- enable – Used to enable or disable 1V reference voltage buffer.

enum __anactrl_interrupt_flags

ANACTRL interrupt flags.

Values:

enumerator kANACTRL_BodVbatFlag

BOD VBAT Interrupt status before Interrupt Enable.

enumerator kANACTRL_BodVbatInterruptFlag

BOD VBAT Interrupt status after Interrupt Enable.

enumerator kANACTRL_BodVbatPowerFlag

Current value of BOD VBAT power status output.

enumerator kANACTRL_BodCoreFlag

BOD CORE Interrupt status before Interrupt Enable.

enumerator kANACTRL_BodCoreInterruptFlag

BOD CORE Interrupt status after Interrupt Enable.

enumerator kANACTRL_BodCorePowerFlag

Current value of BOD CORE power status output.

enumerator kANACTRL_DcdcFlag

DCDC Interrupt status before Interrupt Enable.

enumerator kANACTRL_DcdcInterruptFlag

DCDC Interrupt status after Interrupt Enable.

enumerator kANACTRL_DcdcPowerFlag

Current value of DCDC power status output.

enum __anactrl_interrupt

ANACTRL interrupt control.

Values:

enumerator kANACTRL_BodVbatInterruptEnable

BOD VBAT interrupt control.

enumerator kANACTRL_BodCoreInterruptEnable

BOD CORE interrupt control.

enumerator kANACTRL_DcdcInterruptEnable

DCDC interrupt control.

enum __anactrl_flags

ANACTRL status flags.

Values:

enumerator kANACTRL_FlashPowerDownFlag

Flash power-down status.

enumerator kANACTRL_FlashInitErrorFlag

Flash initialization error status.

enum _anactrl_osc_flags

ANACTRL FRO192M and XO32M status flags.

Values:

enumerator kANACTRL_OutputClkValidFlag

Output clock valid signal.

enumerator kANACTRL_CCOTresholdVoltageFlag

CCO threshold voltage detector output (signal vcco_ok).

enumerator kANACTRL_XO32MOutputReadyFlag

Indicates XO out frequency statibility.

typedef struct *_anactrl_fro192M_config* anactrl_fro192M_config_t

Configuration for FRO192M.

This structure holds the configuration settings for the on-chip high-speed Free Running Oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultFro192MConfig() function and pass a pointer to your config structure instance.

typedef struct *_anactrl_xo32M_config* anactrl_xo32M_config_t

Configuration for XO32M.

This structure holds the configuration settings for the 32 MHz crystal oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultXo32MConfig() function and pass a pointer to your config structure instance.

FSL_ANACTRL_DRIVER_VERSION

ANACTRL driver version.

struct _anactrl_fro192M_config

#include <fsl_anactrl.h> Configuration for FRO192M.

This structure holds the configuration settings for the on-chip high-speed Free Running Oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultFro192MConfig() function and pass a pointer to your config structure instance.

Public Members

bool enable12MHzClk

Enable 12MHz clock.

bool enable96MHzClk

Enable 96MHz clock.

struct _anactrl_xo32M_config

#include <fsl_anactrl.h> Configuration for XO32M.

This structure holds the configuration settings for the 32 MHz crystal oscillator. To initialize this structure to reasonable defaults, call the ANACTRL_GetDefaultXo32MConfig() function and pass a pointer to your config structure instance.

Public Members

`bool enableACBufferBypass`

Enable XO AC buffer bypass in pll and top level.

`bool enableSysCLkOutput`

Enable XO 32 MHz output to CPU system, SCT, and CLKOUT

`bool enableADCOutput`

Enable High speed crystal oscillator output to ADC.

2.2 CASPER: The Cryptographic Accelerator and Signal Processing Engine with RAM sharing

2.3 casper_driver

`FSL_CASPER_DRIVER_VERSION`

CASPER driver version. Version 2.2.4.

Current version: 2.2.4

Change log:

- Version 2.0.0
 - Initial version
- Version 2.0.1
 - Bug fix KPSDK-24531 `double_scalar_multiplication()` result may be all zeroes for some specific input
- Version 2.0.2
 - Bug fix KPSDK-25015 `CASPER_MEMCPY` hard-fault on LPC55xx when both source and destination buffers are outside of `CASPER_RAM`
- Version 2.0.3
 - Bug fix KPSDK-28107 `RSUB`, `FILL` and `ZERO` operations not implemented in `enum_casper_operation`.
- Version 2.0.4
 - For GCC compiler, enforce `O1` optimize level, specifically to remove strict-aliasing option. This driver is very specific and requires `-fno-strict-aliasing`.
- Version 2.0.5
 - Fix sign-compare warning.
- Version 2.0.6
 - Fix IAR Pa082 warning.
- Version 2.0.7
 - Fix MISRA-C 2012 issue.
- Version 2.0.8
 - Add feature macro for `CASPER_RAM_OFFSET`.
- Version 2.0.9
 - Remove unused function `Jac_oncurve()`.
 - Fix ECC384 build.

- Version 2.0.10
 - Fix MISRA-C 2012 issue.
- Version 2.1.0
 - Add ECC NIST P-521 elliptic curve.
- Version 2.2.0
 - Rework driver to support multiple curves at once.
- Version 2.2.1
 - Fix MISRA-C 2012 issue.
- Version 2.2.2
 - Enable hardware interleaving to RAMX0 and RAMX1 for CASPER by feature macro FSL_FEATURE_CASPER_RAM_HW_INTERLEAVE
- Version 2.2.3
 - Added macro into CASPER_Init and CASPER_Deinit to support devices without clock and reset control.
- Version 2.2.4
 - Fix MISRA-C 2012 issue.

enum _casper_operation
CASPER operation.

Values:

enumerator kCASPER_OpMul6464NoSum

enumerator kCASPER_OpMul6464Sum

Walking 1 or more of J loop, doing $r=a*b$ using $64x64=128$

enumerator kCASPER_OpMul6464FullSum

Walking 1 or more of J loop, doing $c,r=r+a*b$ using $64x64=128$, but assume inner j loop

enumerator kCASPER_OpMul6464Reduce

Walking 1 or more of J loop, doing $c,r=r+a*b$ using $64x64=128$, but sum all of w.

enumerator kCASPER_OpAdd64

Walking 1 or more of J loop, doing $c,r[-1]=r+a*b$ using $64x64=128$, but skip 1st write

enumerator kCASPER_OpSub64

Walking add with off_AB, and in/out off_RES doing $c,r=r+a+c$ using $64+64=65$

enumerator kCASPER_OpDouble64

Walking subtract with off_AB, and in/out off_RES doing $r=r-a$ using $64-64=64$, with last borrow implicit if any

enumerator kCASPER_OpXor64

Walking add to self with off_RES doing $c,r=r+r+c$ using $64+64=65$

enumerator kCASPER_OpRSub64

Walking XOR with off_AB, and in/out off_RES doing $r=r^a$ using $64^64=64$

enumerator kCASPER_OpShiftLeft32

Walking subtract with off_AB, and in/out off_RES using $r=a-r$

enumerator kCASPER_OpShiftRight32

Walking shift left doing $r1,r=(b*D)|r1$, where D is 2^{amt} and is loaded by app (off_CD not used)

enumerator kCASPER_OpCopy

Walking shift right doing $r, r1 = (b * D) | r1$, where D is $2^{(32 - \text{amt})}$ and is loaded by app (off_CD not used) and off_RES starts at MSW

enumerator kCASPER_OpRemask

Copy from ABoff to resoff, 64b at a time

enumerator kCASPER_OpFill

Copy and mask from ABoff to resoff, 64b at a time

enumerator kCASPER_OpZero

Fill RESOFF using 64 bits at a time with value in A and B

enumerator kCASPER_OpCompare

Fill RESOFF using 64 bits at a time of 0s

enumerator kCASPER_OpCompareFast

Compare two arrays, running all the way to the end

enum _casper_algo_t

Algorithm used for CASPER operation.

Values:

enumerator kCASPER_ECC_P256

ECC_P256

enumerator kCASPER_ECC_P384

ECC_P384

enumerator kCASPER_ECC_P521

ECC_P521

Values:

enumerator kCASPER_RamOffset_Result

enumerator kCASPER_RamOffset_Base

enumerator kCASPER_RamOffset_TempBase

enumerator kCASPER_RamOffset_Modulus

enumerator kCASPER_RamOffset_M64

typedef enum _casper_operation casper_operation_t

CASPER operation.

typedef enum _casper_algo_t casper_algo_t

Algorithm used for CASPER operation.

void CASPER_Init(CASPER_Type *base)

Enables clock and disables reset for CASPER peripheral.

Enable clock and disable reset for CASPER.

Parameters

- base – CASPER base address

void CASPER_Deinit(CASPER_Type *base)

Disables clock for CASPER peripheral.

Disable clock and enable reset.

Parameters

- base – CASPER base address

CASPER_CP

CASPER_CP_CTRL0

CASPER_CP_CTRL1

CASPER_CP_LOADER

CASPER_CP_STATUS

CASPER_CP_INTENSET

CASPER_CP_INTENCLR

CASPER_CP_INTSTAT

CASPER_CP_AREG

CASPER_CP_BREG

CASPER_CP_CREG

CASPER_CP_DREG

CASPER_CP_RES0

CASPER_CP_RES1

CASPER_CP_RES2

CASPER_CP_RES3

CASPER_CP_MASK

CASPER_CP_REMASK

CASPER_CP_LOCK

CASPER_CP_ID

CASPER_Wr32b(value, off)

CASPER_Wr64b(value, off)

CASPER_Rd32b(off)

N_wordlen_max

2.4 casper_driver_pkha

void CASPER_ModExp(CASPER_Type *base, const uint8_t *signature, const uint8_t *pubN, size_t wordLen, uint32_t pubE, uint8_t *plaintext)

Performs modular exponentiation - $(A^E) \bmod N$.

This function performs modular exponentiation.

Parameters

- base – CASPER base address

- signature – first addend (in little endian format)
- pubN – modulus (in little endian format)
- wordLen – Size of pubN in bytes
- pubE – exponent
- plaintext – **[out]** Output array to store result of operation (in little endian format)

void CASPER_ecc_init(*casper_algo_t* curve)

Initialize prime modulus mod in Casper memory .

Set the prime modulus mod in Casper memory and set N_wordlen according to selected algorithm.

Parameters

- curve – elliptic curve algorithm

void CASPER_ECC_SECP256R1_Mul(CASPER_Type *base, uint32_t resX[8], uint32_t resY[8],
uint32_t X[8], uint32_t Y[8], uint32_t scalar[8])

Performs ECC secp256r1 point single scalar multiplication.

This function performs ECC secp256r1 point single scalar multiplication [resX; resY] = scalar * [X; Y] Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate in normal form, little endian.
- resY – **[out]** Output Y affine coordinate in normal form, little endian.
- X – Input X affine coordinate in normal form, little endian.
- Y – Input Y affine coordinate in normal form, little endian.
- scalar – Input scalar integer, in normal form, little endian.

void CASPER_ECC_SECP256R1_MulAdd(CASPER_Type *base, uint32_t resX[8], uint32_t
resY[8], uint32_t X1[8], uint32_t Y1[8], uint32_t
scalar1[8], uint32_t X2[8], uint32_t Y2[8], uint32_t
scalar2[8])

Performs ECC secp256r1 point double scalar multiplication.

This function performs ECC secp256r1 point double scalar multiplication [resX; resY] = scalar1 * [X1; Y1] + scalar2 * [X2; Y2] Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate.
- resY – **[out]** Output Y affine coordinate.
- X1 – Input X1 affine coordinate.
- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.

- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

```
void CASPER_ECC_SECP384R1_Mul(CASPER_Type *base, uint32_t resX[12], uint32_t resY[12],
                              uint32_t X[12], uint32_t Y[12], uint32_t scalar[12])
```

Performs ECC secp384r1 point single scalar multiplication.

This function performs ECC secp384r1 point single scalar multiplication $[resX; resY] = scalar * [X; Y]$ Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate in normal form, little endian.
- resY – **[out]** Output Y affine coordinate in normal form, little endian.
- X – Input X affine coordinate in normal form, little endian.
- Y – Input Y affine coordinate in normal form, little endian.
- scalar – Input scalar integer, in normal form, little endian.

```
void CASPER_ECC_SECP384R1_MulAdd(CASPER_Type *base, uint32_t resX[12], uint32_t
                                resY[12], uint32_t X1[12], uint32_t Y1[12], uint32_t
                                scalar1[12], uint32_t X2[12], uint32_t Y2[12], uint32_t
                                scalar2[12])
```

Performs ECC secp384r1 point double scalar multiplication.

This function performs ECC secp384r1 point double scalar multiplication $[resX; resY] = scalar1 * [X1; Y1] + scalar2 * [X2; Y2]$ Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate.
- resY – **[out]** Output Y affine coordinate.
- X1 – Input X1 affine coordinate.
- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.
- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

```
void CASPER_ECC_SECP521R1_Mul(CASPER_Type *base, uint32_t resX[18], uint32_t resY[18],
                              uint32_t X[18], uint32_t Y[18], uint32_t scalar[18])
```

Performs ECC secp521r1 point single scalar multiplication.

This function performs ECC secp521r1 point single scalar multiplication $[resX; resY] = scalar * [X; Y]$ Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address

- resX – **[out]** Output X affine coordinate in normal form, little endian.
- resY – **[out]** Output Y affine coordinate in normal form, little endian.
- X – Input X affine coordinate in normal form, little endian.
- Y – Input Y affine coordinate in normal form, little endian.
- scalar – Input scalar integer, in normal form, little endian.

```
void CASPER_ECC_SECP521R1_MulAdd(CASPER_Type *base, uint32_t resX[18], uint32_t  
                                resY[18], uint32_t X1[18], uint32_t Y1[18], uint32_t  
                                scalar1[18], uint32_t X2[18], uint32_t Y2[18], uint32_t  
                                scalar2[18])
```

Performs ECC secp521r1 point double scalar multiplication.

This function performs ECC secp521r1 point double scalar multiplication [resX; resY] = scalar1 * [X1; Y1] + scalar2 * [X2; Y2] Coordinates are affine in normal form, little endian. Scalars are little endian. All arrays are little endian byte arrays, uint32_t type is used only to enforce the 32-bit alignment (0-mod-4 address).

Parameters

- base – CASPER base address
- resX – **[out]** Output X affine coordinate.
- resY – **[out]** Output Y affine coordinate.
- X1 – Input X1 affine coordinate.
- Y1 – Input Y1 affine coordinate.
- scalar1 – Input scalar1 integer.
- X2 – Input X2 affine coordinate.
- Y2 – Input Y2 affine coordinate.
- scalar2 – Input scalar2 integer.

```
void CASPER_ECC_equal(int *res, uint32_t *op1, uint32_t *op2)
```

```
void CASPER_ECC_equal_to_zero(int *res, uint32_t *op1)
```

2.5 Clock Driver

```
enum _clock_ip_name
```

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

Values:

```
enumerator kCLOCK_IpInvalid
```

Invalid Ip Name.

```
enumerator kCLOCK_Rom
```

Clock gate name: Rom.

```
enumerator kCLOCK_Sram1
```

Clock gate name: Sram1.

```
enumerator kCLOCK_Sram2
```

Clock gate name: Sram2.

enumerator kCLOCK_Sram3
Clock gate name: Sram3.

enumerator kCLOCK_Sram4
Clock gate name: Sram4.

enumerator kCLOCK_Flash
Clock gate name: Flash.

enumerator kCLOCK_Fmc
Clock gate name: Fmc.

enumerator kCLOCK_InputMux
Clock gate name: InputMux.

enumerator kCLOCK_Iocon
Clock gate name: Iocon.

enumerator kCLOCK_Gpio0
Clock gate name: Gpio0.

enumerator kCLOCK_Gpio1
Clock gate name: Gpio1.

enumerator kCLOCK_Gpio2
Clock gate name: Gpio2.

enumerator kCLOCK_Gpio3
Clock gate name: Gpio3.

enumerator kCLOCK_Pint
Clock gate name: Pint.

enumerator kCLOCK_Gint
Clock gate name: Gint.

enumerator kCLOCK_Dma0
Clock gate name: Dma0.

enumerator kCLOCK_Crc
Clock gate name: Crc.

enumerator kCLOCK_Wwdt
Clock gate name: Wwdt.

enumerator kCLOCK_Rtc
Clock gate name: Rtc.

enumerator kCLOCK_Mailbox
Clock gate name: Mailbox.

enumerator kCLOCK_Adc0
Clock gate name: Adc0.

enumerator kCLOCK_Mrt
Clock gate name: Mrt.

enumerator kCLOCK_OsTimer0
Clock gate name: OsTimer0.

enumerator kCLOCK_Sct0
Clock gate name: Sct0.

enumerator kCLOCK_Utick0
Clock gate name: Utick0.

enumerator kCLOCK_FlexComm0
Clock gate name: FlexComm0.

enumerator kCLOCK_FlexComm1
Clock gate name: FlexComm1.

enumerator kCLOCK_FlexComm2
Clock gate name: FlexComm2.

enumerator kCLOCK_FlexComm3
Clock gate name: FlexComm3.

enumerator kCLOCK_FlexComm4
Clock gate name: FlexComm4.

enumerator kCLOCK_FlexComm5
Clock gate name: FlexComm5.

enumerator kCLOCK_FlexComm6
Clock gate name: FlexComm6.

enumerator kCLOCK_FlexComm7
Clock gate name: FlexComm7.

enumerator kCLOCK_MinUart0
Clock gate name: MinUart0.

enumerator kCLOCK_MinUart1
Clock gate name: MinUart1.

enumerator kCLOCK_MinUart2
Clock gate name: MinUart2.

enumerator kCLOCK_MinUart3
Clock gate name: MinUart3.

enumerator kCLOCK_MinUart4
Clock gate name: MinUart4.

enumerator kCLOCK_MinUart5
Clock gate name: MinUart5.

enumerator kCLOCK_MinUart6
Clock gate name: MinUart6.

enumerator kCLOCK_MinUart7
Clock gate name: MinUart7.

enumerator kCLOCK_LSpi0
Clock gate name: LSpi0.

enumerator kCLOCK_LSpi1
Clock gate name: LSpi1.

enumerator kCLOCK_LSpi2
Clock gate name: LSpi2.

enumerator kCLOCK_LSpi3
Clock gate name: LSpi3.

enumerator kCLOCK_LSpi4
Clock gate name: LSpi4.

enumerator kCLOCK_LSpi5
Clock gate name: LSpi5.

enumerator kCLOCK_LSpi6
Clock gate name: LSpi6.

enumerator kCLOCK_LSpi7
Clock gate name: LSpi7.

enumerator kCLOCK_BI2c0
Clock gate name: BI2c0.

enumerator kCLOCK_BI2c1
Clock gate name: BI2c1.

enumerator kCLOCK_BI2c2
Clock gate name: BI2c2.

enumerator kCLOCK_BI2c3
Clock gate name: BI2c3.

enumerator kCLOCK_BI2c4
Clock gate name: BI2c4.

enumerator kCLOCK_BI2c5
Clock gate name: BI2c5.

enumerator kCLOCK_BI2c6
Clock gate name: BI2c6.

enumerator kCLOCK_BI2c7
Clock gate name: BI2c7.

enumerator kCLOCK_FlexI2s0
Clock gate name: FlexI2s0.

enumerator kCLOCK_FlexI2s1
Clock gate name: FlexI2s1.

enumerator kCLOCK_FlexI2s2
Clock gate name: FlexI2s2.

enumerator kCLOCK_FlexI2s3
Clock gate name: FlexI2s3.

enumerator kCLOCK_FlexI2s4
Clock gate name: FlexI2s4.

enumerator kCLOCK_FlexI2s5
Clock gate name: FlexI2s5.

enumerator kCLOCK_FlexI2s6
Clock gate name: FlexI2s6.

enumerator kCLOCK_FlexI2s7
Clock gate name: FlexI2s7.

enumerator kCLOCK_Timer2
Clock gate name: Timer2.

enumerator kCLOCK__Usbd0
Clock gate name: Usbd0.

enumerator kCLOCK__Timer0
Clock gate name: Timer0.

enumerator kCLOCK__Timer1
Clock gate name: Timer1.

enumerator kCLOCK__Pvt
Clock gate name: Pvt.

enumerator kCLOCK__Ezha
Clock gate name: Ezha.

enumerator kCLOCK__Ezha
Clock gate name: Ezha.

enumerator kCLOCK__Ezha
Clock gate name: Ezha.

enumerator kCLOCK__Dma1
Clock gate name: Dma1.

enumerator kCLOCK__Comp
Clock gate name: Comp.

enumerator kCLOCK__Sdio
Clock gate name: Sdio.

enumerator kCLOCK__Usbh1
Clock gate name: Usbh1.

enumerator kCLOCK__Usbd1
Clock gate name: Usbd1.

enumerator kCLOCK__UsbRam1
Clock gate name: UsbRam1.

enumerator kCLOCK__Usb1Clk
Clock gate name: Usb1Clk.

enumerator kCLOCK__Freqme
Clock gate name: Freqme.

enumerator kCLOCK__Rng
Clock gate name: Rng.

enumerator kCLOCK__InputMux1
Clock gate name: InputMux1.

enumerator kCLOCK__Sysctl
Clock gate name: Sysctl.

enumerator kCLOCK__Usbhmr0
Clock gate name: Usbhmr0.

enumerator kCLOCK__Usbhsl0
Clock gate name: Usbhsl0.

enumerator kCLOCK__HashCrypt
Clock gate name: HashCrypt.

enumerator kCLOCK__PowerQuad
Clock gate name: PowerQuad.

enumerator kCLOCK_PluLut

Clock gate name: PluLut.

enumerator kCLOCK_Timer3

Clock gate name: Timer3.

enumerator kCLOCK_Timer4

Clock gate name: Timer4.

enumerator kCLOCK_Puf

Clock gate name: Puf.

enumerator kCLOCK_Casper

Clock gate name: Casper.

enumerator kCLOCK_AnalogCtrl

Clock gate name: AnalogCtrl.

enumerator kCLOCK_Hs_Lspi

Clock gate name: Lspi.

enumerator kCLOCK_Gpio_Sec

Clock gate name: GPIO Sec.

enumerator kCLOCK_Gpio_Sec_Int

Clock gate name: GPIO SEC Int.

enum _clock_name

Clock name used to get clock frequency.

Values:

enumerator kCLOCK_CoreSysClk

Core/system clock (aka MAIN_CLK)

enumerator kCLOCK_BusClk

Bus clock (AHB clock)

enumerator kCLOCK_ClockOut

CLOCKOUT

enumerator kCLOCK_FroHf

FRO48/96

enumerator kCLOCK_Pll1Out

PLL1 Output

enumerator kCLOCK_Mclk

MCLK

enumerator kCLOCK_Fro12M

FRO12M

enumerator kCLOCK_ExtClk

External Clock

enumerator kCLOCK_Pll0Out

PLL0 Output

enumerator kCLOCK_FlexI2S

FlexI2S clock

enum _clock_attach_id

The enumerator of clock attach Id.

Values:

enumerator kFRO12M_to_MAIN_CLK

Attach FRO12M to MAIN_CLK.

enumerator kEXT_CLK_to_MAIN_CLK

Attach EXT_CLK to MAIN_CLK.

enumerator kFRO1M_to_MAIN_CLK

Attach FRO1M to MAIN_CLK.

enumerator kFRO_HF_to_MAIN_CLK

Attach FRO_HF to MAIN_CLK.

enumerator kPLL0_to_MAIN_CLK

Attach PLL0 to MAIN_CLK.

enumerator kPLL1_to_MAIN_CLK

Attach PLL1 to MAIN_CLK.

enumerator kOSC32K_to_MAIN_CLK

Attach OSC32K to MAIN_CLK.

enumerator kMAIN_CLK_to_CLKOUT

Attach MAIN_CLK to CLKOUT.

enumerator kPLL0_to_CLKOUT

Attach PLL0 to CLKOUT.

enumerator kEXT_CLK_to_CLKOUT

Attach EXT_CLK to CLKOUT.

enumerator kFRO_HF_to_CLKOUT

Attach FRO_HF to CLKOUT.

enumerator kFRO1M_to_CLKOUT

Attach FRO1M to CLKOUT.

enumerator kPLL1_to_CLKOUT

Attach PLL1 to CLKOUT.

enumerator kOSC32K_to_CLKOUT

Attach OSC32K to CLKOUT.

enumerator kNONE_to_SYS_CLKOUT

Attach NONE to SYS_CLKOUT.

enumerator kFRO12M_to_PLL0

Attach FRO12M to PLL0.

enumerator kEXT_CLK_to_PLL0

Attach EXT_CLK to PLL0.

enumerator kFRO1M_to_PLL0

Attach FRO1M to PLL0.

enumerator kOSC32K_to_PLL0

Attach OSC32K to PLL0.

enumerator kNONE_to_PLL0
Attach NONE to PLL0.

enumerator kMAIN_CLK_to_ADC_CLK
Attach MAIN_CLK to ADC_CLK.

enumerator kPLL0_to_ADC_CLK
Attach PLL0 to ADC_CLK.

enumerator kFRO_HF_to_ADC_CLK
Attach FRO_HF to ADC_CLK.

enumerator kNONE_to_ADC_CLK
Attach NONE to ADC_CLK.

enumerator kMAIN_CLK_to_USB0_CLK
Attach MAIN_CLK to USB0_CLK.

enumerator kPLL0_to_USB0_CLK
Attach PLL0 to USB0_CLK.

enumerator kFRO_HF_to_USB0_CLK
Attach FRO_HF to USB0_CLK.

enumerator kPLL1_to_USB0_CLK
Attach PLL1 to USB0_CLK.

enumerator kNONE_to_USB0_CLK
Attach NONE to USB0_CLK.

enumerator kMAIN_CLK_to_FLEXCOMM0
Attach MAIN_CLK to FLEXCOMM0.

enumerator kPLL0_DIV_to_FLEXCOMM0
Attach PLL0_DIV to FLEXCOMM0.

enumerator kFRO12M_to_FLEXCOMM0
Attach FRO12M to FLEXCOMM0.

enumerator kFRO_HF_DIV_to_FLEXCOMM0
Attach FRO_HF_DIV to FLEXCOMM0.

enumerator kFRO1M_to_FLEXCOMM0
Attach FRO1M to FLEXCOMM0.

enumerator kMCLK_to_FLEXCOMM0
Attach MCLK to FLEXCOMM0.

enumerator kOSC32K_to_FLEXCOMM0
Attach OSC32K to FLEXCOMM0.

enumerator kNONE_to_FLEXCOMM0
Attach NONE to FLEXCOMM0.

enumerator kMAIN_CLK_to_FLEXCOMM1
Attach MAIN_CLK to FLEXCOMM1.

enumerator kPLL0_DIV_to_FLEXCOMM1
Attach PLL0_DIV to FLEXCOMM1.

enumerator kFRO12M_to_FLEXCOMM1
Attach FRO12M to FLEXCOMM1.

enumerator kFRO_HF_DIV_to_FLEXCOMM1

Attach FRO_HF_DIV to FLEXCOMM1.

enumerator kFRO1M_to_FLEXCOMM1

Attach FRO1M to FLEXCOMM1.

enumerator kMCLK_to_FLEXCOMM1

Attach MCLK to FLEXCOMM1.

enumerator kOSC32K_to_FLEXCOMM1

Attach OSC32K to FLEXCOMM1.

enumerator kNONE_to_FLEXCOMM1

Attach NONE to FLEXCOMM1.

enumerator kMAIN_CLK_to_FLEXCOMM2

Attach MAIN_CLK to FLEXCOMM2.

enumerator kPLL0_DIV_to_FLEXCOMM2

Attach PLL0_DIV to FLEXCOMM2.

enumerator kFRO12M_to_FLEXCOMM2

Attach FRO12M to FLEXCOMM2.

enumerator kFRO_HF_DIV_to_FLEXCOMM2

Attach FRO_HF_DIV to FLEXCOMM2.

enumerator kFRO1M_to_FLEXCOMM2

Attach FRO1M to FLEXCOMM2.

enumerator kMCLK_to_FLEXCOMM2

Attach MCLK to FLEXCOMM2.

enumerator kOSC32K_to_FLEXCOMM2

Attach OSC32K to FLEXCOMM2.

enumerator kNONE_to_FLEXCOMM2

Attach NONE to FLEXCOMM2.

enumerator kMAIN_CLK_to_FLEXCOMM3

Attach MAIN_CLK to FLEXCOMM3.

enumerator kPLL0_DIV_to_FLEXCOMM3

Attach PLL0_DIV to FLEXCOMM3.

enumerator kFRO12M_to_FLEXCOMM3

Attach FRO12M to FLEXCOMM3.

enumerator kFRO_HF_DIV_to_FLEXCOMM3

Attach FRO_HF_DIV to FLEXCOMM3.

enumerator kFRO1M_to_FLEXCOMM3

Attach FRO1M to FLEXCOMM3.

enumerator kMCLK_to_FLEXCOMM3

Attach MCLK to FLEXCOMM3.

enumerator kOSC32K_to_FLEXCOMM3

Attach OSC32K to FLEXCOMM3.

enumerator kNONE_to_FLEXCOMM3

Attach NONE to FLEXCOMM3.

enumerator kMAIN_CLK_to_FLEXCOMM4

Attach MAIN_CLK to FLEXCOMM4.

enumerator kPLL0_DIV_to_FLEXCOMM4

Attach PLL0_DIV to FLEXCOMM4.

enumerator kFRO12M_to_FLEXCOMM4

Attach FRO12M to FLEXCOMM4.

enumerator kFRO_HF_DIV_to_FLEXCOMM4

Attach FRO_HF_DIV to FLEXCOMM4.

enumerator kFRO1M_to_FLEXCOMM4

Attach FRO1M to FLEXCOMM4.

enumerator kMCLK_to_FLEXCOMM4

Attach MCLK to FLEXCOMM4.

enumerator kOSC32K_to_FLEXCOMM4

Attach OSC32K to FLEXCOMM4.

enumerator kNONE_to_FLEXCOMM4

Attach NONE to FLEXCOMM4.

enumerator kMAIN_CLK_to_FLEXCOMM5

Attach MAIN_CLK to FLEXCOMM5.

enumerator kPLL0_DIV_to_FLEXCOMM5

Attach PLL0_DIV to FLEXCOMM5.

enumerator kFRO12M_to_FLEXCOMM5

Attach FRO12M to FLEXCOMM5.

enumerator kFRO_HF_DIV_to_FLEXCOMM5

Attach FRO_HF_DIV to FLEXCOMM5.

enumerator kFRO1M_to_FLEXCOMM5

Attach FRO1M to FLEXCOMM5.

enumerator kMCLK_to_FLEXCOMM5

Attach MCLK to FLEXCOMM5.

enumerator kOSC32K_to_FLEXCOMM5

Attach OSC32K to FLEXCOMM5.

enumerator kNONE_to_FLEXCOMM5

Attach NONE to FLEXCOMM5.

enumerator kMAIN_CLK_to_FLEXCOMM6

Attach MAIN_CLK to FLEXCOMM6.

enumerator kPLL0_DIV_to_FLEXCOMM6

Attach PLL0_DIV to FLEXCOMM6.

enumerator kFRO12M_to_FLEXCOMM6

Attach FRO12M to FLEXCOMM6.

enumerator kFRO_HF_DIV_to_FLEXCOMM6

Attach FRO_HF_DIV to FLEXCOMM6.

enumerator kFRO1M_to_FLEXCOMM6

Attach FRO1M to FLEXCOMM6.

enumerator kMCLK_to_FLEXCOMM6
Attach MCLK to FLEXCOMM6.

enumerator kOSC32K_to_FLEXCOMM6
Attach OSC32K to FLEXCOMM6.

enumerator kNONE_to_FLEXCOMM6
Attach NONE to FLEXCOMM6.

enumerator kMAIN_CLK_to_FLEXCOMM7
Attach MAIN_CLK to FLEXCOMM7.

enumerator kPLL0_DIV_to_FLEXCOMM7
Attach PLL0_DIV to FLEXCOMM7.

enumerator kFRO12M_to_FLEXCOMM7
Attach FRO12M to FLEXCOMM7.

enumerator kFRO_HF_DIV_to_FLEXCOMM7
Attach FRO_HF_DIV to FLEXCOMM7.

enumerator kFRO1M_to_FLEXCOMM7
Attach FRO1M to FLEXCOMM7.

enumerator kMCLK_to_FLEXCOMM7
Attach MCLK to FLEXCOMM7.

enumerator kOSC32K_to_FLEXCOMM7
Attach OSC32K to FLEXCOMM7.

enumerator kNONE_to_FLEXCOMM7
Attach NONE to FLEXCOMM7.

enumerator kMAIN_CLK_to_HSLSPI
Attach MAIN_CLK to HSLSPI.

enumerator kPLL0_DIV_to_HSLSPI
Attach PLL0_DIV to HSLSPI.

enumerator kFRO12M_to_HSLSPI
Attach FRO12M to HSLSPI.

enumerator kFRO_HF_DIV_to_HSLSPI
Attach FRO_HF_DIV to HSLSPI.

enumerator kFRO1M_to_HSLSPI
Attach FRO1M to HSLSPI.

enumerator kOSC32K_to_HSLSPI
Attach OSC32K to HSLSPI.

enumerator kNONE_to_HSLSPI
Attach NONE to HSLSPI.

enumerator kFRO_HF_to_MCLK
Attach FRO_HF to MCLK.

enumerator kPLL0_to_MCLK
Attach PLL0 to MCLK.

enumerator kNONE_to_MCLK
Attach NONE to MCLK.

enumerator kMAIN_CLK_to_SCT_CLK
Attach MAIN_CLK to SCT_CLK.

enumerator kPLL0_to_SCT_CLK
Attach PLL0 to SCT_CLK.

enumerator kEXT_CLK_to_SCT_CLK
Attach EXT_CLK to SCT_CLK.

enumerator kFRO_HF_to_SCT_CLK
Attach FRO_HF to SCT_CLK.

enumerator kMCLK_to_SCT_CLK
Attach MCLK to SCT_CLK.

enumerator kNONE_to_SCT_CLK
Attach NONE to SCT_CLK.

enumerator kMAIN_CLK_to_SDIO_CLK
Attach MAIN_CLK to SDIO_CLK.

enumerator kPLL0_to_SDIO_CLK
Attach PLL0 to SDIO_CLK.

enumerator kFRO_HF_to_SDIO_CLK
Attach FRO_HF to SDIO_CLK.

enumerator kPLL1_to_SDIO_CLK
Attach PLL1 to SDIO_CLK.

enumerator kNONE_to_SDIO_CLK
Attach NONE to SDIO_CLK.

enumerator kFRO32K_to_OSC32K
Attach FRO32K to OSC32K.

enumerator kXTAL32K_to_OSC32K
Attach XTAL32K to OSC32K.

enumerator kTRACE_DIV_to_TRACE
Attach TRACE_DIV to TRACE.

enumerator kFRO1M_to_TRACE
Attach FRO1M to TRACE.

enumerator kOSC32K_to_TRACE
Attach OSC32K to TRACE.

enumerator kNONE_to_TRACE
Attach NONE to TRACE.

enumerator kSYSTICK_DIV0_to_SYSTICK0
Attach SYSTICK_DIV0 to SYSTICK0.

enumerator kFRO1M_to_SYSTICK0
Attach FRO1M to SYSTICK0.

enumerator kOSC32K_to_SYSTICK0
Attach OSC32K to SYSTICK0.

enumerator kNONE_to_SYSTICK0
Attach NONE to SYSTICK0.

enumerator kSYSTICK_DIV1_to_SYSTICK1

Attach SYSTICK_DIV1 to SYSTICK1.

enumerator kFRO1M_to_SYSTICK1

Attach FRO1M to SYSTICK1.

enumerator kOSC32K_to_SYSTICK1

Attach OSC32K to SYSTICK1.

enumerator kNONE_to_SYSTICK1

Attach NONE to SYSTICK1.

enumerator kFRO12M_to_PLL1

Attach FRO12M to PLL1.

enumerator kEXT_CLK_to_PLL1

Attach EXT_CLK to PLL1.

enumerator kFRO1M_to_PLL1

Attach FRO1M to PLL1.

enumerator kOSC32K_to_PLL1

Attach OSC32K to PLL1.

enumerator kNONE_to_PLL1

Attach NONE to PLL1.

enumerator kMAIN_CLK_to_CTIMER0

Attach MAIN_CLK to CTIMER0.

enumerator kPLL0_to_CTIMER0

Attach PLL0 to CTIMER0.

enumerator kFRO_HF_to_CTIMER0

Attach FRO_HF to CTIMER0.

enumerator kFRO1M_to_CTIMER0

Attach FRO1M to CTIMER0.

enumerator kMCLK_to_CTIMER0

Attach MCLK to CTIMER0.

enumerator kOSC32K_to_CTIMER0

Attach OSC32K to CTIMER0.

enumerator kNONE_to_CTIMER0

Attach NONE to CTIMER0.

enumerator kMAIN_CLK_to_CTIMER1

Attach MAIN_CLK to CTIMER1.

enumerator kPLL0_to_CTIMER1

Attach PLL0 to CTIMER1.

enumerator kFRO_HF_to_CTIMER1

Attach FRO_HF to CTIMER1.

enumerator kFRO1M_to_CTIMER1

Attach FRO1M to CTIMER1.

enumerator kMCLK_to_CTIMER1

Attach MCLK to CTIMER1.

enumerator kOSC32K_to_CTIMER1
Attach OSC32K to CTIMER1.

enumerator kNONE_to_CTIMER1
Attach NONE to CTIMER1.

enumerator kMAIN_CLK_to_CTIMER2
Attach MAIN_CLK to CTIMER2.

enumerator kPLL0_to_CTIMER2
Attach PLL0 to CTIMER2.

enumerator kFRO_HF_to_CTIMER2
Attach FRO_HF to CTIMER2.

enumerator kFRO1M_to_CTIMER2
Attach FRO1M to CTIMER2.

enumerator kMCLK_to_CTIMER2
Attach MCLK to CTIMER2.

enumerator kOSC32K_to_CTIMER2
Attach OSC32K to CTIMER2.

enumerator kNONE_to_CTIMER2
Attach NONE to CTIMER2.

enumerator kMAIN_CLK_to_CTIMER3
Attach MAIN_CLK to CTIMER3.

enumerator kPLL0_to_CTIMER3
Attach PLL0 to CTIMER3.

enumerator kFRO_HF_to_CTIMER3
Attach FRO_HF to CTIMER3.

enumerator kFRO1M_to_CTIMER3
Attach FRO1M to CTIMER3.

enumerator kMCLK_to_CTIMER3
Attach MCLK to CTIMER3.

enumerator kOSC32K_to_CTIMER3
Attach OSC32K to CTIMER3.

enumerator kNONE_to_CTIMER3
Attach NONE to CTIMER3.

enumerator kMAIN_CLK_to_CTIMER4
Attach MAIN_CLK to CTIMER4.

enumerator kPLL0_to_CTIMER4
Attach PLL0 to CTIMER4.

enumerator kFRO_HF_to_CTIMER4
Attach FRO_HF to CTIMER4.

enumerator kFRO1M_to_CTIMER4
Attach FRO1M to CTIMER4.

enumerator kMCLK_to_CTIMER4
Attach MCLK to CTIMER4.

enumerator kOSC32K_to_CTIMER4

Attach OSC32K to CTIMER4.

enumerator kNONE_to_CTIMER4

Attach NONE to CTIMER4.

enumerator kNONE_to_NONE

Attach NONE to NONE.

enum _clock_div_name

Clock dividers.

Values:

enumerator kCLOCK_DivSystickClk0

Systick Clk0 Divider.

enumerator kCLOCK_DivSystickClk1

Systick Clk1 Divider.

enumerator kCLOCK_DivArmTrClkDiv

Arm Tr Clk Div Divider.

enumerator kCLOCK_DivFlexFrg0

Flex Frg0 Divider.

enumerator kCLOCK_DivFlexFrg1

Flex Frg1 Divider.

enumerator kCLOCK_DivFlexFrg2

Flex Frg2 Divider.

enumerator kCLOCK_DivFlexFrg3

Flex Frg3 Divider.

enumerator kCLOCK_DivFlexFrg4

Flex Frg4 Divider.

enumerator kCLOCK_DivFlexFrg5

Flex Frg5 Divider.

enumerator kCLOCK_DivFlexFrg6

Flex Frg6 Divider.

enumerator kCLOCK_DivFlexFrg7

Flex Frg7 Divider.

enumerator kCLOCK_DivAhbClk

Ahb Clock Divider.

enumerator kCLOCK_DivClkOut

Clk Out Divider.

enumerator kCLOCK_DivFrohfClk

Frohf Clock Divider.

enumerator kCLOCK_DivWdtClk

Wdt Clock Divider.

enumerator kCLOCK_DivAdcAsyncClk

Adc Async Clock Divider.

enumerator kCLOCK_DivUsb0Clk
 Usb0 Clock Divider.

enumerator kCLOCK_DivMClk
 I2S MCLK Clock Divider.

enumerator kCLOCK_DivSctClk
 Sct Clock Divider.

enumerator kCLOCK_DivSdioClk
 Sdio Clock Divider.

enumerator kCLOCK_DivPll0Clk
 PLL clock divider.

enum _ss_progmodfm

PLL Spread Spectrum (SS) Programmable modulation frequency See (MF) field in the PLL0SSCG1 register in the UM.

Values:

enumerator kSS_MF_512
 Nss = 512 (fm ? 3.9 - 7.8 kHz)

enumerator kSS_MF_384
 Nss ?= 384 (fm ? 5.2 - 10.4 kHz)

enumerator kSS_MF_256
 Nss = 256 (fm ? 7.8 - 15.6 kHz)

enumerator kSS_MF_128
 Nss = 128 (fm ? 15.6 - 31.3 kHz)

enumerator kSS_MF_64
 Nss = 64 (fm ? 32.3 - 64.5 kHz)

enumerator kSS_MF_32
 Nss = 32 (fm ? 62.5- 125 kHz)

enumerator kSS_MF_24
 Nss ?= 24 (fm ? 83.3- 166.6 kHz)

enumerator kSS_MF_16
 Nss = 16 (fm ? 125- 250 kHz)

enum _ss_progmoddp

PLL Spread Spectrum (SS) Programmable frequency modulation depth See (MR) field in the PLL0SSCG1 register in the UM.

Values:

enumerator kSS_MR_K0
 k = 0 (no spread spectrum)

enumerator kSS_MR_K1
 k = 1

enumerator kSS_MR_K1_5
 k = 1.5

enumerator kSS_MR_K2
 k = 2

enumerator kSS_MR_K3

k = 3

enumerator kSS_MR_K4

k = 4

enumerator kSS_MR_K6

k = 6

enumerator kSS_MR_K8

k = 8

enum __ss_modwvctrl

PLL Spread Spectrum (SS) Modulation waveform control See (MC) field in the PLL0SSCG1 register in the UM.

Compensation for low pass filtering of the PLL to get a triangular modulation at the output of the PLL, giving a flat frequency spectrum.

Values:

enumerator kSS_MC_NOC

no compensation

enumerator kSS_MC_RECC

recommended setting

enumerator kSS_MC_MAXC

max. compensation

enum __pll_error

PLL status definitions.

Values:

enumerator kStatus_PLL_Success

PLL operation was successful

enumerator kStatus_PLL_OutputTooLow

PLL output rate request was too low

enumerator kStatus_PLL_OutputTooHigh

PLL output rate request was too high

enumerator kStatus_PLL_InputTooLow

PLL input rate is too low

enumerator kStatus_PLL_InputTooHigh

PLL input rate is too high

enumerator kStatus_PLL_OutsideIntLimit

Requested output rate isn't possible

enumerator kStatus_PLL_CCOTooLow

Requested CCO rate isn't possible

enumerator kStatus_PLL_CCOTooHigh

Requested CCO rate isn't possible

enum __clock_usbfs_src

USB FS clock source definition.

Values:

enumerator kCLOCK__UsbfsSrcFro

Use FRO 96 MHz.

enumerator kCLOCK__UsbfsSrcPll0

Use PLL0 output.

enumerator kCLOCK__UsbfsSrcMainClock

Use Main clock.

enumerator kCLOCK__UsbfsSrcPll1

Use PLL1 clock.

enumerator kCLOCK__UsbfsSrcNone

this may be selected in order to reduce power when no output is needed.

enum _clock_usbhs_src

USBhs clock source definition.

Values:

enumerator kCLOCK__UsbSrcUnused

Used when the function does not care the clock source.

enum _clock_usb_phy_src

Source of the USB HS PHY.

Values:

enumerator kCLOCK__UsbPhySrcExt

Use external crystal.

typedef enum _clock_ip_name clock_ip_name_t

Clock gate name used for CLOCK_EnableClock/CLOCK_DisableClock.

typedef enum _clock_name clock_name_t

Clock name used to get clock frequency.

typedef enum _clock_attach_id clock_attach_id_t

The enumerator of clock attach Id.

typedef enum _clock_div_name clock_div_name_t

Clock dividers.

typedef enum _ss_progmodfm ss_progmodfm_t

PLL Spread Spectrum (SS) Programmable modulation frequency See (MF) field in the PLL0SSCG1 register in the UM.

typedef enum _ss_progmoddp ss_progmoddp_t

PLL Spread Spectrum (SS) Programmable frequency modulation depth See (MR) field in the PLL0SSCG1 register in the UM.

typedef enum _ss_modwvctrl ss_modwvctrl_t

PLL Spread Spectrum (SS) Modulation waveform control See (MC) field in the PLL0SSCG1 register in the UM.

Compensation for low pass filtering of the PLL to get a triangular modulation at the output of the PLL, giving a flat frequency spectrum.

typedef struct _pll_config pll_config_t

PLL configuration structure.

This structure can be used to configure the settings for a PLL setup structure. Fill in the desired configuration for the PLL and call the PLL setup function to fill in a PLL setup structure.

typedef struct *_pll_setup* pll_setup_t

PLL0 setup structure This structure can be used to pre-build a PLL setup configuration at run-time and quickly set the PLL to the configuration. It can be populated with the PLL setup function. If powering up or waiting for PLL lock, the PLL input clock source should be configured prior to PLL setup.

typedef enum *_pll_error* pll_error_t

PLL status definitions.

typedef enum *_clock_usbfs_src* clock_usbfs_src_t

USB FS clock source definition.

typedef enum *_clock_usbhs_src* clock_usbhs_src_t

USBhs clock source definition.

typedef enum *_clock_usb_phy_src* clock_usb_phy_src_t

Source of the USB HS PHY.

static inline void CLOCK_EnableClock(*clock_ip_name_t* clk)

Enable the clock for specific IP.

Parameters

- clk – : Clock to be enabled.

Returns

Nothing

static inline void CLOCK_DisableClock(*clock_ip_name_t* clk)

Disable the clock for specific IP.

Parameters

- clk – : Clock to be Disabled.

Returns

Nothing

status_t CLOCK_SetupFROClocking(uint32_t iFreq)

Initialize the Core clock to given frequency (12, 48 or 96 MHz). Turns on FRO and uses default CCO, if freq is 12000000, then high speed output is off, else high speed output is enabled.

Parameters

- iFreq – : Desired frequency (must be one of CLK_FRO_12MHZ or CLK_FRO_48MHZ or CLK_FRO_96MHZ)

Returns

returns success or fail status.

void CLOCK_SetFLASHAccessCyclesForFreq(uint32_t iFreq)

Set the flash wait states for the input frequency.

Parameters

- iFreq – : Input frequency

Returns

Nothing

status_t CLOCK_SetupExtClocking(uint32_t iFreq)

Initialize the external osc clock to given frequency.

Parameters

- iFreq – : Desired frequency (must be equal to exact rate in Hz)

Returns

returns success or fail status.

status_t CLOCK_SetupI2SMClkClocking(*uint32_t* iFreq)

Initialize the I2S MCLK clock to given frequency.

Parameters

- iFreq – : Desired frequency (must be equal to exact rate in Hz)

Returns

returns success or fail status.

status_t CLOCK_SetupPLUClkInClocking(*uint32_t* iFreq)

Initialize the PLU CLKIN clock to given frequency.

Parameters

- iFreq – : Desired frequency (must be equal to exact rate in Hz)

Returns

returns success or fail status.

void CLOCK_AttachClk(*clock_attach_id_t* connection)

Configure the clock selection muxes.

Parameters

- connection – : Clock to be configured.

Returns

Nothing

clock_attach_id_t CLOCK_GetClockAttachId(*clock_attach_id_t* attachId)

Get the actual clock attach id. This function uses the offset in input attach id, then it reads the actual source value in the register and combine the offset to obtain an actual attach id.

Parameters

- attachId – : Clock attach id to get.

Returns

Clock source value.

void CLOCK_SetClkDiv(*clock_div_name_t* div_name, *uint32_t* divided_by_value, *bool* reset)

Setup peripheral clock dividers.

Parameters

- div_name – : Clock divider name
- divided_by_value – Value to be divided
- reset – : Whether to reset the divider counter.

Returns

Nothing

void CLOCK_SetRtc1khzClkDiv(*uint32_t* divided_by_value)

Setup rtc 1khz clock divider.

Parameters

- divided_by_value – Value to be divided

Returns

Nothing

void CLOCK_SetRtc1hzClkDiv(uint32_t divided_by_value)

Setup rtc 1hz clock divider.

Parameters

- divided_by_value – Value to be divided

Returns

Nothing

uint32_t CLOCK_SetFlexCommClock(uint32_t id, uint32_t freq)

Set the flexcomm output frequency.

Parameters

- id – : flexcomm instance id
- freq – : output frequency

Returns

0 : the frequency range is out of range. 1 : switch successfully.

uint32_t CLOCK_GetFlexCommInputClock(uint32_t id)

Return Frequency of flexcomm input clock.

Parameters

- id – : flexcomm instance id

Returns

Frequency value

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Return Frequency of selected clock.

Returns

Frequency of selected clock

uint32_t CLOCK_GetFro12MFreq(void)

Return Frequency of FRO 12MHz.

Returns

Frequency of FRO 12MHz

uint32_t CLOCK_GetFro1MFreq(void)

Return Frequency of FRO 1MHz.

Returns

Frequency of FRO 1MHz

uint32_t CLOCK_GetClockOutClkFreq(void)

Return Frequency of ClockOut.

Returns

Frequency of ClockOut

uint32_t CLOCK_GetAdcClkFreq(void)

Return Frequency of Adc Clock.

Returns

Frequency of Adc.

uint32_t CLOCK_GetUsb0ClkFreq(void)

Return Frequency of Usb0 Clock.

Returns

Frequency of Usb0 Clock.

uint32_t CLOCK_GetUsb1ClkFreq(void)

Return Frequency of Usb1 Clock.

Returns

Frequency of Usb1 Clock.

uint32_t CLOCK_GetMclkClkFreq(void)

Return Frequency of MClk Clock.

Returns

Frequency of MClk Clock.

uint32_t CLOCK_GetSctClkFreq(void)

Return Frequency of SCTimer Clock.

Returns

Frequency of SCTimer Clock.

uint32_t CLOCK_GetSdioClkFreq(void)

Return Frequency of SDIO Clock.

Returns

Frequency of SDIO Clock.

uint32_t CLOCK_GetExtClkFreq(void)

Return Frequency of External Clock.

Returns

Frequency of External Clock. If no external clock is used returns 0.

uint32_t CLOCK_GetWdtClkFreq(void)

Return Frequency of Watchdog.

Returns

Frequency of Watchdog

uint32_t CLOCK_GetFroHfFreq(void)

Return Frequency of High-Freq output of FRO.

Returns

Frequency of High-Freq output of FRO

uint32_t CLOCK_GetPll0OutFreq(void)

Return Frequency of PLL.

Returns

Frequency of PLL

uint32_t CLOCK_GetPll1OutFreq(void)

Return Frequency of USB PLL.

Returns

Frequency of PLL

uint32_t CLOCK_GetOsc32KFreq(void)

Return Frequency of 32kHz osc.

Returns

Frequency of 32kHz osc

uint32_t CLOCK_GetCoreSysClkFreq(void)

Return Frequency of Core System.

Returns

Frequency of Core System

uint32_t CLOCK_GetI2SMClkFreq(void)

Return Frequency of I2S MCLK Clock.

Returns

Frequency of I2S MCLK Clock

uint32_t CLOCK_GetPLUClkInFreq(void)

Return Frequency of PLU CLKIN Clock.

Returns

Frequency of PLU CLKIN Clock

uint32_t CLOCK_GetFlexCommClkFreq(uint32_t id)

Return Frequency of FlexComm Clock.

Returns

Frequency of FlexComm Clock

uint32_t CLOCK_GetHsLspiClkFreq(void)

Return Frequency of High speed SPI Clock.

Returns

Frequency of High speed SPI Clock

uint32_t CLOCK_GetCTimerClkFreq(uint32_t id)

Return Frequency of CTimer functional Clock.

Returns

Frequency of CTimer functional Clock

uint32_t CLOCK_GetSystickClkFreq(uint32_t id)

Return Frequency of SystickClock.

Returns

Frequency of Systick Clock

uint32_t CLOCK_GetPLL0InClockRate(void)

Return PLL0 input clock rate.

Returns

PLL0 input clock rate

uint32_t CLOCK_GetPLL1InClockRate(void)

Return PLL1 input clock rate.

Returns

PLL1 input clock rate

uint32_t CLOCK_GetPLL0OutClockRate(bool recompute)

Return PLL0 output clock rate.

Note: The PLL rate is cached in the driver in a variable as the rate computation function can take some time to perform. It is recommended to use 'false' with the 'recompute' parameter.

Parameters

- recompute – : Forces a PLL rate recomputation if true

Returns

PLL0 output clock rate

__STATIC_INLINE void CLOCK_SetBypassPLL0 (bool bypass)

Enables and disables PLL0 bypass mode.

bypass : true to bypass PLL0 (PLL0 output = PLL0 input, false to disable bypass)

Returns

PLL0 output clock rate

__STATIC_INLINE void CLOCK_SetBypassPLL1 (bool bypass)

Enables and disables PLL1 bypass mode.

bypass : true to bypass PLL1 (PLL1 output = PLL1 input, false to disable bypass)

Returns

PLL1 output clock rate

__STATIC_INLINE bool CLOCK_IsPLL0Locked (void)

Check if PLL is locked or not.

Returns

true if the PLL is locked, false if not locked

__STATIC_INLINE bool CLOCK_IsPLL1Locked (void)

Check if PLL1 is locked or not.

Returns

true if the PLL1 is locked, false if not locked

void CLOCK_SetStoredPLL0ClockRate(uint32_t rate)

Store the current PLL0 rate.

Parameters

- rate – Current rate of the PLL0

Returns

Nothing

uint32_t CLOCK_GetPLL0OutFromSetup(*pll_setup_t* *pSetup)

Return PLL0 output clock rate from setup structure.

Parameters

- pSetup – : Pointer to a PLL setup structure

Returns

System PLL output clock rate the setup structure will generate

pll_error_t CLOCK_SetupPLL0Data(*pll_config_t* *pControl, *pll_setup_t* *pSetup)

Set PLL0 output based on the passed PLL setup data.

Note: Actual frequency for setup may vary from the desired frequency based on the accuracy of input clocks, rounding, non-fractional PLL mode, etc.

Parameters

- pControl – : Pointer to populated PLL control structure to generate setup with
- pSetup – : Pointer to PLL setup structure to be filled

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

pll_error_t CLOCK_SetupPLL0Prec(*pll_setup_t* *pSetup, uint32_t flagcfg)

Set PLL output from PLL setup structure (precise frequency)

Note: This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- pSetup – : Pointer to populated PLL setup structure
- flagcfg – : Flag configuration for PLL config structure

Returns

PLL_ERROR_SUCCESS on success, or PLL setup error code

pll_error_t CLOCK_SetPLL0Freq(const *pll_setup_t* *pSetup)

Set PLL output from PLL setup structure (precise frequency)

Note: This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- pSetup – : Pointer to populated PLL setup structure

Returns

kStatus_PLL_Success on success, or PLL setup error code

pll_error_t CLOCK_SetPLL1Freq(const *pll_setup_t* *pSetup)

Set PLL output from PLL setup structure (precise frequency)

Note: This function will power off the PLL, setup the PLL with the new setup data, and then optionally powerup the PLL, wait for PLL lock, and adjust system voltages to the new PLL rate. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- pSetup – : Pointer to populated PLL setup structure

Returns

kStatus_PLL_Success on success, or PLL setup error code

void CLOCK_SetupPLL0Mult(uint32_t multiply_by, uint32_t input_freq)

Set PLL0 output based on the multiplier and input frequency.

Note: Unlike the Chip_Clock_SetupSystemPLLPrec() function, this function does not disable or enable PLL power, wait for PLL lock, or adjust system voltages. These must be done in the application. The function will not alter any source clocks (ie, main system clock) that may use the PLL, so these should be setup prior to and after exiting the function.

Parameters

- `multiply_by` – : multiplier
- `input_freq` – : Clock input frequency of the PLL

Returns

Nothing

`static inline void CLOCK_DisableUsbDevicefs0Clock(clock_ip_name_t clk)`

Disable USB clock.

Disable USB clock.

`bool CLOCK_EnableUsbfs0DeviceClock(clock_usbfs_src_t src, uint32_t freq)`

Enable USB Device FS clock.

Parameters

- `src` – : clock source
- `freq` – clock frequency Enable USB Device Full Speed clock.

`bool CLOCK_EnableUsbfs0HostClock(clock_usbfs_src_t src, uint32_t freq)`

Enable USB HOST FS clock.

Parameters

- `src` – : clock source
- `freq` – clock frequency Enable USB HOST Full Speed clock.

`bool CLOCK_EnableUsbhs0PhyPllClock(clock_usb_phy_src_t src, uint32_t freq)`

Enable USB phy clock. Enable USB phy clock.

`bool CLOCK_EnableUsbhs0DeviceClock(clock_usbhs_src_t src, uint32_t freq)`

Enable USB Device HS clock. Enable USB Device High Speed clock.

`bool CLOCK_EnableUsbhs0HostClock(clock_usbhs_src_t src, uint32_t freq)`

Enable USB HOST HS clock. Enable USB HOST High Speed clock.

`void CLOCK_EnableOstimer32kClock(void)`

Enable the OSTIMER 32k clock.

Returns

Nothing

`FSL_CLOCK_DRIVER_VERSION`

CLOCK driver version 2.3.8.

`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.

`CLOCK_USR_CFG_PLL_CONFIG_CACHE_COUNT`

User-defined the size of cache for `CLOCK_PllGetConfig()` function.

Once define this MACRO to be non-zero value, `CLOCK_PllGetConfig()` function would cache the recent calculation and accelerate the execution to get the right settings.

`SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY`

ROM_CLOCKS

Clock ip name array for ROM.

SRAM_CLOCKS

Clock ip name array for SRAM.

FLASH_CLOCKS

Clock ip name array for FLASH.

FMC_CLOCKS

Clock ip name array for FMC.

INPUTMUX_CLOCKS

Clock ip name array for INPUTMUX.

IOCON_CLOCKS

Clock ip name array for IOCON.

GPIO_CLOCKS

Clock ip name array for GPIO.

PINT_CLOCKS

Clock ip name array for PINT.

GINT_CLOCKS

Clock ip name array for GINT.

DMA_CLOCKS

Clock ip name array for DMA.

CRC_CLOCKS

Clock ip name array for CRC.

WWDT_CLOCKS

Clock ip name array for WWDT.

RTC_CLOCKS

Clock ip name array for RTC.

MAILBOX_CLOCKS

Clock ip name array for Mailbox.

LPADC_CLOCKS

Clock ip name array for LPADC.

MRT_CLOCKS

Clock ip name array for MRT.

OSTIMER_CLOCKS

Clock ip name array for OSTIMER.

SCT_CLOCKS

Clock ip name array for SCT0.

UTICK_CLOCKS

Clock ip name array for UTICK.

FLEXCOMM_CLOCKS

Clock ip name array for FLEXCOMM.

LPUART_CLOCKS

Clock ip name array for LPUART.

BI2C_CLOCKS

Clock ip name array for BI2C.

LPSPi_CLOCKS

Clock ip name array for LSPi.

FLEXI2S_CLOCKS

Clock ip name array for FLEXI2S.

CTIMER_CLOCKS

Clock ip name array for CTIMER.

COMP_CLOCKS

Clock ip name array for COMP.

SDIO_CLOCKS

Clock ip name array for SDIO.

USB1CLK_CLOCKS

Clock ip name array for USB1CLK.

FREQME_CLOCKS

Clock ip name array for FREQME.

USBRAM_CLOCKS

Clock ip name array for USBRAM.

RNG_CLOCKS

Clock ip name array for RNG.

USBHMR0_CLOCKS

Clock ip name array for USBHMR0.

USBHSL0_CLOCKS

Clock ip name array for USBHSL0.

HASHCRYPT_CLOCKS

Clock ip name array for HashCrypt.

POWERQUAD_CLOCKS

Clock ip name array for PowerQuad.

PLULUT_CLOCKS

Clock ip name array for PLULUT.

PUF_CLOCKS

Clock ip name array for PUF.

CASPER_CLOCKS

Clock ip name array for CASPER.

ANALOGCTRL_CLOCKS

Clock ip name array for ANALOGCTRL.

HS_LSPi_CLOCKS

Clock ip name array for HS_LSPi.

GPIO_SEC_CLOCKS

Clock ip name array for GPIO_SEC.

GPIO_SEC_INT_CLOCKS

Clock ip name array for GPIO_SEC_INT.

USBD_CLOCKS

Clock ip name array for USB.

USBH_CLOCKS

Clock ip name array for USBH.

PLU_CLOCKS

SYSCTL_CLOCKS

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)

AHB_CLK_CTRL0

AHB_CLK_CTRL1

AHB_CLK_CTRL2

BUS_CLK

Peripherals clock source definition.

I2C0_CLK_SRC

CLK_ATTACH_ID(mux, sel, pos)

Clock Mux Switches The encoding is as follows each connection identified is 32bits wide while 24bits are valuable starting from LSB upwards.

[4 bits for choice, 0 means invalid choice] [8 bits mux ID]*

MUX_A(mux, sel)

MUX_B(mux, sel, selector)

GET_ID_ITEM(connection)

GET_ID_NEXT_ITEM(connection)

GET_ID_ITEM_MUX(connection)

GET_ID_ITEM_SEL(connection)

GET_ID_SELECTOR(connection)

CM_SYSTICKCLKSEL0

CM_SYSTICKCLKSEL1

CM_TRACECLKSEL

CM_CTIMERCLKSEL0

CM_CTIMERCLKSEL1

CM_CTIMERCLKSEL2
CM_CTIMERCLKSEL3
CM_CTIMERCLKSEL4
CM_MAINCLKSELA
CM_MAINCLKSELB
CM_CLKOUTCLKSEL
CM_PLL0CLKSEL
CM_PLL1CLKSEL
CM_ADCASYNCCLKSEL
CM_USB0CLKSEL
CM_FXCOMCLKSEL0
CM_FXCOMCLKSEL1
CM_FXCOMCLKSEL2
CM_FXCOMCLKSEL3
CM_FXCOMCLKSEL4
CM_FXCOMCLKSEL5
CM_FXCOMCLKSEL6
CM_FXCOMCLKSEL7
CM_HSLSPICLKSEL
CM_MCLKCLKSEL
CM_SCTCLKSEL
CM_SDIOCLKSEL
CM_RTCOSC32KCLKSEL

PLL_CONFIGFLAG_USEINRATE

PLL configuration structure flags for 'flags' field These flags control how the PLL configuration function sets up the PLL setup structure.

When the PLL_CONFIGFLAG_USEINRATE flag is selected, the 'InputRate' field in the configuration structure must be assigned with the expected PLL frequency. If the PLL_CONFIGFLAG_USEINRATE is not used, 'InputRate' is ignored in the configuration function and the driver will determine the PLL rate from the currently selected PLL source. This flag might be used to configure the PLL input clock more accurately when using the WDT oscillator or a more dyanmic CLKIN source.

When the PLL_CONFIGFLAG_FORCENOFRACT flag is selected, the PLL hardware for the automatic bandwidth selection, Spread Spectrum (SS) support, and fractional M-divider are not used.

Flag to use InputRate in PLL configuration structure for setup

PLL_CONFIGFLAG_FORCENOFRACT

Force non-fractional output mode, PLL output will not use the fractional, automatic bandwidth, or SS hardware

PLL_SETUPFLAG_POWERUP

PLL setup structure flags for 'flags' field These flags control how the PLL setup function sets up the PLL.

Setup will power on the PLL after setup

PLL_SETUPFLAG_WAITLOCK

Setup will wait for PLL lock, implies the PLL will be powered on

PLL_SETUPFLAG_ADGVOLT

Optimize system voltage for the new PLL rate

PLL_SETUPFLAG_USEFEEDBACKDIV2

Use feedback divider by 2 in divider path

uint32_t desiredRate

Desired PLL rate in Hz

uint32_t inputRate

PLL input clock in Hz, only used if PLL_CONFIGFLAG_USEINRATE flag is set

uint32_t flags

PLL configuration flags, Or'ed value of PLL_CONFIGFLAG_* definitions

ss_progmodfm_t ss_mf

SS Programmable modulation frequency, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

ss_progmoddp_t ss_mr

SS Programmable frequency modulation depth, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

ss_modwvctrl_t ss_mc

SS Modulation waveform control, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

bool mfDither

false for fixed modulation frequency or true for dithering, only applicable when not using PLL_CONFIGFLAG_FORCENOFRACT flag

uint32_t pllctrl

PLL control register PLL0CTRL

uint32_t pllndec

PLL NDEC register PLL0NDEC

uint32_t pllpedec

PLL PDEC register PLL0PDEC

uint32_t pllmddec

PLL MDEC registers PLL0PDEC

uint32_t pllsscg[2]

PLL SSCTL registers PLL0SSCG

uint32_t pllRate

Actual PLL rate

uint32_t flags

PLL setup flags, Or'ed value of PLL_SETUPFLAG_* definitions

struct __pll_config

#include <fsl_clock.h> PLL configuration structure.

This structure can be used to configure the settings for a PLL setup structure. Fill in the desired configuration for the PLL and call the PLL setup function to fill in a PLL setup structure.

struct __pll_setup

#include <fsl_clock.h> PLL0 setup structure This structure can be used to pre-build a PLL setup configuration at run-time and quickly set the PLL to the configuration. It can be populated with the PLL setup function. If powering up or waiting for PLL lock, the PLL input clock source should be configured prior to PLL setup.

2.6 CMP: Analog Comparator Driver

void CMP_Init(const *cmp_config_t* *config)

CMP initialization.

This function enables the CMP module and do necessary settings.

Parameters

- config – Pointer to the configuration structure.

void CMP_Deinit(void)

CMP deinitialization.

This function gates the clock for CMP module.

void CMP_GetDefaultConfig(*cmp_config_t* *config)

Initializes the CMP user configuration structure.

This function initializes the user configuration structure to these default values.

```
config->enableHysteresis = true;
config->enableLowPower   = true;
config->filterClockDivider = kCMP_FilterClockDivide1;
config->filterSampleMode  = kCMP_FilterSampleMode0;
```

Parameters

- config – Pointer to the configuration structure.

static inline void CMP_SetInputChannels(uint8_t positiveChannel, uint8_t negativeChannel)

void CMP_SetVREF(const *cmp_vref_config_t* *config)

Configures the VREFINPUT.

Parameters

- config – Pointer to the configuration structure.

static inline bool CMP_GetOutput(void)

Get CMP compare output.

Returns

The output result. true: voltage on positive side is greater than negative side.
false: voltage on positive side is lower than negative side.

static inline void CMP_EnableInterrupt(uint32_t type)

CMP enable interrupt.

Parameters

- type – CMP interrupt type. See “_cmp_interrupt_type”.

static inline void CMP_DisableInterrupt(void)

CMP disable interrupt.

static inline void CMP_ClearInterrupt(void)

CMP clear interrupt.

static inline void CMP_EnableFilteredInterruptSource(bool enable)

Select which Analog comparator output (filtered or un-filtered) is used for interrupt detection.

Note: : When CMP is configured as the wakeup source in power down mode, this function must use the raw output as the interrupt source, that is, call this function and set parameter enable to false.

Parameters

- enable – false: Select Analog Comparator raw output (unfiltered) as input for interrupt detection. true: Select Analog Comparator filtered output as input for interrupt detection.

static inline bool CMP_GetPreviousInterruptStatus(void)

Get CMP interrupt status before interrupt enable.

Returns

Interrupt status. true: interrupt pending, false: no interrupt pending.

static inline bool CMP_GetInterruptStatus(void)

Get CMP interrupt status after interrupt enable.

Returns

Interrupt status. true: interrupt pending, false: no interrupt pending.

static inline void CMP_FilterSampleConfig(*cmp_filtercfg_samplemode_t* filterSampleMode,
cmp_filtercfg_clkdiv_t filterClockDivider)

CMP Filter Sample Config.

This function allows the users to configure the sampling mode and clock divider of the CMP Filter.

Parameters

- filterSampleMode – CMP Select filter sample mode
- filterClockDivider – CMP Set filter clock divider

FSL_CMP_DRIVER_VERSION

Driver version 2.2.1.

enum _cmp_input_mux

CMP input mux for positive and negative sides.

Values:

enumerator kCMP_InputVREF

Cmp input from VREF.

enumerator kCMP_Input1

Cmp input source 1.

enumerator kCMP_Input2

Cmp input source 2.

enumerator kCMP_Input3

Cmp input source 3.

enumerator kCMP_Input4

Cmp input source 4.

enumerator kCMP_Input5

Cmp input source 5.

enum _cmp_interrupt_type

CMP interrupt type.

Values:

enumerator kCMP_EdgeDisable

Disable edge interrupt.

enumerator kCMP_EdgeRising

Interrupt on falling edge.

enumerator kCMP_EdgeFalling

Interrupt on rising edge.

enumerator kCMP_EdgeRisingFalling

Interrupt on both rising and falling edges.

enumerator kCMP_LevelDisable

Disable level interrupt.

enumerator kCMP_LevelHigh

Interrupt on high level.

enumerator kCMP_LevelLow

Interrupt on low level.

enum _cmp_vref_source

CMP Voltage Reference source.

Values:

enumerator KCMP_VREFSourceVDDA

Select VDDA as VREF.

enumerator KCMP_VREFSourceInternalVREF

Select internal VREF as VREF.

enum _cmp_filtercgf_samplemode

CMP Filter sample mode.

Values:

enumerator kCMP_FilterSampleMode0

Bypass mode. Filtering is disabled.

enumerator kCMP_FilterSampleMode1

Filter 1 clock period.

enumerator kCMP_FilterSampleMode2
Filter 2 clock period.

enumerator kCMP_FilterSampleMode3
Filter 3 clock period.

enum _cmp_filtercgf_clkdiv
CMP Filter clock divider.

Values:

enumerator kCMP_FilterClockDivide1
Filter clock period duration equals 1 analog comparator clock period.

enumerator kCMP_FilterClockDivide2
Filter clock period duration equals 2 analog comparator clock period.

enumerator kCMP_FilterClockDivide4
Filter clock period duration equals 4 analog comparator clock period.

enumerator kCMP_FilterClockDivide8
Filter clock period duration equals 8 analog comparator clock period.

enumerator kCMP_FilterClockDivide16
Filter clock period duration equals 16 analog comparator clock period.

enumerator kCMP_FilterClockDivide32
Filter clock period duration equals 32 analog comparator clock period.

enumerator kCMP_FilterClockDivide64
Filter clock period duration equals 64 analog comparator clock period.

typedef enum _cmp_vref_source cmp_vref_source_t
CMP Voltage Reference source.

typedef struct _cmp_vref_config cmp_vref_config_t

typedef enum _cmp_filtercgf_samplemode cmp_filtercgf_samplemode_t
CMP Filter sample mode.

typedef enum _cmp_filtercgf_clkdiv cmp_filtercgf_clkdiv_t
CMP Filter clock divider.

typedef struct _cmp_config cmp_config_t
CMP configuration structure.

struct _cmp_vref_config
#include <fsl_cmp.h>

Public Members

cmp_vref_source_t vrefSource
Reference voltage source.

uint8_t vrefValue
Reference voltage step. Available range is 0-31. Per step equals to VREFINPUT/31.

struct _cmp_config
#include <fsl_cmp.h> CMP configuration structure.

Public Members

`bool enableHysteresis`
Enable hysteresis.

`bool enableLowPower`
Enable low power mode.

2.7 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.

2.8 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

2.9 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 desscriptor 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:

- a. 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 superseded 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:

- a. 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)
```

- 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[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_int* *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_NUMBER_OF_CHANNELSn(x)

DMA channel numbers.

FSL_FEATURE_DMA_MAX_CHANNELS

FSL_FEATURE_DMA_ALL_CHANNELS

FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE

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 decriptor 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 decriptor 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 algin 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 IAP: In Application Programming Driver

enum _flash_driver_version_constants

Flash driver version for ROM.

Values:

enumerator kFLASH_DriverVersionName

Flash driver version name.

enumerator kFLASH_DriverVersionMajor

Major flash driver version.

enumerator kFLASH_DriverVersionMinor

Minor flash driver version.

enumerator kFLASH_DriverVersionBugfix

Bugfix for flash driver version.

MAKE_VERSION(major, minor, bugfix)

Constructs the version number for drivers.

FSL_FLASH_DRIVER_VERSION

Flash driver version for SDK.

Version 2.1.5.

enum _flash_status

Flash driver status codes.

Values:

enumerator kStatus_FLASH_Success

API is executed successfully

enumerator kStatus_FLASH_InvalidArgument

Invalid argument

enumerator kStatus_FLASH_SizeError

Error size

- enumerator kStatus_FLASH_AlignmentError
Parameter is not aligned with the specified baseline
- enumerator kStatus_FLASH_AddressError
Address is out of range
- enumerator kStatus_FLASH_AccessError
Invalid instruction codes and out-of bound addresses
- enumerator kStatus_FLASH_ProtectionViolation
The program/erase operation is requested to execute on protected areas
- enumerator kStatus_FLASH_CommandFailure
Run-time error during command execution.
- enumerator kStatus_FLASH_UnknownProperty
Unknown property.
- enumerator kStatus_FLASH_EraseKeyError
API erase key is invalid.
- enumerator kStatus_FLASH_RegionExecuteOnly
The current region is execute-only.
- enumerator kStatus_FLASH_ExecuteInRamFunctionNotReady
Execute-in-RAM function is not available.
- enumerator kStatus_FLASH_CommandNotSupported
Flash API is not supported.
- enumerator kStatus_FLASH_ReadOnlyProperty
The flash property is read-only.
- enumerator kStatus_FLASH_InvalidPropertyValue
The flash property value is out of range.
- enumerator kStatus_FLASH_InvalidSpeculationOption
The option of flash prefetch speculation is invalid.
- enumerator kStatus_FLASH_EccError
A correctable or uncorrectable error during command execution.
- enumerator kStatus_FLASH_CompareError
Destination and source memory contents do not match.
- enumerator kStatus_FLASH_RegulationLoss
A loss of regulation during read.
- enumerator kStatus_FLASH_InvalidWaitStateCycles
The wait state cycle set to r/w mode is invalid.
- enumerator kStatus_FLASH_OutOfDateCfpaPage
CFPA page version is out of date.
- enumerator kStatus_FLASH_BlankIfrPageData
Blank page cannot be read.
- enumerator kStatus_FLASH_EncryptedRegionsEraseNotDoneAtOnce
Encrypted flash subregions are not erased at once.
- enumerator kStatus_FLASH_ProgramVerificationNotAllowed
Program verification is not allowed when the encryption is enabled.

enumerator kStatus_FLASH_HashCheckError

Hash check of page data is failed.

enumerator kStatus_FLASH_SealedFfrRegion

The FFR region is sealed.

enumerator kStatus_FLASH_FfrRegionWriteBroken

The FFR Spec region is not allowed to be written discontinuously.

enumerator kStatus_FLASH_NmpaAccessNotAllowed

The NMPA region is not allowed to be read/written/erased.

enumerator kStatus_FLASH_CmpaCfgDirectEraseNotAllowed

The CMPA Cfg region is not allowed to be erased directly.

enumerator kStatus_FLASH_FfrBankIsLocked

The FFR bank region is locked.

enumerator kStatus_FLASH_EraseFrequencyError

Core frequency is over 100MHZ.

enumerator kStatus_FLASH_ProgramFrequencyError

Core frequency is over 100MHZ.

kStatusGroupGeneric

Flash driver status group.

kStatusGroupFlashDriver

MAKE_STATUS(group, code)

Constructs a status code value from a group and a code number.

enum _flash_driver_api_keys

Enumeration for Flash driver API keys.

Note: The resulting value is built with a byte order such that the string being readable in expected order when viewed in a hex editor, if the value is treated as a 32-bit little endian value.

Values:

enumerator kFLASH_ApiEraseKey

Key value used to validate all flash erase APIs.

FOUR_CHAR_CODE(a, b, c, d)

Constructs the four character code for the Flash driver API key.

status_t FLASH_Init(*flash_config_t* *config)

Initializes the global flash properties structure members.

This function checks and initializes the Flash module for the other Flash APIs.

Parameters

- config – Pointer to the storage for the driver runtime state.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.

- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.
- `kStatus_FLASH_EccError` – A correctable or uncorrectable error during command execution.

`status_t FLASH_Erase(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, uint32_t key)`

Erases the flash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length.

Parameters

- `config` – The pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be erased. The start address need to be 512bytes-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words) to be erased. Must be 512bytes-aligned.
- `key` – The value used to validate all flash erase APIs.

Return values

- `kStatus_FLASH_Success` – API was executed successfully; the appropriate number of flash sectors based on the desired start address and length were erased successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – The parameter is not aligned with the specified baseline.
- `kStatus_FLASH_AddressError` – The address is out of range.
- `kStatus_FLASH_EraseKeyError` – The API erase key is invalid.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.
- `kStatus_FLASH_EccError` – A correctable or uncorrectable error during command execution.

`status_t FLASH_Program(flash_config_t *config, uint32_t start, const uint8_t *src, uint32_t lengthInBytes)`

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be programmed. Must be 512bytes-aligned.
- `src` – A pointer to the source buffer of data that is to be programmed into the flash.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be programmed. Must be 512bytes-aligned.

Return values

- `kStatus_FLASH_Success` – API was executed successfully; the desired data were programmed successfully into flash based on desired start address and length.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with the specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.
- `kStatus_FLASH_EccError` – A correctable or uncorrectable error during command execution.

`status_t FLASH_VerifyErase(flash_config_t *config, uint32_t start, uint32_t lengthInBytes)`

Verifies an erasure of the desired flash area at a specified margin level.

This function checks the appropriate number of flash sectors based on the desired start address and length to check whether the flash is erased to the specified read margin level.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. The start address need to be 512bytes-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. Must be 512bytes-aligned.

Return values

- `kStatus_FLASH_Success` – API was executed successfully; the specified FLASH region has been erased.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.
- `kStatus_FLASH_EccError` – A correctable or uncorrectable error during command execution.

`status_t FLASH_VerifyProgram(flash_config_t *config, uint32_t start, uint32_t lengthInBytes, const uint8_t *expectedData, uint32_t *failedAddress, uint32_t *failedData)`

Verifies programming of the desired flash area at a specified margin level.

This function verifies the data programed in the flash memory using the Flash Program Check Command and compares it to the expected data for a given flash area as determined by the start address and length.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `start` – The start address of the desired flash memory to be verified. need be 512bytes-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words), to be verified. need be 512bytes-aligned.
- `expectedData` – A pointer to the expected data that is to be verified against.
- `failedAddress` – A pointer to the returned failing address.
- `failedData` – A pointer to the returned failing data. Some derivatives do not include failed data as part of the FCCOBx registers. In this case, zeros are returned upon failure.

Return values

- `kStatus_FLASH_Success` – API was executed successfully; the desired data have been successfully programed into specified FLASH region.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FLASH_AlignmentError` – Parameter is not aligned with specified baseline.
- `kStatus_FLASH_AddressError` – Address is out of range.
- `kStatus_FLASH_AccessError` – Invalid instruction codes and out-of bounds addresses.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandFailure` – Run-time error during the command execution.
- `kStatus_FLASH_CommandNotSupported` – Flash API is not supported.
- `kStatus_FLASH_EccError` – A correctable or uncorrectable error during command execution.

status_t FLASH_GetProperty(*flash_config_t* *config, *flash_property_tag_t* whichProperty, *uint32_t* *value)

Returns the desired flash property.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `whichProperty` – The desired property from the list of properties in enum `flash_property_tag_t`
- `value` – A pointer to the value returned for the desired flash property.

Return values

- `kStatus_FLASH_Success` – API was executed successfully; the flash property was stored to `value`.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.

- kStatus_FLASH_UnknownProperty – An unknown property tag.

void BOOTLOADER_UserEntry(void *arg)

Run the Bootloader API to force into the ISP mode base on the user arg.

Parameters

- arg – Indicates API prototype fields definition. Refer to the above user_app_boot_invoke_option_t structure

FSL_FEATURE_FLASH_IP_IS_C040HD_ATFC

Flash IP Type.

FSL_FEATURE_FLASH_IP_IS_C040HD_FC

enum _flash_property_tag

Enumeration for various flash properties.

Values:

enumerator kFLASH_PropertyPflashSectorSize

Pflash sector size property.

enumerator kFLASH_PropertyPflashTotalSize

Pflash total size property.

enumerator kFLASH_PropertyPflashBlockSize

Pflash block size property.

enumerator kFLASH_PropertyPflashBlockCount

Pflash block count property.

enumerator kFLASH_PropertyPflashBlockBaseAddr

Pflash block base address property.

enumerator kFLASH_PropertyPflashPageSize

Pflash page size property.

enumerator kFLASH_PropertyPflashSystemFreq

System Frequency System Frequency.

enumerator kFLASH_PropertyFfrSectorSize

FFR sector size property.

enumerator kFLASH_PropertyFfrTotalSize

FFR total size property.

enumerator kFLASH_PropertyFfrBlockBaseAddr

FFR block base address property.

enumerator kFLASH_PropertyFfrPageSize

FFR page size property.

enum _flash_max_erase_page_value

Enumeration for flash max pages to erase.

Values:

enumerator kFLASH_MaxPagesToErase

The max value in pages to erase.

enum _flash_alignment_property

Enumeration for flash alignment property.

Values:

enumerator kFLASH_AlignementUnitVerifyErase

The alignment unit in bytes used for verify erase operation.

enumerator kFLASH_AlignementUnitProgram

The alignment unit in bytes used for program operation.

enumerator kFLASH_AlignementUnitSingleWordRead

The alignment unit in bytes used for verify program operation. The alignment unit in bytes used for SingleWordRead command.

enum _flash_read_ecc_option

Enumeration for flash read ecc option.

Values:

enumerator kFLASH_ReadWithEccOn

enumerator kFLASH_ReadWithEccOff

ECC is on

enum _flash_freq_tag

Values:

enumerator kSysToFlashFreq_lowInMHz

enumerator kSysToFlashFreq_defaultInMHz

enum _flash_read_margin_option

Enumeration for flash read margin option.

Values:

enumerator kFLASH_ReadMarginNormal

Normal read

enumerator kFLASH_ReadMarginVsProgram

Margin vs. program

enumerator kFLASH_ReadMarginVsErase

Margin vs. erase

enumerator kFLASH_ReadMarginIllegalBitCombination

Illegal bit combination

enum _flash_read_dmacc_option

Enumeration for flash read dmacc option.

Values:

enumerator kFLASH_ReadDmaccDisabled

Memory word

enumerator kFLASH_ReadDmaccEnabled

DMACC word

enum _flash_ramp_control_option

Enumeration for flash ramp control option.

Values:

enumerator kFLASH_RampControlDivisionFactorReserved

Reserved

enumerator kFLASH_RampControlDivisionFactor256

clk48mhz / 256 = 187.5KHz

enumerator kFLASH_RampControlDivisionFactor128
clk48mhz / 128 = 375KHz

enumerator kFLASH_RampControlDivisionFactor64
clk48mhz / 64 = 750KHz

typedef enum *_flash_property_tag* flash_property_tag_t
Enumeration for various flash properties.

typedef struct *_flash_ecc_log* flash_ecc_log_t
Flash ECC log info.

typedef struct *_flash_mode_config* flash_mode_config_t
Flash controller paramter config.

typedef struct *_flash_ffr_config* flash_ffr_config_t
Flash controller paramter config.

typedef struct *_flash_config* flash_config_t
Flash driver state information.

An instance of this structure is allocated by the user of the flash driver and passed into each of the driver APIs.

status_t FLASH_Read(*flash_config_t* *config, uint32_t start, uint8_t *dest, uint32_t lengthInBytes)

Reads flash at locations passed in through parameters.

This function read the flash memory from a given flash area as determined by the start address and the length.

Parameters

- config – A pointer to the storage for the driver runtime state.
- start – The start address of the desired flash memory to be read.
- dest – A pointer to the dest buffer of data that is to be read from the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be read.

Return values

- kStatus_FLASH_Success – API was executed successfully.
- kStatus_FLASH_InvalidArgument – An invalid argument is provided.
- kStatus_FLASH_AlignmentError – Parameter is not aligned with the specified baseline.
- kStatus_FLASH_AddressError – Address is out of range.
- kStatus_FLASH_AccessError – Invalid instruction codes and out-of bounds addresses.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandFailure – Run-time error during the command execution.
- kStatus_FLASH_CommandNotSupported – Flash API is not supported.
- kStatus_FLASH_EccError – A correctable or uncorrectable error during command execution.

```
struct _flash_ecc_log
    #include <fsl_iap.h> Flash ECC log info.
struct _flash_mode_config
    #include <fsl_iap.h> Flash controller paramter config.
struct _flash_ffr_config
    #include <fsl_iap.h> Flash controller paramter config.
struct _flash_config
    #include <fsl_iap.h> Flash driver state information.
    An instance of this structure is allocated by the user of the flash driver and passed into each
    of the driver APIs.
```

Public Members

```
uint32_t PFlashBlockBase
    A base address of the first PFlash block
uint32_t PFlashTotalSize
    The size of the combined PFlash block.
uint32_t PFlashBlockCount
    A number of PFlash blocks.
uint32_t PFlashPageSize
    The size in bytes of a page of PFlash.
uint32_t PFlashSectorSize
    The size in bytes of a sector of PFlash.
struct user_app_boot_invoke_option_t
    #include <fsl_iap.h>
struct readSingleWord
struct setWriteMode
struct setReadMode
union option
```

Public Members

```
struct user_app_boot_invoke_option_t B
uint32_t U
struct B
```

2.11 IAP_FFR Driver

```
status_t FFR_Init(flash_config_t *config)
    Initializes the global FFR properties structure members.
```

Parameters

- config – A pointer to the storage for the driver runtime state.

Return values

- `kStatus_FLASH_Success` – API was executed successfully.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.

status_t FFR_Lock_All(*flash_config_t* *config)

Enable firewall for all flash banks.

CFPA, CMPA, and NMPA flash areas region will be locked, After this function executed; Unless the board is reset again.

Parameters

- `config` – A pointer to the storage for the driver runtime state.

Return values

- `kStatus_FLASH_Success` – An invalid argument is provided.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.

status_t FFR_InfieldPageWrite(*flash_config_t* *config, *uint8_t* *page_data, *uint32_t* valid_len)

APIs to access CFPA pages.

This routine will erase CFPA and program the CFPA page with passed data.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `page_data` – A pointer to the source buffer of data that is to be programmed into the CFPA.
- `valid_len` – The length, given in bytes, to be programmed.

Return values

- `kStatus_FLASH_Success` – The desire page-data were programed successfully into CFPA.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FLASH_FfrBankIsLocked` – The CFPA was locked.
- `kStatus_FLASH_OutOfDateCfpaPage` – It is not newest CFPA page.

status_t FFR_GetCustomerInfieldData(*flash_config_t* *config, *uint8_t* *pData, *uint32_t* offset, *uint32_t* len)

APIs to access CFPA pages.

Generic read function, used by customer to read data stored in 'Customer In-field Page'.

Parameters

- `config` – A pointer to the storage for the driver runtime state.
- `pData` – A pointer to the dest buffer of data that is to be read from 'Customer In-field Page'.
- `offset` – An offset from the 'Customer In-field Page' start address.
- `len` – The length, given in bytes, to be read.

Return values

- `kStatus_FLASH_Success` – Get data from 'Customer In-field Page'.
- `kStatus_FLASH_InvalidArgument` – An invalid argument is provided.
- `kStatus_FTFx_AddressError` – Address is out of range.

- `kStatus_FLASH_CommandFailure` – access error.

`status_t` FFR_CustFactoryPageWrite(*flash_config_t* *config, *uint8_t* *page_data, bool seal_part)

APIs to access CMPA pages.

This routine will erase “customer factory page” and program the page with passed data. If ‘seal_part’ parameter is TRUE then the routine will compute SHA256 hash of the page contents and then programs the pages. 1.During development customer code uses this API with ‘seal_part’ set to FALSE. 2.During manufacturing this parameter should be set to TRUE to seal the part from further modifications 3.This routine checks if the page is sealed or not. A page is said to be sealed if the SHA256 value in the page has non-zero value. On boot ROM locks the firewall for the region if hash is programmed anyways. So, write/erase commands will fail eventually.

Parameters

- config – A pointer to the storage for the driver runtime state.
- page_data – A pointer to the source buffer of data that is to be programmed into the “customer factory page”.
- seal_part – Set false for During development customer code.

Return values

- `kStatus_FLASH_Success` – The desire page-data were programed successfully into CMPA.
- `kStatus_FLASH_InvalidArgument` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FLASH_CommandFailure` – access error.

`status_t` FFR_GetCustomerData(*flash_config_t* *config, *uint8_t* *pData, *uint32_t* offset, *uint32_t* len)

APIs to access CMPA page.

Read data stored in ‘Customer Factory CFG Page’.

Parameters

- config – A pointer to the storage for the driver runtime state.
- pData – A pointer to the dest buffer of data that is to be read from the Customer Factory CFG Page.
- offset – Address offset relative to the CMPA area.
- len – The length, given in bytes to be read.

Return values

- `kStatus_FLASH_Success` – Get data from ‘Customer Factory CFG Page’.
- `kStatus_FLASH_InvalidArgument` – Parameter is not aligned with the specified baseline.
- `kStatus_FTFx_AddressError` – Address is out of range.
- `kStatus_FLASH_CommandFailure` – access error.

`status_t` FFR_GetUUID(*flash_config_t* *config, *uint8_t* *uuid)

APIs to access CMPA page.

1.SW should use this API routine to get the UUID of the chip. 2.Calling routine should pass a pointer to buffer which can hold 128-bit value.

status_t FFR_KeystoreWrite(*flash_config_t* *config, *ffr_key_store_t* *pKeyStore)

This routine writes the 3 pages allocated for Key store data,.

1.Used during manufacturing. Should write pages when ‘customer factory page’ is not in sealed state. 2.Optional routines to set individual data members (activation code, key codes etc) to construct the key store structure in RAM before committing it to IFR/FFR.

Parameters

- config – A pointer to the storage for the driver runtime state.
- pKeyStore – A Pointer to the 3 pages allocated for Key store data. that will be written to ‘customer factory page’.

Return values

- kStatus_FLASH_Success – The key were programed successfully into FFR.
- kStatus_FLASH_InvalidArgument – Parameter is not aligned with the specified baseline.
- kStatus_FTFx_AddressError – Address is out of range.
- kStatus_FLASH_CommandFailure – access error.

status_t FFR_KeystoreGetAC(*flash_config_t* *config, *uint8_t* *pActivationCode)

Get/Read Key store code routines.

- a. Calling code should pass buffer pointer which can hold activation code 1192 bytes.
- b. Check if flash aperture is small or regular and read the data appropriately.

status_t FFR_KeystoreGetKC(*flash_config_t* *config, *uint8_t* *pKeyCode, *ffr_key_type_t* keyIndex)

Get/Read Key store code routines.

- a. Calling code should pass buffer pointer which can hold key code 52 bytes.
- b. Check if flash aperture is small or regular and read the data appropriately.
- c. keyIndex specifies which key code is read.

FSL_FLASH_IFR_DRIVER_VERSION

Flash IFR driver version for SDK.

Version 2.1.0.

enum _flash_ffr_page_offset

flash ffr page offset.

Values:

enumerator kFfrPageOffset_CFPA

Customer In-Field programmed area

enumerator kFfrPageOffset_CFPA_Scratch

CFPA Scratch page

enumerator kFfrPageOffset_CFPA_Cfg

CFPA Configuration area (Ping page)

enumerator kFfrPageOffset_CFPA_CfgPong

Same as CFPA page (Pong page)

enumerator kFfrPageOffset_CMPA

Customer Manufacturing programmed area

enumerator kFfrPageOffset__CMPA__Cfg
CMPA Configuration area (Part of CMPA)

enumerator kFfrPageOffset__CMPA__Key
Key Store area (Part of CMPA)

enumerator kFfrPageOffset__NMPA
NXP Manufacturing programmed area

enumerator kFfrPageOffset__NMPA__Romcp
ROM patch area (Part of NMPA)

enumerator kFfrPageOffset__NMPA__Repair
Repair area (Part of NMPA)

enumerator kFfrPageOffset__NMPA__Cfg
NMPA configuration area (Part of NMPA)

enumerator kFfrPageOffset__NMPA__End
Reserved (Part of NMPA)

enum __flash_ffr_page_num
flash ffr page number.
Values:

enumerator kFfrPageNum__CFPA
Customer In-Field programmed area

enumerator kFfrPageNum__CMPA
Customer Manufacturing programmed area

enumerator kFfrPageNum__NMPA
NXP Manufacturing programmed area

enumerator kFfrPageNum__CMPA__Cfg

enumerator kFfrPageNum__CMPA__Key

enumerator kFfrPageNum__NMPA__Romcp

enumerator kFfrPageNum__SpecArea

enumerator kFfrPageNum__Total

enum __flash_ffr_block_size
Values:

enumerator kFfrBlockSize__Key

enumerator kFfrBlockSize__ActivationCode

enum __cfpa_cfg_cmpa_prog_process
Values:

enumerator kFfrCmpaProgProcess__Pre

enumerator kFfrCmpaProgProcess__Post

enum __ffr_key_type
Values:

enumerator kFFR__KeyTypeSbkek


```
enumerator kFFR__KeyTypeUser
enumerator kFFR__KeyTypeUds
enumerator kFFR__KeyTypePrinceRegion0
enumerator kFFR__KeyTypePrinceRegion1
enumerator kFFR__KeyTypePrinceRegion2
enum __ffr_bank_type
    Values:
    enumerator kFFR__BankTypeBank0__NMPA
    enumerator kFFR__BankTypeBank1__CMPA
    enumerator kFFR__BankTypeBank2__CFPA
typedef enum __cfpa_cfg_cmpa_prog_process cmpa__prog__process__t
typedef struct __cfpa_cfg_iv_code cfpa__cfg_iv__code__t
typedef struct __cfpa_cfg_info cfpa__cfg_info__t
typedef struct __cmpa_cfg_info cmpa__cfg_info__t
typedef struct __cmpa_key_store_header cmpa__key__store__header__t
typedef struct __nmpa_cfg_info nmpa__cfg_info__t
typedef struct __ffr_key_store ffr__key__store__t
typedef enum __ffr_key_type ffr__key__type__t
typedef enum __ffr_bank_type ffr__bank__type__t
ALIGN_DOWN(x, a)
    Alignment(down) utility.
ALIGN_UP(x, a)
    Alignment(up) utility.
FLASH_FFR_MAX_PAGE_SIZE
FLASH_FFR_HASH_DIGEST_SIZE
FLASH_FFR_IV_CODE_SIZE
FFR_BOOTCFG_BOOTSPEED_MASK
FFR_BOOTCFG_BOOTSPEED_SHIFT
FFR_BOOTCFG_BOOTSPEED_48MHZ
FFR_BOOTCFG_BOOTSPEED_96MHZ
FFR_USPID_VENDORID_MASK
FFR_USPID_VENDORID_SHIFT
FFR_USPID_PRODUCTID_MASK
FFR_USPID_PRODUCTID_SHIFT
```

```
FFR_SYSTEM_SPEED_CODE_MASK
FFR_SYSTEM_SPEED_CODE_SHIFT
FFR_SYSTEM_SPEED_CODE_FRO12MHZ_12MHZ
FFR_SYSTEM_SPEED_CODE_FROHF96MHZ_24MHZ
FFR_SYSTEM_SPEED_CODE_FROHF96MHZ_48MHZ
FFR_SYSTEM_SPEED_CODE_FROHF96MHZ_96MHZ
FFR_PERIPHERALCFG_PERI_MASK
FFR_PERIPHERALCFG_PERI_SHIFT
FFR_PERIPHERALCFG_COREEN_MASK
FFR_PERIPHERALCFG_COREEN_SHIFT
struct _cfpa_cfg_iv_code
    #include <fsl_iap_ffr.h>
struct _cfpa_cfg_info
    #include <fsl_iap_ffr.h>
```

Public Members

```
uint32_t header
    [0x000-0x003]
uint32_t version
    [0x004-0x007]
uint32_t secureFwVersion
    [0x008-0x00b]
uint32_t nsFwVersion
    [0x00c-0x00f]
uint32_t imageKeyRevoke
    [0x010-0x013]
uint8_t reserved0[4]
    [0x014-0x017]
uint32_t rotkhRevoke
    [0x018-0x01b]
uint32_t vendorUsage
    [0x01c-0x01f]
uint32_t dcfgNsPin
    [0x020-0x013]
uint32_t dcfgNsDflt
    [0x024-0x017]
uint32_t enableFaMode
    [0x028-0x02b]
uint8_t reserved1[4]
    [0x02c-0x02f]
```

```
    cfpa_cfg_iv_code_t ivCodePrinceRegion[3]
        [0x030-0x0d7]
    uint8_t reserved2[264]
        [0x0d8-0x1df]
    uint8_t sha256[32]
        [0x1e0-0x1ff]
struct __cmpa_cfg_info
    #include <fsl_iap_ffr.h>
```

Public Members

```
uint32_t bootCfg
    [0x000-0x003]
uint32_t spiFlashCfg
    [0x004-0x007]
struct __cmpa_cfg_info usbId
    [0x008-0x00b]
uint32_t sdioCfg
    [0x00c-0x00f]
uint32_t dcfgPin
    [0x010-0x013]
uint32_t dcfgDflt
    [0x014-0x017]
uint32_t dapVendorUsage
    [0x018-0x01b]
uint32_t secureBootCfg
    [0x01c-0x01f]
uint32_t princeBaseAddr
    [0x020-0x023]
uint32_t princeSr[3]
    [0x024-0x02f]
uint8_t reserved0[32]
    [0x030-0x04f]
uint32_t rotkh[8]
    [0x050-0x06f]
uint8_t reserved1[368]
    [0x070-0x1df]
uint8_t sha256[32]
    [0x1e0-0x1ff]
struct __cmpa_key_store_header
    #include <fsl_iap_ffr.h>
struct __nmpa_cfg_info
    #include <fsl_iap_ffr.h>
```

Public Members

uint16_t fro32kCfg
[0x000-0x001]

uint8_t reserved0[6]
[0x002-0x007]

uint8_t sysCfg
[0x008-0x008]

uint8_t reserved1[7]
[0x009-0x00f]

struct *_nmpa_cfg_info* GpoInitData[3]
[0x010-0x03f]

uint32_t GpoDataChecksum[4]
[0x040-0x04f]

uint32_t finalTestBatchId[4]
[0x050-0x05f]

uint32_t deviceType
[0x060-0x063]

uint32_t finalTestProgVersion
[0x064-0x067]

uint32_t finalTestDate
[0x068-0x06b]

uint32_t finalTestTime
[0x06c-0x06f]

uint32_t uuid[4]
[0x070-0x07f]

uint8_t reserved2[32]
[0x080-0x09f]

uint32_t peripheralCfg
[0x0a0-0x0a3]

uint32_t ramSizeCfg
[0x0a4-0x0a7]

uint32_t flashSizeCfg
[0x0a8-0x0ab]

uint8_t reserved3[36]
[0x0ac-0x0cf]

uint8_t fro1mCfg
[0x0d0-0x0d0]

uint8_t reserved4[15]
[0x0d1-0x0df]

uint32_t dcde[4]
[0x0e0-0x0ef]

```
uint32_t bod
    [0x0f0-0x0f3]
uint8_t reserved5[12]
    [0x0f4-0x0ff]
uint8_t calcHashReserved[192]
    [0x100-0x1bf]
uint8_t sha256[32]
    [0x1c0-0x1df]
uint32_t ecdBackup[4]
    [0x1e0-0x1ef]
uint32_t pageChecksum[4]
    [0x1f0-0x1ff]
struct __ffr_key_store
    #include <fsl_iap_ffr.h>
struct usbId
struct GpoInitData
```

2.12 FLEXCOMM: FLEXCOMM Driver

2.13 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.

2.14 GINT: Group GPIO Input Interrupt Driver

FSL_GINT_DRIVER_VERSION

Driver version.

enum _gint_comb

GINT combine inputs type.

Values:

enumerator kGINT_CombineOr

A grouped interrupt is generated when any one of the enabled inputs is active

enumerator kGINT_CombineAnd

A grouped interrupt is generated when all enabled inputs are active

enum _gint_trig

GINT trigger type.

Values:

enumerator kGINT_TrigEdge

Edge triggered based on polarity

enumerator kGINT_TrigLevel

Level triggered based on polarity

enum _gint_port

Values:

enumerator kGINT_Port0

typedef enum _gint_comb gint_comb_t

GINT combine inputs type.

typedef enum _gint_trig gint_trig_t

GINT trigger type.

typedef enum _gint_port gint_port_t

typedef void (*gint_cb_t)(void)

GINT Callback function.

void GINT_Init(GINT_Type *base)

Initialize GINT peripheral.

This function initializes the GINT peripheral and enables the clock.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

void GINT_SetCtrl(GINT_Type *base, *gint_comb_t* comb, *gint_trig_t* trig, *gint_cb_t* callback)

Setup GINT peripheral control parameters.

This function sets the control parameters of GINT peripheral.

Parameters

- base – Base address of the GINT peripheral.
- comb – Controls if the enabled inputs are logically ORed or ANDed for interrupt generation.
- trig – Controls if the enabled inputs are level or edge sensitive based on polarity.
- callback – This function is called when configured group interrupt is generated.

Return values

None. –

void GINT_GetCtrl(GINT_Type *base, *gint_comb_t* *comb, *gint_trig_t* *trig, *gint_cb_t* *callback)

Get GINT peripheral control parameters.

This function returns the control parameters of GINT peripheral.

Parameters

- base – Base address of the GINT peripheral.
- comb – Pointer to store combine input value.
- trig – Pointer to store trigger value.
- callback – Pointer to store callback function.

Return values

None. –

void GINT_ConfigPins(GINT_Type *base, *gint_port_t* port, uint32_t polarityMask, uint32_t enableMask)

Configure GINT peripheral pins.

This function enables and controls the polarity of enabled pin(s) of a given port.

Parameters

- base – Base address of the GINT peripheral.
- port – Port number.
- polarityMask – Each bit position selects the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
- enableMask – Each bit position selects if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

```
void GINT_GetConfigPins(GINT_Type *base, gint_port_t port, uint32_t *polarityMask, uint32_t *enableMask)
```

Get GINT peripheral pin configuration.

This function returns the pin configuration of a given port.

Parameters

- base – Base address of the GINT peripheral.
- port – Port number.
- polarityMask – Pointer to store the polarity mask Each bit position indicates the polarity of the corresponding enabled pin. 0 = The pin is active LOW. 1 = The pin is active HIGH.
- enableMask – Pointer to store the enable mask. Each bit position indicates if the corresponding pin is enabled or not. 0 = The pin is disabled. 1 = The pin is enabled.

Return values

None. –

```
void GINT_EnableCallback(GINT_Type *base)
```

Enable callback.

This function enables the interrupt for the selected GINT 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 GINT peripheral.

Return values

None. –

```
void GINT_DisableCallback(GINT_Type *base)
```

Disable callback.

This function disables the interrupt for the selected GINT 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. –

```
static inline void GINT_ClrStatus(GINT_Type *base)
```

Clear GINT status.

This function clears the GINT status bit.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

```
static inline uint32_t GINT_GetStatus(GINT_Type *base)
```

Get GINT status.

This function returns the GINT status.

Parameters

- base – Base address of the GINT peripheral.

Return values

status – = 0 No group interrupt request. = 1 Group interrupt request active.

void GINT_Deinit(GINT_Type *base)

Deinitialize GINT peripheral.

This function disables the GINT clock.

Parameters

- base – Base address of the GINT peripheral.

Return values

None. –

2.15 Hashcrypt: The Cryptographic Accelerator

2.16 Hashcrypt Background HASH

void HASHCRYPT_SHA_SetCallback(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, *hashcrypt_callback_t* callback, void *userData)

Initializes the HASHCRYPT handle for background hashing.

This function initializes the hash context for background hashing (Non-blocking) APIs. This is less typical interface to hash function, but can be used for parallel processing, when main CPU has something else to do. Example is digital signature RSASSA-PKCS1-V1_5-VERIFY((n,e),M,S) algorithm, where background hashing of M can be started, then CPU can compute $S^e \bmod n$ (in parallel with background hashing) and once the digest becomes available, CPU can proceed to comparison of EM with EM'.

Parameters

- base – HASHCRYPT peripheral base address.
- ctx – **[out]** Hash context.
- callback – Callback function.
- userData – User data (to be passed as an argument to callback function, once callback is invoked from isr).

status_t HASHCRYPT_SHA_UpdateNonBlocking(HASHCRYPT_Type *base, *hashcrypt_hash_ctx_t* *ctx, const uint8_t *input, size_t inputSize)

Create running hash on given data.

Configures the HASHCRYPT to compute new running hash as AHB master and returns immediately. HASHCRYPT AHB Master mode supports only aligned input address and can be called only once per continuous block of data. Every call to this function must be preceded with HASHCRYPT_SHA_Init() and finished with HASHCRYPT_SHA_Finish(). Once callback function is invoked by HASHCRYPT isr, it should set a flag for the main application to finalize the hashing (padding) and to read out the final digest by calling HASHCRYPT_SHA_Finish().

Parameters

- base – HASHCRYPT peripheral base address
- ctx – Specifies callback. Last incomplete 512-bit block of the input is copied into clear buffer for padding.
- input – 32-bit word aligned pointer to Input data.
- inputSize – Size of input data in bytes (must be word aligned)

Returns

Status of the hash update operation.

2.17 Hashcrypt common functions

FSL_HASHCRYPT_DRIVER_VERSION

HASHCRYPT driver version. Version 2.2.16.

Current version: 2.2.16

Change log:

- Version 2.0.0
 - Initial version
- Version 2.0.1
 - Support loading AES key from unaligned address
- Version 2.0.2
 - Support loading AES key from unaligned address for different compiler and core variants
- Version 2.0.3
 - Remove SHA512 and AES ICB algorithm definitions
- Version 2.0.4
 - Add SHA context switch support
- Version 2.1.0
 - Update the register name and macro to align with new header.
- Version 2.1.1
 - Fix MISRA C-2012.
- Version 2.1.2
 - Support loading AES input data from unaligned address.
- Version 2.1.3
 - Fix MISRA C-2012.
- Version 2.1.4
 - Fix context switch cannot work when switching from AES.
- Version 2.1.5
 - Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() to prevent possible optimization issue.
- Version 2.2.0
 - Add AES-OFB and AES-CFB mixed IP/SW modes.
- Version 2.2.1
 - Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() prevent compiler from reordering memory write when -O2 or higher is used.
- Version 2.2.2
 - Add data synchronization barrier inside hashcrypt_sha_ldm_stm_16_words() to fix optimization issue

- Version 2.2.3
 - Added check for size in `hashcrypt_aes_one_block` to prevent overflowing COUNT field in MEMCTRL register, if its bigger than COUNT field do a multiple runs.
- Version 2.2.4
 - In all `HASHCRYPT_AES_xx` functions have been added setting CTRL_MODE bitfield to 0 after processing data, which decreases power consumption.
- Version 2.2.5
 - Add data synchronization barrier and instruction synchronization barrier inside `hashcrypt_sha_process_message_data()` to fix optimization issue
- Version 2.2.6
 - Add data synchronization barrier inside `HASHCRYPT_SHA_Update()` and `hashcrypt_get_data()` function to fix optimization issue on MDK and ARMGCC release targets
- Version 2.2.7
 - Add data synchronization barrier inside `HASHCRYPT_SHA_Update()` to fix optimization issue on MCUX IDE release target
- Version 2.2.8
 - Unify hashcrypt hashing behavior between aligned and unaligned input data
- Version 2.2.9
 - Add handling of set ERROR bit in the STATUS register
- Version 2.2.10
 - Fix missing error statement in `hashcrypt_save_running_hash()`
- Version 2.2.11
 - Fix incorrect SHA-256 calculation for long messages with reload
- Version 2.2.12
 - Fix hardfault issue on the Keil compiler due to unaligned `memcpy()` input on some optimization levels
- Version 2.2.13
 - Added function `hashcrypt_seed_prng()` which loading random number into PRNG_SEED register before AES operation for SCA protection
- Version 2.2.14
 - Modify function `hashcrypt_get_data()` to prevent issue with unaligned access
- Version 2.2.15
 - Add wait on DIGEST BIT inside `hashcrypt_sha_one_block()` to fix issues with some optimization flags
- Version 2.2.16
 - Add DSB instruction inside `hashcrypt_sha_ldm_stm_16_words()` to fix issues with some optimization flags

`enum __hashcrypt_algo_t`

Algorithm used for Hashcrypt operation.

Values:

enumerator kHASHCRYPT_Sha1
SHA_1

enumerator kHASHCRYPT_Sha256
SHA_256

enumerator kHASHCRYPT_Aes
AES

typedef enum *hashcrypt_algo_t* hashcrypt_algo_t
Algorithm used for Hashcrypt operation.

void HASHCRYPT_Init(HASHCRYPT_Type *base)
Enables clock and disables reset for HASHCRYPT peripheral.
Enable clock and disable reset for HASHCRYPT.

Parameters

- base – HASHCRYPT base address

void HASHCRYPT_Deinit(HASHCRYPT_Type *base)
Disables clock for HASHCRYPT peripheral.
Disable clock and enable reset.

Parameters

- base – HASHCRYPT base address

HASHCRYPT_MODE_SHA1

Algorithm definitions correspond with the values for Mode field in Control register !

HASHCRYPT_MODE_SHA256

HASHCRYPT_MODE_AES

2.18 Hashcrypt AES

enum *_hashcrypt_aes_mode_t*
AES mode.

Values:

enumerator kHASHCRYPT_AesEcb
AES ECB mode

enumerator kHASHCRYPT_AesCbc
AES CBC mode

enumerator kHASHCRYPT_AesCtr
AES CTR mode

enum *_hashcrypt_aes_keysize_t*
Size of AES key.

Values:

enumerator kHASHCRYPT_Aes128
AES 128 bit key

enumerator kHASHCRYPT_Aes192
AES 192 bit key

enumerator kHASHCRYPT_Aes256

AES 256 bit key

enumerator kHASHCRYPT_InvalidKey

AES invalid key

enum _hashcrypt_key

HASHCRYPT key source selection.

Values:

enumerator kHASHCRYPT_UserKey

HASHCRYPT user key

enumerator kHASHCRYPT_SecretKey

HASHCRYPT secret key (dedicated hw bus from PUF)

typedef enum _hashcrypt_aes_mode_t hashcrypt_aes_mode_t

AES mode.

typedef enum _hashcrypt_aes_keysize_t hashcrypt_aes_keysize_t

Size of AES key.

typedef enum _hashcrypt_key hashcrypt_key_t

HASHCRYPT key source selection.

typedef struct _hashcrypt_handle hashcrypt_handle_t

struct _hashcrypt_handle __attribute__((aligned))

status_t HASHCRYPT_AES_SetKey(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle,
const uint8_t *key, size_t keySize)

Set AES key to *hashcrypt_handle_t* struct and optionally to HASHCRYPT.

Sets the AES key for encryption/decryption with the *hashcrypt_handle_t* structure. The *hashcrypt_handle_t* input argument specifies key source.

Parameters

- base – HASHCRYPT peripheral base address.
- handle – Handle used for the request.
- key – 0-mod-4 aligned pointer to AES key.
- keySize – AES key size in bytes. Shall equal 16, 24 or 32.

Returns

status from set key operation

status_t HASHCRYPT_AES_EncryptEcb(HASHCRYPT_Type *base, *hashcrypt_handle_t* *handle,
const uint8_t *plaintext, uint8_t *ciphertext, size_t
size)

Encrypts AES on one or multiple 128-bit block(s).

Encrypts AES. The source plaintext and destination ciphertext can overlap in system memory.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text

- size – Size of input and output data in bytes. Must be multiple of 16 bytes.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_DecryptEcb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                  const uint8_t *ciphertext, uint8_t *plaintext, size_t  
                                  size)
```

Decrypts AES on one or multiple 128-bit block(s).

Decrypts AES. The source ciphertext and destination plaintext can overlap in system memory.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- ciphertext – Input plain text to encrypt
- plaintext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.

Returns

Status from decrypt operation

```
status_t HASHCRYPT_AES_EncryptCbc(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                  const uint8_t *plaintext, uint8_t *ciphertext, size_t  
                                  size, const uint8_t iv[16])
```

Encrypts AES using CBC block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- plaintext – Input plain text to encrypt
- ciphertext – **[out]** Output cipher text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_DecryptCbc(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                  const uint8_t *ciphertext, uint8_t *plaintext, size_t  
                                  size, const uint8_t iv[16])
```

Decrypts AES using CBC block mode.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- ciphertext – Input cipher text to decrypt
- plaintext – **[out]** Output plain text
- size – Size of input and output data in bytes. Must be multiple of 16 bytes.
- iv – Input initial vector to combine with the first input block.

Returns

Status from decrypt operation

```
status_t HASHCRYPT_AES_CryptCtr(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                const uint8_t *input, uint8_t *output, size_t size, uint8_t  
                                counter[16U], uint8_t counterlast[16U], size_t *szLeft)
```

Encrypts or decrypts AES using CTR block mode.

Encrypts or decrypts AES using CTR block mode. AES CTR mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- input – Input data for CTR block mode
- output – **[out]** Output data for CTR block mode
- size – Size of input and output data in bytes
- counter – **[inout]** Input counter (updates on return)
- counterlast – **[out]** Output cipher of last counter, for chained CTR calls (statefull encryption). NULL can be passed if chained calls are not used.
- szLeft – **[out]** Output number of bytes in left unused in counterlast block. NULL can be passed if chained calls are not used.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_CryptOfb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                const uint8_t *input, uint8_t *output, size_t size, const  
                                uint8_t iv[16U])
```

Encrypts or decrypts AES using OFB block mode.

Encrypts or decrypts AES using OFB block mode. AES OFB mode uses only forward AES cipher and same algorithm for encryption and decryption. The only difference between encryption and decryption is that, for encryption, the input argument is plain text and the output argument is cipher text. For decryption, the input argument is cipher text and the output argument is plain text.

Parameters

- base – HASHCRYPT peripheral base address
- handle – Handle used for this request.
- input – Input data for OFB block mode
- output – **[out]** Output data for OFB block mode
- size – Size of input and output data in bytes
- iv – Input initial vector to combine with the first input block.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_EncryptCfb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,  
                                   const uint8_t *plaintext, uint8_t *ciphertext, size_t size,  
                                   const uint8_t iv[16])
```

Encrypts AES using CFB block mode.

Parameters

- `base` – HASHCRYPT peripheral base address
- `handle` – Handle used for this request.
- `plaintext` – Input plain text to encrypt
- `ciphertext` – **[out]** Output cipher text
- `size` – Size of input and output data in bytes. Must be multiple of 16 bytes.
- `iv` – Input initial vector to combine with the first input block.

Returns

Status from encrypt operation

```
status_t HASHCRYPT_AES_DecryptCfb(HASHCRYPT_Type *base, hashcrypt_handle_t *handle,
                                   const uint8_t *ciphertext, uint8_t *plaintext, size_t size,
                                   const uint8_t iv[16])
```

Decrypts AES using CFB block mode.

Parameters

- `base` – HASHCRYPT peripheral base address
- `handle` – Handle used for this request.
- `ciphertext` – Input cipher text to decrypt
- `plaintext` – **[out]** Output plaintext text
- `size` – Size of input and output data in bytes. Must be multiple of 16 bytes.
- `iv` – Input initial vector to combine with the first input block.

Returns

Status from encrypt operation

```
HASHCRYPT_AES_BLOCK_SIZE
```

AES block size in bytes

```
AES_ENCRYPT
```

```
AES_DECRYPT
```

```
struct _hashcrypt_handle
```

#include <fsl_hashcrypt.h> Specify HASHCRYPT's key resource.

Public Members

```
uint32_t keyWord[8]
```

Copy of user key (set by HASHCRYPT_AES_SetKey()).

```
hashcrypt_key_t keyType
```

For operations with key (such as AES encryption/decryption), specify key type.

2.19 Hashcrypt HASH

```
typedef struct _hashcrypt_hash_ctx_t hashcrypt_hash_ctx_t
```

Storage type used to save hash context.

```
typedef void (*hashcrypt_callback_t)(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx,
status_t status, void *userData)
```

HASHCRYPT background hash callback function.


```
status_t HASHCRYPT_SHA(HASHCRYPT_Type *base, hashcrypt_algo_t algo, const uint8_t
                        *input, size_t inputSize, uint8_t *output, size_t *outputSize)
```

Create HASH on given data.

Perform the full SHA in one function call. The function is blocking.

Parameters

- base – HASHCRYPT peripheral base address
- algo – Underlying algorithm to use for hash computation.
- input – Input data
- inputSize – Size of input data in bytes
- output – **[out]** Output hash data
- outputSize – **[out]** Output parameter storing the size of the output hash in bytes

Returns

Status of the one call hash operation.

```
status_t HASHCRYPT_SHA_Init(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx,
                           hashcrypt_algo_t algo)
```

Initialize HASH context.

This function initializes the HASH.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[out]** Output hash context
- algo – Underlying algorithm to use for hash computation.

Returns

Status of initialization

```
status_t HASHCRYPT_SHA_Update(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, const
                             uint8_t *input, size_t inputSize)
```

Add data to current HASH.

Add data to current HASH. This can be called repeatedly with an arbitrary amount of data to be hashed. The function blocks. If it returns `kStatus_Success`, the running hash has been updated (HASHCRYPT has processed the input data), so the memory at `input` pointer can be released back to system. The HASHCRYPT context buffer is updated with the running hash and with all necessary information to support possible context switch.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[inout]** HASH context
- input – Input data
- inputSize – Size of input data in bytes

Returns

Status of the hash update operation

```
status_t HASHCRYPT_SHA_Finish(HASHCRYPT_Type *base, hashcrypt_hash_ctx_t *ctx, uint8_t
                             *output, size_t *outputSize)
```

Finalize hashing.

Outputs the final hash (computed by `HASHCRYPT_HASH_Update()`) and erases the context.

Parameters

- base – HASHCRYPT peripheral base address
- ctx – **[inout]** Input hash context
- output – **[out]** Output hash data
- outputSize – **[inout]** Optional parameter (can be passed as NULL). On function entry, it specifies the size of output[] buffer. On function return, it stores the number of updated output bytes.

Returns

Status of the hash finish operation

HASHCRYPT_HASH_CTX_SIZE

HASHCRYPT HASH Context size.

struct _hashcrypt_hash_ctx_t

#include <fsl_hashcrypt.h> Storage type used to save hash context.

Public Members

uint32_t x[22]

storage

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 – Sucessully 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.

`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_SaveDeselected
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 superseded 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 call-back.

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 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
```

2.27 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;
```

(continues on next page)

(continued from previous page)

```
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 = 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

2.28 INPUTMUX: Input Multiplexing Driver

`enum __inputmux_connection_t`
INPUTMUX connections type.

Values:

`enumerator kINPUTMUX__SctGpi0ToSct0`
SCT0 INMUX.

`enumerator kINPUTMUX__SctGpi1ToSct0`

`enumerator kINPUTMUX__SctGpi2ToSct0`

`enumerator kINPUTMUX__SctGpi3ToSct0`

`enumerator kINPUTMUX__SctGpi4ToSct0`

`enumerator kINPUTMUX__SctGpi5ToSct0`

`enumerator kINPUTMUX__SctGpi6ToSct0`

`enumerator kINPUTMUX__SctGpi7ToSct0`

`enumerator kINPUTMUX__Ctimer0M0ToSct0`

`enumerator kINPUTMUX__Ctimer1M0ToSct0`

enumerator kINPUTMUX__Ctimer2M0ToSct0

enumerator kINPUTMUX__Ctimer3M0ToSct0

enumerator kINPUTMUX__Ctimer4M0ToSct0

enumerator kINPUTMUX__AdcIrqToSct0

enumerator kINPUTMUX__GpioIntBmatchToSct0

enumerator kINPUTMUX__Usb0FrameToggleToSct0

enumerator kINPUTMUX__Usb1FrameToggleToSct0

enumerator kINPUTMUX__CompOutToSct0

enumerator kINPUTMUX__I2sSharedSck0ToSct0

enumerator kINPUTMUX__I2sSharedSck1ToSct0

enumerator kINPUTMUX__I2sSharedWs0ToSct0

enumerator kINPUTMUX__I2sSharedWs1ToSct0

enumerator kINPUTMUX__ArmTxeVToSct0

enumerator kINPUTMUX__DebugHaltedToSct0

TIMER0 CAPTSEL.

enumerator kINPUTMUX__CtimerInp0ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp1ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp2ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp3ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp4ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp5ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp6ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp7ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp8ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp9ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp10ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp11ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp12ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp13ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp14ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp15ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp16ToTimer0Captsel

enumerator kINPUTMUX__CtimerInp17ToTimer0Captsel

```
enumerator kINPUTMUX__CtimerInp18ToTimer0Captsel
enumerator kINPUTMUX__CtimerInp19ToTimer0Captsel
enumerator kINPUTMUX__Usb0FrameToggleToTimer0Captsel
enumerator kINPUTMUX__Usb1FrameToggleToTimer0Captsel
enumerator kINPUTMUX__CompOutToTimer0Captsel
enumerator kINPUTMUX__I2sSharedWs0ToTimer0Captsel
enumerator kINPUTMUX__I2sSharedWs1ToTimer0Captsel
    TIMER1 CAPTSEL.
enumerator kINPUTMUX__CtimerInp0ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp1ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp2ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp3ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp4ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp5ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp6ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp7ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp8ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp9ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp10ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp11ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp12ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp13ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp14ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp15ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp16ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp17ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp18ToTimer1Captsel
enumerator kINPUTMUX__CtimerInp19ToTimer1Captsel
enumerator kINPUTMUX__Usb0FrameToggleToTimer1Captsel
enumerator kINPUTMUX__Usb1FrameToggleToTimer1Captsel
enumerator kINPUTMUX__CompOutToTimer1Captsel
enumerator kINPUTMUX__I2sSharedWs0ToTimer1Captsel
enumerator kINPUTMUX__I2sSharedWs1ToTimer1Captsel
    TIMER2 CAPTSEL.
```


enumerator kINPUTMUX_CtimerInp0ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp1ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp2ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp3ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp4ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp5ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp6ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp7ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp8ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp9ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp10ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp11ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp12ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp13ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp14ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp15ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp16ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp17ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp18ToTimer2Cptsel
enumerator kINPUTMUX_CtimerInp19ToTimer2Cptsel
enumerator kINPUTMUX_Usb0FrameToggleToTimer2Cptsel
enumerator kINPUTMUX_Usb1FrameToggleToTimer2Cptsel
enumerator kINPUTMUX_CompOutToTimer2Cptsel
enumerator kINPUTMUX_I2sSharedWs0ToTimer2Cptsel
enumerator kINPUTMUX_I2sSharedWs1ToTimer2Cptsel
Pin interrupt select.
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

DMA0 Input trigger.

enumerator kINPUTMUX_PinInt0ToDma0
enumerator kINPUTMUX_PinInt1ToDma0
enumerator kINPUTMUX_PinInt2ToDma0
enumerator kINPUTMUX_PinInt3ToDma0
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__Ctimer4M0ToDma0

enumerator kINPUTMUX__Ctimer4M1ToDma0

enumerator kINPUTMUX__CompOutToDma0

enumerator kINPUTMUX__Otrig0ToDma0

enumerator kINPUTMUX__Otrig1ToDma0

enumerator kINPUTMUX__Otrig2ToDma0

enumerator kINPUTMUX__Otrig3ToDma0

enumerator kINPUTMUX__Sct0DmaReq0ToDma0

enumerator kINPUTMUX__Sct0DmaReq1ToDma0

enumerator kINPUTMUX__HashDmaRxToDma0

DMA0 output trigger.

enumerator kINPUTMUX__Dma0Hash0TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0HsLspiRxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0HsLspiTxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm0RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm0TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm1RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm1TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm3RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm3TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm2RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm2TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm4RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm4TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm5RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm5TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm6RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm6TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm7RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Flexcomm7TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Adc0Ch0TrigoutToTriginChannels

enumerator kINPUTMUX__Dma0Adc0Ch1TrigoutToTriginChannels

Selection for frequency measurement reference clock.

enumerator kINPUTMUX__ExternOscToFreqmeasRef

enumerator kINPUTMUX__Fro12MhzToFreqmeasRef

enumerator kINPUTMUX__Fro96MhzToFreqmeasRef

enumerator kINPUTMUX__WdtOscToFreqmeasRef

enumerator kINPUTMUX__32KhzOscToFreqmeasRef

enumerator kINPUTMUX__MainClkToFreqmeasRef

enumerator kINPUTMUX__FreqmeGpioClk__aRef

enumerator kINPUTMUX__FreqmeGpioClk__bRef

Selection for frequency measurement target clock.

enumerator kINPUTMUX__ExternOscToFreqmeasTarget

enumerator kINPUTMUX__Fro12MhzToFreqmeasTarget

enumerator kINPUTMUX__Fro96MhzToFreqmeasTarget

enumerator kINPUTMUX__WdtOscToFreqmeasTarget

enumerator kINPUTMUX__32KhzOscToFreqmeasTarget

enumerator kINPUTMUX__MainClkToFreqmeasTarget

enumerator kINPUTMUX__FreqmeGpioClk__aTarget

enumerator kINPUTMUX__FreqmeGpioClk__bTarget

TIMER3 CAPTSEL.

enumerator kINPUTMUX__CtimerInp0ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp1ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp2ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp3ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp4ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp5ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp6ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp7ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp8ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp9ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp10ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp11ToTimer3Captsel

enumerator kINPUTMUX__CtimerInp12ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp13ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp14ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp15ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp16ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp17ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp18ToTimer3Captsel
enumerator kINPUTMUX__CtimerInp19ToTimer3Captsel
enumerator kINPUTMUX__Usb0FrameToggleToTimer3Captsel
enumerator kINPUTMUX__Usb1FrameToggleToTimer3Captsel
enumerator kINPUTMUX__CompOutToTimer3Captsel
enumerator kINPUTMUX__I2sSharedWs0ToTimer3Captsel
enumerator kINPUTMUX__I2sSharedWs1ToTimer3Captsel
Timer4 CAPTSEL.
enumerator kINPUTMUX__CtimerInp0ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp1ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp2ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp3ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp4ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp5ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp6ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp7ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp8ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp9ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp10ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp11ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp12ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp13ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp14ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp15ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp16ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp17ToTimer4Captsel
enumerator kINPUTMUX__CtimerInp18ToTimer4Captsel

enumerator kINPUTMUX__CtimerInp19ToTimer4Captsel
enumerator kINPUTMUX__Usb0FrameToggleToTimer4Captsel
enumerator kINPUTMUX__Usb1FrameToggleToTimer4Captsel
enumerator kINPUTMUX__CompOutToTimer4Captsel
enumerator kINPUTMUX__I2sSharedWs0ToTimer4Captsel
enumerator kINPUTMUX__I2sSharedWs1ToTimer4Captsel
enumerator kINPUTMUX__GpioPort0Pin0ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin1ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin2ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin3ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin4ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin5ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin6ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin7ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin8ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin9ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin10ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin11ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin12ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin13ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin14ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin15ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin16ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin17ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin18ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin19ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin20ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin21ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin22ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin23ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin24ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin25ToPintSecsel
enumerator kINPUTMUX__GpioPort0Pin26ToPintSecsel

enumerator kINPUTMUX__GpioPort0Pin27ToPintSecsel

enumerator kINPUTMUX__GpioPort0Pin28ToPintSecsel

enumerator kINPUTMUX__GpioPort0Pin29ToPintSecsel

enumerator kINPUTMUX__GpioPort0Pin30ToPintSecsel

enumerator kINPUTMUX__GpioPort0Pin31ToPintSecsel

DMA1 Input trigger.

enumerator kINPUTMUX__PinInt0ToDma1

enumerator kINPUTMUX__PinInt1ToDma1

enumerator kINPUTMUX__PinInt2ToDma1

enumerator kINPUTMUX__PinInt3ToDma1

enumerator kINPUTMUX__Ctimer0M0ToDma1

enumerator kINPUTMUX__Ctimer0M1ToDma1

enumerator kINPUTMUX__Ctimer2M0ToDma1

enumerator kINPUTMUX__Ctimer4M0ToDma1

enumerator kINPUTMUX__Otrig0ToDma1

enumerator kINPUTMUX__Otrig1ToDma1

enumerator kINPUTMUX__Otrig2ToDma1

enumerator kINPUTMUX__Otrig3ToDma1

enumerator kINPUTMUX__Sct0DmaReq0ToDma1

enumerator kINPUTMUX__Sct0DmaReq1ToDma1

enumerator kINPUTMUX__HashDmaRxToDma1

DMA1 output trigger.

enumerator kINPUTMUX__Dma1Hash0TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1HsLspiRxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1HsLspiTxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1Flexcomm0RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1Flexcomm0TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1Flexcomm1RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1Flexcomm1TxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1Flexcomm3RxTrigoutToTriginChannels

enumerator kINPUTMUX__Dma1Flexcomm3TxTrigoutToTriginChannels

enum __inputmux_signal_t

INPUTMUX signal enable/disable type.

Values:

enumerator kINPUTMUX_HashCryptToDmac0Ch0RequestEna
DMA0 REQ signal.

enumerator kINPUTMUX_Flexcomm8RxToDmac0Ch2RequestEna

enumerator kINPUTMUX_Flexcomm8TxToDmac0Ch3RequestEna

enumerator kINPUTMUX_Flexcomm0RxToDmac0Ch4RequestEna

enumerator kINPUTMUX_Flexcomm0TxToDmac0Ch5RequestEna

enumerator kINPUTMUX_Flexcomm1RxToDmac0Ch6RequestEna

enumerator kINPUTMUX_Flexcomm1TxToDmac0Ch7RequestEna

enumerator kINPUTMUX_Flexcomm3RxToDmac0Ch8RequestEna

enumerator kINPUTMUX_Flexcomm3TxToDmac0Ch9RequestEna

enumerator kINPUTMUX_Flexcomm2RxToDmac0Ch10RequestEna

enumerator kINPUTMUX_Flexcomm2TxToDmac0Ch11RequestEna

enumerator kINPUTMUX_Flexcomm4RxToDmac0Ch12RequestEna

enumerator kINPUTMUX_Flexcomm4TxToDmac0Ch13RequestEna

enumerator kINPUTMUX_Flexcomm5RxToDmac0Ch14RequestEna

enumerator kINPUTMUX_Flexcomm5TxToDmac0Ch15RequestEna

enumerator kINPUTMUX_Flexcomm6RxToDmac0Ch16RequestEna

enumerator kINPUTMUX_Flexcomm6TxToDmac0Ch17RequestEna

enumerator kINPUTMUX_Flexcomm7RxToDmac0Ch18RequestEna

enumerator kINPUTMUX_Flexcomm7TxToDmac0Ch19RequestEna

enumerator kINPUTMUX_Adc0FIFO0ToDmac0Ch21RequestEna

enumerator kINPUTMUX_Adc0FIFO1ToDmac0Ch22RequestEna

DMA1 REQ signal.

enumerator kINPUTMUX_HashCryptToDmac1Ch0RequestEna

enumerator kINPUTMUX_Flexcomm8RxToDmac1Ch2RequestEna

enumerator kINPUTMUX_Flexcomm8TxToDmac1Ch3RequestEna

enumerator kINPUTMUX_Flexcomm0RxToDmac1Ch4RequestEna

enumerator kINPUTMUX_Flexcomm0TxToDmac1Ch5RequestEna

enumerator kINPUTMUX_Flexcomm1RxToDmac1Ch6RequestEna

enumerator kINPUTMUX_Flexcomm1TxToDmac1Ch7RequestEna

enumerator kINPUTMUX_Flexcomm3RxToDmac1Ch8RequestEna

enumerator kINPUTMUX_Flexcomm3TxToDmac1Ch9RequestEna

DMA0 input trigger source enable.

enumerator kINPUTMUX_Dmac0InputTriggerPint0Ena

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__Dmac0InputTriggerCtimer4M0Ena
enumerator kINPUTMUX__Dmac0InputTriggerCtimer4M1Ena
enumerator kINPUTMUX__Dmac0InputTriggerCompOutEna
enumerator kINPUTMUX__Dmac0InputTriggerDma0Out0Ena
enumerator kINPUTMUX__Dmac0InputTriggerDma0Out1Ena
enumerator kINPUTMUX__Dmac0InputTriggerDma0Out2Ena
enumerator kINPUTMUX__Dmac0InputTriggerDma0Out3Ena
enumerator kINPUTMUX__Dmac0InputTriggerSctDmac0Ena
enumerator kINPUTMUX__Dmac0InputTriggerSctDmac1Ena
enumerator kINPUTMUX__Dmac0InputTriggerHashOutEna
 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__Dmac1InputTriggerCtimer2M0Ena
enumerator kINPUTMUX__Dmac1InputTriggerCtimer4M0Ena
enumerator kINPUTMUX__Dmac1InputTriggerDma1Out0Ena
enumerator kINPUTMUX__Dmac1InputTriggerDma1Out1Ena
enumerator kINPUTMUX__Dmac1InputTriggerDma1Out2Ena

enumerator kINPUTMUX_Dmac1InputTriggerDma1Out3Ena
enumerator kINPUTMUX_Dmac1InputTriggerSctDmac0Ena
enumerator kINPUTMUX_Dmac1InputTriggerSctDmac1Ena
enumerator kINPUTMUX_Dmac1InputTriggerHashOutEna
typedef enum *_inputmux_connection_t* inputmux_connection_t
INPUTMUX connections type.
typedef enum *_inputmux_signal_t* inputmux_signal_t
INPUTMUX signal enable/disable type.
SCT0_INMUX0
Periphinmux IDs.
TIMER0CAPTSEL0
TIMER1CAPTSEL0
TIMER2CAPTSEL0
PINTSEL_PMUX_ID
PINTSEL0
DMA0_ITRIG_INMUX0
DMA0_OTRIG_INMUX0
FREQMEAS_REF_REG
FREQMEAS_TARGET_REG
TIMER3CAPTSEL0
TIMER4CAPTSEL0
PINTSECSEL0
DMA1_ITRIG_INMUX0
DMA1_OTRIG_INMUX0
DMA0_REQ_ENA_ID
DMA1_REQ_ENA_ID
DMA0_ITRIG_ENA_ID
DMA1_ITRIG_ENA_ID
ENA_SHIFT
PMUX_SHIFT
FSL_INPUTMUX_DRIVER_VERSION
Group interrupt driver version for SDK.

`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. –

2.29 IAP_KBP Driver

ROM API status codes.

Values:

enumerator kStatus_RomApiExecuteCompleted

ROM successfully process the whole sb file/boot image.

enumerator kStatus_RomApiNeedMoreData

ROM needs more data to continue processing the boot image.

enumerator kStatus_RomApiBufferSizeNotEnough

The user buffer is not enough for use by Kboot during execution of the operation.

enumerator kStatus_RomApiInvalidBuffer

The user buffer is not ok for sbloader or authentication.

enum _kb_operation

Details of the operation to be performed by the ROM.

The kRomAuthenticateImage operation requires the entire signed image to be available to the application.

Values:

enumerator kRomAuthenticateImage

Authenticate a signed image.

enumerator kRomLoadImage

Load SB file.

enumerator kRomOperationCount

enum _kb_security_profile

Security constraint flags, Security profile flags.

Values:

enumerator kKbootMinRSA4096

typedef enum _kb_operation kb_operation_t

Details of the operation to be performed by the ROM.

The kRomAuthenticateImage operation requires the entire signed image to be available to the application.

typedef struct _kb_region kb_region_t

Memory region definition.

typedef struct _kb_load_sb kb_load_sb_t

User-provided options passed into kb_init().

The buffer field is a pointer to memory provided by the caller for use by Kboot during execution of the operation. Minimum size is the size of each certificate in the chain plus 432 bytes additional per certificate.

The profile field is a mask that specifies which features are required in the SB file or image being processed. This includes the minimum AES and RSA key sizes. See the _kb_security_profile enum for profile mask constants. The image being loaded or authenticated must match the profile or an error will be returned.

minBuildNumber is an optional field that can be used to prevent version rollback. The API will check the build number of the image, and if it is less than minBuildNumber will fail with an error.

maxImageLength is used to verify the offsetToCertificateBlockHeaderInBytes value at the beginning of a signed image. It should be set to the length of the SB file. If verifying an image in flash, it can be set to the internal flash size or a large number like 0x10000000.

userRHK can optionally be used by the user to override the RHK in IFR. If userRHK is not NULL, it points to a 32-byte array containing the SHA-256 of the root certificate's RSA public key.

The regions field points to an array of memory regions that the SB file being loaded is allowed to access. If regions is NULL, then all memory is accessible by the SB file. This feature is required to prevent a malicious image from erasing good code or RAM contents while it is being loaded, only for us to find that the image is inauthentic when we hit the end of the section.

overrideSBBootSectionID lets the caller override the default section of the SB file that is processed during a kKbootLoadSB operation. By default, the section specified in the first-BootableSectionID field of the SB header is loaded. If overrideSBBootSectionID is non-zero, then the section with the given ID will be loaded instead.

The userSBKEK field lets a user provide their own AES-256 key for unwrapping keys in an SB file during the kKbootLoadSB operation. userSBKEK should point to a 32-byte AES-256 key. If userSBKEK is NULL then the IFR SBKEK will be used. After kb_init() returns, the caller should zero out the data pointed to by userSBKEK, as the API will have installed the key in the CAU3.

```
typedef struct _kb_authenticate kb_authenticate_t
```

```
typedef struct _kb_options kb_options_t
```

```
typedef struct _memory_region_interface memory_region_interface_t
```

Interface to memory operations for one region of memory.

```
typedef struct _memory_map_entry memory_map_entry_t
```

Structure of a memory map entry.

```
typedef struct _kb_opaque_session_ref kb_session_ref_t
```

```
status_t kb_init(kb_session_ref_t **session, const kb_options_t *options)
```

Initialize ROM API for a given operation.

Initials the ROM API based on the options provided by the application in the second argument. Every call to rom_init() should be paired with a call to rom_deinit().

Return values

- kStatus_Success – API was executed successfully.
- kStatus_InvalidArgument – An invalid argument is provided.
- kStatus_RomApiBufferSizeNotEnough – The user buffer is not enough for use by Kboot during execution of the operation.
- kStatus_RomApiInvalidBuffer – The user buffer is not ok for sbloader or authentication.
- kStatus_SKBOOT_Fail – Return the failed status of secure boot.
- kStatus_SKBOOT_KeyStoreMarkerInvalid – The key code for the particular PRINCE region is not present in the keystore
- kStatus_SKBOOT_Success – Return the successful status of secure boot.

```
status_t kb_deinit(kb_session_ref_t *session)
```

Cleans up the ROM API context.

After this call, the context parameter can be reused for another operation by calling rom_init() again.

Return values

kStatus_Success – API was executed successfully

status_t kb_execute(*kb_session_ref_t* *session, const uint8_t *data, uint32_t dataLength)

Perform the operation configured during init.

This application must call this API repeatedly, passing in sequential chunks of data from the boot image (SB file) that is to be processed. The ROM will perform the selected operation on this data and return. The application may call this function with as much or as little data as it wishes, which can be used to select the granularity of time given to the application in between executing the operation.

Parameters

- session – Current ROM context pointer.
- data – Buffer of boot image data provided to the ROM by the application.
- dataLength – Length in bytes of the data in the buffer provided to the ROM.

Return values

- kStatus_Success – ROM successfully process the part of sb file/boot image.
- kStatus_RomApiExecuteCompleted – ROM successfully process the whole sb file/boot image.
- kStatus_Fail – An error occurred while executing the operation.
- kStatus_RomApiNeedMoreData – No error occurred, but the ROM needs more data to continue processing the boot image.
- kStatus_RomApiBufferSizeNotEnough – user buffer is not enough for use by Kboot during execution of the operation.

kStatusGroup_RomApi

ROM API status group number.

struct _kb_region

#include <fsl_iap_kbp.h> Memory region definition.

struct _kb_load_sb

#include <fsl_iap_kbp.h> User-provided options passed into kb_init().

The buffer field is a pointer to memory provided by the caller for use by Kboot during execution of the operation. Minimum size is the size of each certificate in the chain plus 432 bytes additional per certificate.

The profile field is a mask that specifies which features are required in the SB file or image being processed. This includes the minimum AES and RSA key sizes. See the _kb_security_profile enum for profile mask constants. The image being loaded or authenticated must match the profile or an error will be returned.

minBuildNumber is an optional field that can be used to prevent version rollback. The API will check the build number of the image, and if it is less than minBuildNumber will fail with an error.

maxImageLength is used to verify the offsetToCertificateBlockHeaderInBytes value at the beginning of a signed image. It should be set to the length of the SB file. If verifying an image in flash, it can be set to the internal flash size or a large number like 0x10000000.

userRHK can optionally be used by the user to override the RHK in IFR. If userRHK is not NULL, it points to a 32-byte array containing the SHA-256 of the root certificate's RSA public key.

The regions field points to an array of memory regions that the SB file being loaded is allowed to access. If regions is NULL, then all memory is accessible by the SB file. This feature is required to prevent a malicious image from erasing good code or RAM contents while it

is being loaded, only for us to find that the image is inauthentic when we hit the end of the section.

overrideSBBootSectionID lets the caller override the default section of the SB file that is processed during a kKbootLoadSB operation. By default, the section specified in the first-BootableSectionID field of the SB header is loaded. If overrideSBBootSectionID is non-zero, then the section with the given ID will be loaded instead.

The userSBKEK field lets a user provide their own AES-256 key for unwrapping keys in an SB file during the kKbootLoadSB operation. userSBKEK should point to a 32-byte AES-256 key. If userSBKEK is NULL then the IFR SBKEK will be used. After kb_init() returns, the caller should zero out the data pointed to by userSBKEK, as the API will have installed the key in the CAU3.

```
struct __kb_authenticate
    #include <fsl_iap_kbp.h>

struct __kb_options
    #include <fsl_iap_kbp.h>
```

Public Members

uint32_t version
Should be set to kKbootApiVersion.

uint8_t *buffer
Caller-provided buffer used by Kboot.

```
struct __memory_region_interface
    #include <fsl_iap_kbp.h> Interface to memory operations for one region of memory.
```

```
struct __memory_map_entry
    #include <fsl_iap_kbp.h> Structure of a memory map entry.
```

```
struct __kb_opaque_session_ref
    #include <fsl_iap_kbp.h>
```

```
union __unnamed11__
```

Public Members

kb_authenticate_t authenticate

kb_load_sb_t loadSB
Settings for kKbootAuthenticate operation.

2.30 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 UART 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_MIPIDSI
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_AUDMIX
Group number for AUDMIX 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__SERIALMANAGER
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.

2.31 LPADC: 12-bit SAR Analog-to-Digital Converter Driver

`enum __lpadc_status_flags`

Define hardware flags of the module.

Values:

enumerator `kLPADC_ResultFIFO0OverflowFlag`

Indicates that more data has been written to the Result FIFO 0 than it can hold.

enumerator `kLPADC_ResultFIFO0ReadyFlag`

Indicates when the number of valid datawords in the result FIFO 0 is greater than the setting watermark level.

enumerator `kLPADC_TriggerExceptionFlag`

Indicates that a trigger exception event has occurred.

enumerator `kLPADC_TriggerCompletionFlag`

Indicates that a trigger completion event has occurred.

enumerator kLPADC_CalibrationReadyFlag

Indicates that the calibration process is done.

enumerator kLPADC_ActiveFlag

Indicates that the ADC is in active state.

enumerator kLPADC_ResultFIFOOverflowFlag

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowFlag as instead.

enumerator kLPADC_ResultFIFOReadyFlag

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0ReadyFlag as instead.

enum __lpadc_interrupt_enable

Define interrupt switchers of the module.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_ResultFIFO0OverflowInterruptEnable

Configures ADC to generate overflow interrupt requests when FOF0 flag is asserted.

enumerator kLPADC_FIFO0WatermarkInterruptEnable

Configures ADC to generate watermark interrupt requests when RDY0 flag is asserted.

enumerator kLPADC_ResultFIFOOverflowInterruptEnable

To compilitable with old version, do not recommend using this, please use kLPADC_ResultFIFO0OverflowInterruptEnable as instead.

enumerator kLPADC_FIFOWatermarkInterruptEnable

To compilitable with old version, do not recommend using this, please use kLPADC_FIFO0WatermarkInterruptEnable as instead.

enumerator kLPADC_TriggerExceptionInterruptEnable

Configures ADC to generate trigger exception interrupt.

enumerator kLPADC_Trigger0CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 0 completion.

enumerator kLPADC_Trigger1CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 1 completion.

enumerator kLPADC_Trigger2CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 2 completion.

enumerator kLPADC_Trigger3CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 3 completion.

enumerator kLPADC_Trigger4CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 4 completion.

enumerator kLPADC_Trigger5CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 5 completion.

enumerator kLPADC_Trigger6CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 6 completion.

enumerator kLPADC_Trigger7CompletionInterruptEnable

Configures ADC to generate interrupt when trigger 7 completion.

enumerator kLPADC_Trigger8CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 8 completion.

enumerator kLPADC_Trigger9CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 9 completion.

enumerator kLPADC_Trigger10CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 10 completion.

enumerator kLPADC_Trigger11CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 11 completion.

enumerator kLPADC_Trigger12CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 12 completion.

enumerator kLPADC_Trigger13CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 13 completion.

enumerator kLPADC_Trigger14CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 14 completion.

enumerator kLPADC_Trigger15CompletionInterruptEnable
 Configures ADC to generate interrupt when trigger 15 completion.

enum _lpadc_trigger_status_flags

The enumerator of lpadc trigger status flags, including interrupted flags and completed flags.

Note: LPADC of different chips supports different number of trigger sources, please check the Reference Manual for details.

Values:

enumerator kLPADC_Trigger0InterruptedFlag
 Trigger 0 is interrupted by a high priority exception.

enumerator kLPADC_Trigger1InterruptedFlag
 Trigger 1 is interrupted by a high priority exception.

enumerator kLPADC_Trigger2InterruptedFlag
 Trigger 2 is interrupted by a high priority exception.

enumerator kLPADC_Trigger3InterruptedFlag
 Trigger 3 is interrupted by a high priority exception.

enumerator kLPADC_Trigger4InterruptedFlag
 Trigger 4 is interrupted by a high priority exception.

enumerator kLPADC_Trigger5InterruptedFlag
 Trigger 5 is interrupted by a high priority exception.

enumerator kLPADC_Trigger6InterruptedFlag
 Trigger 6 is interrupted by a high priority exception.

enumerator kLPADC_Trigger7InterruptedFlag
 Trigger 7 is interrupted by a high priority exception.

enumerator kLPADC_Trigger8InterruptedFlag
 Trigger 8 is interrupted by a high priority exception.

enumerator kLPADC_Trigger9InterruptedFlag
 Trigger 9 is interrupted by a high priority exception.

enumerator kLPADC_Trigger10InterruptedFlag

Trigger 10 is interrupted by a high priority exception.

enumerator kLPADC_Trigger11InterruptedFlag

Trigger 11 is interrupted by a high priority exception.

enumerator kLPADC_Trigger12InterruptedFlag

Trigger 12 is interrupted by a high priority exception.

enumerator kLPADC_Trigger13InterruptedFlag

Trigger 13 is interrupted by a high priority exception.

enumerator kLPADC_Trigger14InterruptedFlag

Trigger 14 is interrupted by a high priority exception.

enumerator kLPADC_Trigger15InterruptedFlag

Trigger 15 is interrupted by a high priority exception.

enumerator kLPADC_Trigger0CompletedFlag

Trigger 0 is completed and trigger 0 has enabled completion interrupts.

enumerator kLPADC_Trigger1CompletedFlag

Trigger 1 is completed and trigger 1 has enabled completion interrupts.

enumerator kLPADC_Trigger2CompletedFlag

Trigger 2 is completed and trigger 2 has enabled completion interrupts.

enumerator kLPADC_Trigger3CompletedFlag

Trigger 3 is completed and trigger 3 has enabled completion interrupts.

enumerator kLPADC_Trigger4CompletedFlag

Trigger 4 is completed and trigger 4 has enabled completion interrupts.

enumerator kLPADC_Trigger5CompletedFlag

Trigger 5 is completed and trigger 5 has enabled completion interrupts.

enumerator kLPADC_Trigger6CompletedFlag

Trigger 6 is completed and trigger 6 has enabled completion interrupts.

enumerator kLPADC_Trigger7CompletedFlag

Trigger 7 is completed and trigger 7 has enabled completion interrupts.

enumerator kLPADC_Trigger8CompletedFlag

Trigger 8 is completed and trigger 8 has enabled completion interrupts.

enumerator kLPADC_Trigger9CompletedFlag

Trigger 9 is completed and trigger 9 has enabled completion interrupts.

enumerator kLPADC_Trigger10CompletedFlag

Trigger 10 is completed and trigger 10 has enabled completion interrupts.

enumerator kLPADC_Trigger11CompletedFlag

Trigger 11 is completed and trigger 11 has enabled completion interrupts.

enumerator kLPADC_Trigger12CompletedFlag

Trigger 12 is completed and trigger 12 has enabled completion interrupts.

enumerator kLPADC_Trigger13CompletedFlag

Trigger 13 is completed and trigger 13 has enabled completion interrupts.

enumerator kLPADC_Trigger14CompletedFlag

Trigger 14 is completed and trigger 14 has enabled completion interrupts.

enumerator kLPADC_Trigger15CompletedFlag

Trigger 15 is completed and trigger 15 has enabled completion interrupts.

enum __lpadc_sample_scale_mode

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

Values:

enumerator kLPADC_SamplePartScale

Use divided input voltage signal. (For scale select, please refer to the reference manual).

enumerator kLPADC_SampleFullScale

Full scale (Factor of 1).

enum __lpadc_sample_channel_mode

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

Values:

enumerator kLPADC_SampleChannelSingleEndSideA

Single-end mode, only A-side channel is converted.

enumerator kLPADC_SampleChannelSingleEndSideB

Single-end mode, only B-side channel is converted.

enumerator kLPADC_SampleChannelDiffBothSideAB

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDiffBothSideBA

Differential mode, the ADC result is (CHnB-CHnA).

enumerator kLPADC_SampleChannelDiffBothSide

Differential mode, the ADC result is (CHnA-CHnB).

enumerator kLPADC_SampleChannelDualSingleEndBothSide

Dual-Single-Ended Mode. Both A side and B side channels are converted independently.

enum __lpadc_hardware_average_mode

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

Values:

enumerator kLPADC_HardwareAverageCount1

Single conversion.

enumerator kLPADC_HardwareAverageCount2
2 conversions averaged.

enumerator kLPADC_HardwareAverageCount4
4 conversions averaged.

enumerator kLPADC_HardwareAverageCount8
8 conversions averaged.

enumerator kLPADC_HardwareAverageCount16
16 conversions averaged.

enumerator kLPADC_HardwareAverageCount32
32 conversions averaged.

enumerator kLPADC_HardwareAverageCount64
64 conversions averaged.

enumerator kLPADC_HardwareAverageCount128
128 conversions averaged.

enum _lpadc_sample_time_mode

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

Values:

enumerator kLPADC_SampleTimeADCK3
3 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK5
5 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK7
7 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK11
11 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK19
19 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK35
35 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK67
69 ADCK cycles total sample time.

enumerator kLPADC_SampleTimeADCK131
131 ADCK cycles total sample time.

enum _lpadc_hardware_compare_mode

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

Values:

enumerator kLPADC_HardwareCompareDisabled
Compare disabled.

enumerator kLPADC_HardwareCompareStoreOnTrue
Compare enabled. Store on true.

enumerator kLPADC_HardwareCompareRepeatUntilTrue
Compare enabled. Repeat channel acquisition until true.

enum _lpadc_conversion_resolution_mode

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to lpadc_sample_channel_mode_t

Values:

enumerator kLPADC_ConversionResolutionStandard
Standard resolution. Single-ended 12-bit conversion, Differential 13-bit conversion with 2's complement output.

enumerator kLPADC_ConversionResolutionHigh
High resolution. Single-ended 16-bit conversion; Differential 16-bit conversion with 2's complement output.

enum _lpadc_conversion_average_mode

Define enumeration of conversion averages mode.

Configure the conversion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

Values:

enumerator kLPADC_ConversionAverage1
Single conversion.

enumerator kLPADC_ConversionAverage2
2 conversions averaged.

enumerator kLPADC_ConversionAverage4
4 conversions averaged.

enumerator kLPADC_ConversionAverage8
8 conversions averaged.

enumerator kLPADC_ConversionAverage16
16 conversions averaged.

enumerator kLPADC_ConversionAverage32
32 conversions averaged.

enumerator kLPADC_ConversionAverage64
64 conversions averaged.

enumerator kLPADC_ConversionAverage128
128 conversions averaged.

enum `_lpadc_reference_voltage_mode`

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

Values:

enumerator `kLPADC_ReferenceVoltageAlt1`

Option 1 setting.

enumerator `kLPADC_ReferenceVoltageAlt2`

Option 2 setting.

enumerator `kLPADC_ReferenceVoltageAlt3`

Option 3 setting.

enum `_lpadc_power_level_mode`

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

Values:

enumerator `kLPADC_PowerLevelAlt1`

Lowest power setting.

enumerator `kLPADC_PowerLevelAlt2`

Next lowest power setting.

enumerator `kLPADC_PowerLevelAlt3`

...

enumerator `kLPADC_PowerLevelAlt4`

Highest power setting.

enum `_lpadc_offset_calibration_mode`

Define enumeration of offset calibration mode.

Values:

enumerator `kLPADC_OffsetCalibration12bitMode`

12 bit offset calibration mode.

enumerator `kLPADC_OffsetCalibration16bitMode`

16 bit offset calibration mode.

enum `_lpadc_trigger_priority_policy`

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: `kLPADC_TriggerPriorityPreemptSubsequently` is not available on some devices, mainly depends on the size of `TPRCTRL` field in `CFG` register.

Values:

enumerator `kLPADC_ConvPreemptImmediatelyNotAutoResumed`

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion is not automatically resumed or restarted.

enumerator `kLPADC_ConvPreemptSoftlyNotAutoResumed`

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion is not resumed or restarted.

enumerator `kLPADC_ConvPreemptImmediatelyAutoRestarted`

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator `kLPADC_ConvPreemptSoftlyAutoRestarted`

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will automatically be restarted.

enumerator `kLPADC_ConvPreemptImmediatelyAutoResumed`

If a higher priority trigger is detected during command processing, the current conversion is aborted and the new command specified by the trigger is started, when higher priority conversion finishes, the preempted conversion will automatically be resumed.

enumerator `kLPADC_ConvPreemptSoftlyAutoResumed`

If a higher priority trigger is received during command processing, the current conversion is completed (including averaging iterations and compare function if enabled) and stored to the result FIFO before the higher priority trigger/command is initiated, when higher priority conversion finishes, the preempted conversion will be automatically be resumed.

enumerator `kLPADC_TriggerPriorityPreemptImmediately`

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator `kLPADC_TriggerPriorityPreemptSoftly`

Legacy support is not recommended as it only ensures compatibility with older versions.

enumerator `kLPADC_TriggerPriorityExceptionDisabled`

High priority trigger exception disabled.

typedef enum `_lpadc_sample_scale_mode` `lpadc_sample_scale_mode_t`

Define enumeration of sample scale mode.

The sample scale mode is used to reduce the selected ADC analog channel input voltage level by a factor. The maximum possible voltage on the ADC channel input should be considered when selecting a scale mode to ensure that the reducing factor always results voltage level at or below the VREFH reference. This reducing capability allows conversion of analog inputs higher than VREFH. A-side and B-side channel inputs are both scaled using the scale mode.

typedef enum `_lpadc_sample_channel_mode` `lpadc_sample_channel_mode_t`

Define enumeration of channel sample mode.

The channel sample mode configures the channel with single-end/differential/dual-single-end, side A/B.

typedef enum `_lpadc_hardware_average_mode` `lpadc_hardware_average_mode_t`

Define enumeration of hardware average selection.

It Selects how many ADC conversions are averaged to create the ADC result. An internal storage buffer is used to capture temporary results while the averaging iterations are executed.

Note: Some enumerator values are not available on some devices, mainly depends on the size of AVGS field in CMDH register.

`typedef enum _lpadc_sample_time_mode lpadc_sample_time_mode_t`

Define enumeration of sample time selection.

The shortest sample time maximizes conversion speed for lower impedance inputs. Extending sample time allows higher impedance inputs to be accurately sampled. Longer sample times can also be used to lower overall power consumption when command looping and sequencing is configured and high conversion rates are not required.

`typedef enum _lpadc_hardware_compare_mode lpadc_hardware_compare_mode_t`

Define enumeration of hardware compare mode.

After an ADC channel input is sampled and converted and any averaging iterations are performed, this mode setting guides operation of the automatic compare function to optionally only store when the compare operation is true. When compare is enabled, the conversion result is compared to the compare values.

`typedef enum _lpadc_conversion_resolution_mode lpadc_conversion_resolution_mode_t`

Define enumeration of conversion resolution mode.

Configure the resolution bit in specific conversion type. For detailed resolution accuracy, see to `lpadc_sample_channel_mode_t`

`typedef enum _lpadc_conversion_average_mode lpadc_conversion_average_mode_t`

Define enumeration of conversion averages mode.

Configure the conversion average number for auto-calibration.

Note: Some enumerator values are not available on some devices, mainly depends on the size of CAL_AVGS field in CTRL register.

`typedef enum _lpadc_reference_voltage_mode lpadc_reference_voltage_source_t`

Define enumeration of reference voltage source.

For detail information, need to check the SoC's specification.

`typedef enum _lpadc_power_level_mode lpadc_power_level_mode_t`

Define enumeration of power configuration.

Configures the ADC for power and performance. In the highest power setting the highest conversion rates will be possible. Refer to the device data sheet for power and performance capabilities for each setting.

`typedef enum _lpadc_offset_calibration_mode lpadc_offset_calibration_mode_t`

Define enumeration of offset calibration mode.

`typedef enum _lpadc_trigger_priority_policy lpadc_trigger_priority_policy_t`

Define enumeration of trigger priority policy.

This selection controls how higher priority triggers are handled.

Note: `kLPADC_TriggerPriorityPreemptSubsequently` is not available on some devices, mainly depends on the size of TPRCTRL field in CFG register.

```
typedef struct lpadc_calibration_value lpadc_calibration_value_t
```

A structure of calibration value.

```
LPADC_CONVERSION_COMPLETE_TIMEOUT
```

Max loops to wait for LPADC conversion complete.

When doing calibration, driver will wait for the completion of conversion. This parameter defines how many loops to check completion before return timeout. If defined as 0, driver will wait forever until completion.

```
LPADC_CALIBRATION_READY_TIMEOUT
```

Max loops to wait for LPADC calibration ready.

Before doing calibration, driver will wait for the calibration ready. This parameter defines how many loops to check the calibration ready. If defined as 0, driver will wait forever until ready.

```
LPADC_GAIN_CAL_READY_TIMEOUT
```

Max loops to wait for LPADC gain calibration GAIN_CAL ready.

Before doing calibration, driver will wait for the gain calibration GAIN_CAL ready. This parameter defines how many loops to check the gain calibration GAIN_CAL ready. If defined as 0, driver will wait forever until ready.

```
ADC_OFSTRIM_OFSTRIM_MAX
```

```
ADC_OFSTRIM_OFSTRIM_SIGN
```

```
LPADC_GET_ACTIVE_COMMAND_STATUS(statusVal)
```

Define the MACRO function to get command status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

```
LPADC_GET_ACTIVE_TRIGGER_STATUE(statusVal)
```

Define the MACRO function to get trigger status from status value.

The statusVal is the return value from LPADC_GetStatusFlags().

```
void LPADC_Init(ADC_Type *base, const lpadc_config_t *config)
```

Initializes the LPADC module.

Parameters

- base – LPADC peripheral base address.
- config – Pointer to configuration structure. See “*lpadc_config_t*”.

```
void LPADC_GetDefaultConfig(lpadc_config_t *config)
```

Gets an available pre-defined settings for initial configuration.

This function initializes the converter configuration structure with an available settings. The default values are:

```
config->enableInDozeMode      = true;
config->enableAnalogPreliminary = false;
config->powerUpDelay           = 0x80;
config->referenceVoltageSource  = kLPADC_ReferenceVoltageAlt1;
config->powerLevelMode         = kLPADC_PowerLevelAlt1;
config->triggerPriorityPolicy    = kLPADC_TriggerPriorityPreemptImmediately;
config->enableConvPause        = false;
config->convPauseDelay          = 0U;
config->FIFOWatermark           = 0U;
```

Parameters

- config – Pointer to configuration structure.

`void LPADC_Deinit(ADC_Type *base)`
De-initializes the LPADC module.

Parameters

- `base` – LPADC peripheral base address.

`static inline void LPADC_Enable(ADC_Type *base, bool enable)`
Switch on/off the LPADC module.

Parameters

- `base` – LPADC peripheral base address.
- `enable` – switcher to the module.

`static inline void LPADC_DoResetFIFO(ADC_Type *base)`
Do reset the conversion FIFO.

Parameters

- `base` – LPADC peripheral base address.

`static inline void LPADC_DoResetConfig(ADC_Type *base)`
Do reset the module's configuration.

Reset all ADC internal logic and registers, except the Control Register (ADCx_CTRL).

Parameters

- `base` – LPADC peripheral base address.

`static inline uint32_t LPADC_GetStatusFlags(ADC_Type *base)`
Get status flags.

Parameters

- `base` – LPADC peripheral base address.

Returns

status flags' mask. See to `_lpadc_status_flags`.

`static inline void LPADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)`
Clear status flags.

Only the flags can be cleared by writing ADCx_STATUS register would be cleared by this API.

Parameters

- `base` – LPADC peripheral base address.
- `mask` – Mask value for flags to be cleared. See to `_lpadc_status_flags`.

`static inline uint32_t LPADC_GetTriggerStatusFlags(ADC_Type *base)`
Get trigger status flags to indicate which trigger sequences have been completed or interrupted by a high priority trigger exception.

Parameters

- `base` – LPADC peripheral base address.

Returns

The OR'ed value of `_lpadc_trigger_status_flags`.

`static inline void LPADC_ClearTriggerStatusFlags(ADC_Type *base, uint32_t mask)`
Clear trigger status flags.

Parameters

- `base` – LPADC peripheral base address.

- mask – The mask of trigger status flags to be cleared, should be the OR'ed value of `_lpadc_trigger_status_flags`.

static inline void LPADC__EnableInterrupts(ADC_Type *base, uint32_t mask)

Enable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to `_lpadc_interrupt_enable`.

static inline void LPADC__DisableInterrupts(ADC_Type *base, uint32_t mask)

Disable interrupts.

Parameters

- base – LPADC peripheral base address.
- mask – Mask value for interrupt events. See to `_lpadc_interrupt_enable`.

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

Switch on/off the DMA trigger for FIFO watermark event.

Parameters

- base – LPADC peripheral base address.
- enable – Switcher to the event.

static inline uint32_t LPADC__GetConvResultCount(ADC_Type *base)

Get the count of result kept in conversion FIFO.

Parameters

- base – LPADC peripheral base address.

Returns

The count of result kept in conversion FIFO.

bool LPADC__GetConvResult(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

Returns

Status whether FIFO entry is valid.

void LPADC__GetConvResultBlocking(ADC_Type *base, *lpadc_conv_result_t* *result)

Get the result in conversion FIFO using blocking method.

Parameters

- base – LPADC peripheral base address.
- result – Pointer to structure variable that keeps the conversion result in conversion FIFO.

void LPADC__SetConvTriggerConfig(ADC_Type *base, uint32_t triggerId, const *lpadc_conv_trigger_config_t* *config)

Configure the conversion trigger source.

Each programmable trigger can launch the conversion command in command buffer.

Parameters

- base – LPADC peripheral base address.
- triggerId – ID for each trigger. Typically, the available value range is from 0.
- config – Pointer to configuration structure. See to `lpadc_conv_trigger_config_t`.

void LPADC_GetDefaultConvTriggerConfig(*lpadc_conv_trigger_config_t* *config)

Gets an available pre-defined settings for trigger's configuration.

This function initializes the trigger's configuration structure with an available settings. The default values are:

```
config->targetCommandId    = 0U;  
config->delayPower         = 0U;  
config->priority           = 0U;  
config->channelAFIFOSelect = 0U;  
config->channelBFIFOSelect = 0U;  
config->enableHardwareTrigger = false;
```

Parameters

- config – Pointer to configuration structure.

static inline void LPADC_DoSoftwareTrigger(ADC_Type *base, uint32_t triggerIdMask)

Do software trigger to conversion command.

Parameters

- base – LPADC peripheral base address.
- triggerIdMask – Mask value for software trigger indexes, which count from zero.

void LPADC_SetConvCommandConfig(ADC_Type *base, uint32_t commandId, const *lpadc_conv_command_config_t* *config)

Configure conversion command.

Note: The number of compare value register on different chips is different, that is mean in some chips, some command buffers do not have the compare functionality.

Parameters

- base – LPADC peripheral base address.
- commandId – ID for command in command buffer. Typically, the available value range is 1 - 15.
- config – Pointer to configuration structure. See to `lpadc_conv_command_config_t`.

void LPADC_GetDefaultConvCommandConfig(*lpadc_conv_command_config_t* *config)

Gets an available pre-defined settings for conversion command's configuration.

This function initializes the conversion command's configuration structure with an available settings. The default values are:

```
config->sampleScaleMode    = kLPADC_SampleFullScale;  
config->channelBScaleMode  = kLPADC_SampleFullScale;  
config->sampleChannelMode  = kLPADC_SampleChannelSingleEndSideA;  
config->channelNumber      = 0U;  
config->channelBNumber     = 0U;
```

(continues on next page)

(continued from previous page)

```

config->chainedNextCommandNumber = 0U;
config->enableAutoChannelIncrement = false;
config->loopCount = 0U;
config->hardwareAverageMode = kLPADC_HardwareAverageCount1;
config->sampleTimeMode = kLPADC_SampleTimeADCK3;
config->hardwareCompareMode = kLPADC_HardwareCompareDisabled;
config->hardwareCompareValueHigh = 0U;
config->hardwareCompareValueLow = 0U;
config->conversionResolutionMode = kLPADC_ConversionResolutionStandard;
config->enableWaitTrigger = false;
config->enableChannelB = false;

```

Parameters

- config – Pointer to configuration structure.

void LPADC_EnableCalibration(ADC_Type *base, bool enable)

Enable the calibration function.

When CALOFS is set, the ADC is configured to perform a calibration function anytime the ADC executes a conversion. Any channel selected is ignored and the value returned in the RESFIFO is a signed value between -31 and 31. -32 is not a valid and is never a returned value. Software should copy the lower 6- bits of the conversion result stored in the RESFIFO after a completed calibration conversion to the OFSTRIM field. The OFSTRIM field is used in normal operation for offset correction.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

static inline void LPADC_SetOffsetValue(ADC_Type *base, uint32_t value)

Set proper offset value to trim ADC.

To minimize the offset during normal operation, software should read the conversion result from the RESFIFO calibration operation and write the lower 6 bits to the OFSTRIM register.

Parameters

- base – LPADC peripheral base address.
- value – Setting offset value.

status_t LPADC_DoAutoCalibration(ADC_Type *base)

Do auto calibration.

Calibration function should be executed before using converter in application. It used the software trigger and a dummy conversion, get the offset and write them into the OFSTRIM register. It called some of functional API including: -LPADC_EnableCalibration(...) -LPADC_LPADC_SetOffsetValue(...) -LPADC_SetConvCommandConfig(...) -LPADC_SetConvTriggerConfig(...)

Parameters

- base – LPADC peripheral base address.
- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
static inline void LPADC_SetOffsetValue(ADC_Type *base, int16_t value)
```

Set trim value for offset.

Note: For 16-bit conversions, each increment is 1/2 LSB resulting in a programmable offset range of -256 LSB to 255.5 LSB; For 12-bit conversions, each increment is 1/32 LSB resulting in a programmable offset range of -16 LSB to 15.96875 LSB.

Parameters

- base – LPADC peripheral base address.
- value – Offset trim value, is a 10-bit signed value between -512 and 511.

```
static inline void LPADC_GetOffsetValue(ADC_Type *base, int16_t *pValue)
```

Get trim value of offset.

Parameters

- base – LPADC peripheral base address.
- pValue – Pointer to the variable in type of int16_t to store offset value.

```
static inline void LPADC_EnableOffsetCalibration(ADC_Type *base, bool enable)
```

Enable the offset calibration function.

Parameters

- base – LPADC peripheral base address.
- enable – switcher to the calibration function.

```
static inline void LPADC_SetOffsetCalibrationMode(ADC_Type *base,  
                                                  lpadc_offset_calibration_mode_t mode)
```

Set offset calibration mode.

Parameters

- base – LPADC peripheral base address.
- mode – set offset calibration mode.see to lpadc_offset_calibration_mode_t .

```
status_t LPADC_DoOffsetCalibration(ADC_Type *base)
```

Do offset calibration.

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

```
void LPADC_PrepareAutoCalibration(ADC_Type *base)
```

Prepare auto calibration, LPADC_FinishAutoCalibration has to be called before using the LPADC. LPADC_DoAutoCalibration has been split in two API to avoid to be stuck too long in the function.

Parameters

- base – LPADC peripheral base address.

status_t LPADC_FinishAutoCalibration(ADC_Type *base)

Finish auto calibration start with LPADC_PrepareAutoCalibration.

Note: This feature is used for LPADC with CTRL[CALOFSMODE].

Parameters

- base – LPADC peripheral base address.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

void LPADC_GetCalibrationValue(ADC_Type *base, *lpadc_calibration_value_t* *ptrCalibrationValue)

Get calibration value into the memory which is defined by invoker.

Note: Please note the ADC will be disabled temporary.

Note: This function should be used after finish calibration.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to *lpadc_calibration_value_t* structure, this memory block should be always powered on even in low power modes.

status_t LPADC_SetCalibrationValue(ADC_Type *base, const *lpadc_calibration_value_t* *ptrCalibrationValue)

Set calibration value into ADC calibration registers.

Note: Please note the ADC will be disabled temporary.

Parameters

- base – LPADC peripheral base address.
- ptrCalibrationValue – Pointer to *lpadc_calibration_value_t* structure which contains ADC's calibration value.

Return values

- kStatus_Success – Successfully configured.
- kStatus_Timeout – Timeout occurs while waiting completion.

FSL_LPADC_DRIVER_VERSION

LPADC driver version 2.9.3.

struct *lpadc_config_t*

#include <fsl_lpadc.h> LPADC global configuration.

This structure would used to keep the settings for initialization.

Public Members

`bool enableInternalClock`

Enables the internally generated clock source. The clock source is used in clock selection logic at the chip level and is optionally used for the ADC clock source.

`bool enableVref1LowVoltage`

If voltage reference option1 input is below 1.8V, it should be “true”. If voltage reference option1 input is above 1.8V, it should be “false”.

`bool enableInDozeMode`

Control system transition to Stop and Wait power modes while ADC is converting. When enabled in Doze mode, immediate entries to Wait or Stop are allowed. When disabled, the ADC will wait for the current averaging iteration/FIFO storage to complete before acknowledging stop or wait mode entry.

`lpadc_conversion_average_mode_t conversionAverageMode`

Auto-Calibration Averages.

`bool enableAnalogPreliminary`

ADC analog circuits are pre-enabled and ready to execute conversions without startup delays(at the cost of higher DC current consumption).

`uint32_t powerUpDelay`

When the analog circuits are not pre-enabled, the ADC analog circuits are only powered while the ADC is active and there is a counted delay defined by this field after an initial trigger transitions the ADC from its Idle state to allow time for the analog circuits to stabilize. The startup delay count of (`powerUpDelay * 4`) ADCK cycles must result in a longer delay than the analog startup time.

`lpadc_reference_voltage_source_t referenceVoltageSource`

Selects the voltage reference high used for conversions.

`lpadc_power_level_mode_t powerLevelMode`

Power Configuration Selection.

`lpadc_trigger_priority_policy_t triggerPriorityPolicy`

Control how higher priority triggers are handled, see to `lpadc_trigger_priority_policy_t`.

`bool enableConvPause`

Enables the ADC pausing function. When enabled, a programmable delay is inserted during command execution sequencing between LOOP iterations, between commands in a sequence, and between conversions when command is executing in “Compare Until True” configuration.

`uint32_t convPauseDelay`

Controls the duration of pausing during command execution sequencing. The pause delay is a count of (`convPauseDelay*4`) ADCK cycles. Only available when ADC pausing function is enabled. The available value range is in 9-bit.

`uint32_t FIFOWatermark`

FIFOWatermark is a programmable threshold setting. When the number of datawords stored in the ADC Result FIFO is greater than the value in this field, the ready flag would be asserted to indicate stored data has reached the programmable threshold.

`struct lpadc_conv_command_config_t`

`#include <fsl_lpadc.h>` Define structure to keep the configuration for conversion command.

Public Members

lpadc_sample_scale_mode_t sampleScaleMode

Sample scale mode.

lpadc_sample_scale_mode_t channelBScaleMode

Alternate channel B Scale mode.

lpadc_sample_channel_mode_t sampleChannelMode

Channel sample mode.

uint32_t channelNumber

Channel number; select the channel or channel pair.

uint32_t channelBNumber

Alternate Channel B number; select the channel.

uint32_t chainedNextCommandNumber

Selects the next command to be executed after this command completes. 1-15 is available, 0 is to terminate the chain after this command.

bool enableAutoChannelIncrement

Loop with increment: when disabled, the “loopCount” field selects the number of times the selected channel is converted consecutively; when enabled, the “loopCount” field defines how many consecutive channels are converted as part of the command execution.

uint32_t loopCount

Selects how many times this command executes before finish and transition to the next command or Idle state. Command executes LOOP+1 times. 0-15 is available.

lpadc_hardware_average_mode_t hardwareAverageMode

Hardware average selection.

lpadc_sample_time_mode_t sampleTimeMode

Sample time selection.

lpadc_hardware_compare_mode_t hardwareCompareMode

Hardware compare selection.

uint32_t hardwareCompareValueHigh

Compare Value High. The available value range is in 16-bit.

uint32_t hardwareCompareValueLow

Compare Value Low. The available value range is in 16-bit.

lpadc_conversion_resolution_mode_t conversionResolutionMode

Conversion resolution mode.

bool enableWaitTrigger

Wait for trigger assertion before execution: when disabled, this command will be automatically executed; when enabled, the active trigger must be asserted again before executing this command.

struct lpadc_conv_trigger_config_t

#include <fsl_lpadc.h> Define structure to keep the configuration for conversion trigger.

Public Members

uint32_t targetCommandId

Select the command from command buffer to execute upon detect of the associated trigger event.

uint32_t delayPower

Select the trigger delay duration to wait at the start of servicing a trigger event. When this field is clear, then no delay is incurred. When this field is set to a non-zero value, the duration for the delay is $2^{\text{delayPower}}$ ADCK cycles. The available value range is 4-bit.

uint32_t priority

Sets the priority of the associated trigger source. If two or more triggers have the same priority level setting, the lower order trigger event has the higher priority. The lower value for this field is for the higher priority, the available value range is 1-bit.

bool enableHardwareTrigger

Enable hardware trigger source to initiate conversion on the rising edge of the input trigger source or not. The software trigger is always available.

struct lpadc_conv_result_t

#include <fsl_lpadc.h> Define the structure to keep the conversion result.

Public Members

uint32_t commandIdSource

Indicate the command buffer being executed that generated this result.

uint32_t loopCountIndex

Indicate the loop count value during command execution that generated this result.

uint32_t triggerIdSource

Indicate the trigger source that initiated a conversion and generated this result.

uint16_t convValue

Data result.

struct __lpadc_calibration_value

#include <fsl_lpadc.h> A structure of calibration value.

2.32 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,
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

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.

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

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

2.33 IOCON: I/O pin configuration

FSL_IOCON_DRIVER_VERSION

IOCON driver version.

```
typedef struct iocon_group iocon_group_t
```

Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format.

```
__STATIC_INLINE void IOCON_PinMuxSet (IOCON_Type *base, uint8_t port, uint8_t pin,  
uint32_t modefunc)
```

Sets I/O Control pin mux.

Parameters

- base – : The base of IOCON peripheral on the chip
- port – : GPIO port to mux
- pin – : GPIO pin to mux
- modefunc – : OR'ed values of type IOCON_*

Returns

Nothing

```
__STATIC_INLINE void IOCON_SetPinMuxing (IOCON_Type *base,  
const iocon_group_t *pinArray, uint32_t arrayLength)
```

Set all I/O Control pin muxing.

Parameters

- base – : The base of IOCON peripheral on the chip
- pinArray – : Pointer to array of pin mux selections
- arrayLength – : Number of entries in pinArray

Returns

Nothing

FSL_COMPONENT_ID

IOCON_FUNC0

IOCON function and mode selection definitions.

Note: See the User Manual for specific modes and functions supported by the various pins.
Selects pin function 0

IOCON_FUNC1

Selects pin function 1

IOCON_FUNC2

Selects pin function 2

IOCON_FUNC3

Selects pin function 3

IOCON_FUNC4

Selects pin function 4

IOCON_FUNC5

Selects pin function 5

IOCON_FUNC6

Selects pin function 6

IOCON_FUNC7

Selects pin function 7

struct _iocon_group

#include <fsl_iocon.h> Array of IOCON pin definitions passed to IOCON_SetPinMuxing() must be in this format.

2.34 Mailbox

static inline void MAILBOX_Init(MAILBOX_Type *base)

Initializes the MAILBOX module.

This function enables the MAILBOX clock only.

Parameters

- base – MAILBOX peripheral base address.

static inline void MAILBOX_Deinit(MAILBOX_Type *base)

De-initializes the MAILBOX module.

This function disables the MAILBOX clock only.

Parameters

- base – MAILBOX peripheral base address.

FSL_MAILBOX_DRIVER_VERSION

MAILBOX driver version.

static inline uint32_t MAILBOX_GetMutex(MAILBOX_Type *base)

Get MUTEX state and lock mutex.

Note: Returns '1' if the mutex was taken or '0' if another resources has the mutex locked. Once a mutex is taken, it can be returned with the MAILBOX_SetMutex() function.

Parameters

- base – MAILBOX peripheral base address.

Returns

See note

static inline void MAILBOX_SetMutex(MAILBOX_Type *base)

Set MUTEX state.

Note: Sets mutex state to '1' and allows other resources to get the mutex.

Parameters

- base – MAILBOX peripheral base address.

FSL_COMPONENT_ID

2.35 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

2.36 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 may be gray-code, if so, please using OSTIMER_SetMatchValue().

Parameters

- *base* – OSTIMER peripheral base address.
- *count* – OSTIMER timer match value.(Value may be 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 in API if the IP counter is gray encoded.)
- *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 uint64_t OSTIMER_GetMatchRegister(OSTIMER_Type *base)

Get the match value from OSTIMER.

This function will get the match value from OSTIMER. The value of timer match is gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of match register, data format is gray code.

static inline uint64_t OSTIMER_GetMatchValue(OSTIMER_Type *base)

Get the match value from OSTIMER.

This function will get a match value from OSTIMER.

Parameters

- base – OSTIMER peripheral base address.

Returns

Value of match register.

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 the timer count value from OS timer register. The raw value of timer count may be gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of OSTIMER, may be 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. If the RAW value of timer count is gray code format, it 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 value from OSTIMER. The Raw value of timer capture may be gray code format.

Parameters

- base – OSTIMER peripheral base address.

Returns

Raw value of capture register, data format may be 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. If the RAW value of timer count is gray code format, it 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.

2.37 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

enumerator `kPINT_SecPinInt0`

Secure 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

enumerator kPINT_SecPatternMatchBSlice0
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, pint_pin_int_t 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, pint_pin_int_t 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>
```

2.38 PLU: Programmable Logic Unit

```
void PLU_Init(PLU_Type *base)
    Enable the PLU clock and reset the module.
```

Note: This API should be called at the beginning of the application using the PLU driver.

Parameters

- base – PLU peripheral base address

```
void PLU_Deinit(PLU_Type *base)
    Gate the PLU clock.
```

Parameters

- base – PLU peripheral base address

```
static inline void PLU_SetLutInputSource(PLU_Type *base, plu_lut_index_t lutIndex,
                                         plu_lut_in_index_t lutInIndex, plu_lut_input_source_t
                                         inputSrc)
```

Set Input source of LUT.

Note: An external clock must be applied to the PLU_CLKIN input when using FFs. For each LUT, the slot associated with the output from LUTn itself is tied low.

Parameters

- base – PLU peripheral base address.
- lutIndex – LUT index (see plu_lut_index_t typedef enumeration).
- lutInIndex – LUT input index (see plu_lut_in_index_t typedef enumeration).
- inputSrc – LUT input source (see plu_lut_input_source_t typedef enumeration).

```
static inline void PLU_SetOutputSource(PLU_Type *base, plu_output_index_t outputIndex,  
                                     plu_output_source_t outputSrc)
```

Set Output source of PLU.

Note: An external clock must be applied to the PLU_CLKIN input when using FFs.

Parameters

- base – PLU peripheral base address.
- outputIndex – PLU output index (see *plu_output_index_t* typedef enumeration).
- outputSrc – PLU output source (see *plu_output_source_t* typedef enumeration).

```
static inline void PLU_SetLutTruthTable(PLU_Type *base, plu_lut_index_t lutIndex, uint32_t  
                                       truthTable)
```

Set Truth Table of LUT.

Parameters

- base – PLU peripheral base address.
- lutIndex – LUT index (see *plu_lut_index_t* typedef enumeration).
- truthTable – Truth Table value.

```
static inline uint32_t PLU_ReadOutputState(PLU_Type *base)
```

Read the current state of the 8 designated PLU Outputs.

Note: The PLU bus clock must be re-enabled prior to reading the Outpus Register if PLU bus clock is shut-off.

Parameters

- base – PLU peripheral base address.

Returns

Current PLU output state value.

```
void PLU_GetDefaultWakeIntConfig(plu_wakeint_config_t *config)
```

Gets an available pre-defined settings for wakeup/interrupt control.

This function initializes the initial configuration structure with an available settings. The default values are:

```
config->filterMode = kPLU_WAKEINT_FILTER_MODE_BYPASS;  
config->clockSource = kPLU_WAKEINT_FILTER_CLK_SRC_1MHZ_LPOSC;
```

Parameters

- config – Pointer to configuration structure.

```
void PLU_EnableWakeIntRequest(PLU_Type *base, uint32_t interruptMask, const  
                             plu_wakeint_config_t *config)
```

Enable PLU outputs wakeup/interrupt request.

This function enables Any of the eight selected PLU outputs to contribute to an asynchronous wake-up or an interrupt request.

Note: If a PLU_CLKIN is provided, the raw wake-up/interrupt request will be set on the rising-edge of the PLU_CLKIN whenever the raw request signal is high. This registered signal will be glitch-free and just use the default wakeint config by `PLU_GetDefaultWakeIntConfig()`. If not, have to specify the filter mode and clock source to eliminate the glitches caused by long and widely disparate delays through the network

of LUTs making up the PLU. This way may increase power consumption in low-power operating modes and inject delay before the wake-up/interrupt request is generated.

Parameters

- `base` – PLU peripheral base address.
- `interruptMask` – PLU interrupt mask (see `_plu_interrupt_mask` enumeration).
- `config` – Pointer to configuration structure (see `plu_wakeint_config_t` typedef enumeration)

```
static inline void PLU_LatchInterrupt(PLU_Type *base)
```

Latch an interrupt.

This function latches the interrupt and then it can be cleared with `PLU_ClearLatchedInterrupt()`.

Note: This mode is not compatible with use of the glitch filter. If this bit is set, the `FILTER MODE` should be set to `kPLU_WAKEINT_FILTER_MODE_BYPASS` (Bypass Mode) and `PLU_CLKIN` should be provided. If this bit is set, the wake-up/interrupt request will be set on the rising-edge of `PLU_CLKIN` whenever the raw wake-up/interrupt signal is high. The request must be cleared by software.

Parameters

- `base` – PLU peripheral base address.

```
void PLU_ClearLatchedInterrupt(PLU_Type *base)
```

Clear the latched interrupt.

This function clears the wake-up/interrupt request flag latched by `PLU_LatchInterrupt()`

Note: It is not necessary for the PLU bus clock to be enabled in order to write-to or read-back this bit.

Parameters

- `base` – PLU peripheral base address.

```
FSL_PLU_DRIVER_VERSION
```

Version 2.2.1

```
enum _plu_lut_index
```

Index of LUT.

Values:

```
enumerator kPLU_LUT_0
```

5-input Look-up Table 0

```
enumerator kPLU_LUT_1
```

5-input Look-up Table 1

```
enumerator kPLU_LUT_2
```

5-input Look-up Table 2

```
enumerator kPLU_LUT_3
```

5-input Look-up Table 3

```
enumerator kPLU_LUT_4
```

5-input Look-up Table 4

```
enumerator kPLU_LUT_5
```

5-input Look-up Table 5

enumerator kPLU_LUT_6
5-input Look-up Table 6

enumerator kPLU_LUT_7
5-input Look-up Table 7

enumerator kPLU_LUT_8
5-input Look-up Table 8

enumerator kPLU_LUT_9
5-input Look-up Table 9

enumerator kPLU_LUT_10
5-input Look-up Table 10

enumerator kPLU_LUT_11
5-input Look-up Table 11

enumerator kPLU_LUT_12
5-input Look-up Table 12

enumerator kPLU_LUT_13
5-input Look-up Table 13

enumerator kPLU_LUT_14
5-input Look-up Table 14

enumerator kPLU_LUT_15
5-input Look-up Table 15

enumerator kPLU_LUT_16
5-input Look-up Table 16

enumerator kPLU_LUT_17
5-input Look-up Table 17

enumerator kPLU_LUT_18
5-input Look-up Table 18

enumerator kPLU_LUT_19
5-input Look-up Table 19

enumerator kPLU_LUT_20
5-input Look-up Table 20

enumerator kPLU_LUT_21
5-input Look-up Table 21

enumerator kPLU_LUT_22
5-input Look-up Table 22

enumerator kPLU_LUT_23
5-input Look-up Table 23

enumerator kPLU_LUT_24
5-input Look-up Table 24

enumerator kPLU_LUT_25
5-input Look-up Table 25

enum __plu_lut_in_index
Inputs of LUT. 5 input present for each LUT.

Values:

enumerator kPLU_LUT_IN_0
LUT input 0

enumerator kPLU_LUT_IN_1
LUT input 1

enumerator kPLU_LUT_IN_2
LUT input 2

enumerator kPLU_LUT_IN_3
LUT input 3

enumerator kPLU_LUT_IN_4
LUT input 4

enum _plu_lut_input_source
Available sources of LUT input.

Values:

enumerator kPLU_LUT_IN_SRC_PLU_IN_0
Select PLU input 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_1
Select PLU input 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_2
Select PLU input 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_3
Select PLU input 3 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_4
Select PLU input 4 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_PLU_IN_5
Select PLU input 5 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_0
Select LUT output 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_1
Select LUT output 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_2
Select LUT output 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_3
Select LUT output 3 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_4
Select LUT output 4 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_5
Select LUT output 5 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_6
Select LUT output 6 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_7
Select LUT output 7 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_8
Select LUT output 8 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_9
Select LUT output 9 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_10
Select LUT output 10 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_11
Select LUT output 11 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_12
Select LUT output 12 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_13
Select LUT output 13 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_14
Select LUT output 14 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_15
Select LUT output 15 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_16
Select LUT output 16 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_17
Select LUT output 17 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_18
Select LUT output 18 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_19
Select LUT output 19 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_20
Select LUT output 20 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_21
Select LUT output 21 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_22
Select LUT output 22 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_23
Select LUT output 23 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_24
Select LUT output 24 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_LUT_OUT_25
Select LUT output 25 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_0
Select Flip-Flops state 0 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_1
Select Flip-Flops state 1 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_2
Select Flip-Flops state 2 to be connected to LUTn Input x

enumerator kPLU_LUT_IN_SRC_FLIPFLOP_3

Select Flip-Flops state 3 to be connected to LUTn Input x

enum __plu_output_index

PLU output multiplexer registers.

Values:

enumerator kPLU_OUTPUT_0

PLU OUTPUT 0

enumerator kPLU_OUTPUT_1

PLU OUTPUT 1

enumerator kPLU_OUTPUT_2

PLU OUTPUT 2

enumerator kPLU_OUTPUT_3

PLU OUTPUT 3

enumerator kPLU_OUTPUT_4

PLU OUTPUT 4

enumerator kPLU_OUTPUT_5

PLU OUTPUT 5

enumerator kPLU_OUTPUT_6

PLU OUTPUT 6

enumerator kPLU_OUTPUT_7

PLU OUTPUT 7

enum __plu_output_source

Available sources of PLU output.

Values:

enumerator kPLU_OUT_SRC_LUT_0

Select LUT0 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_1

Select LUT1 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_2

Select LUT2 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_3

Select LUT3 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_4

Select LUT4 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_5

Select LUT5 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_6

Select LUT6 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_7

Select LUT7 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_8

Select LUT8 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_9
Select LUT9 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_10
Select LUT10 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_11
Select LUT11 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_12
Select LUT12 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_13
Select LUT13 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_14
Select LUT14 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_15
Select LUT15 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_16
Select LUT16 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_17
Select LUT17 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_18
Select LUT18 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_19
Select LUT19 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_20
Select LUT20 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_21
Select LUT21 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_22
Select LUT22 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_23
Select LUT23 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_24
Select LUT24 output to be connected to PLU output

enumerator kPLU_OUT_SRC_LUT_25
Select LUT25 output to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_0
Select Flip-Flops state(0) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_1
Select Flip-Flops state(1) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_2
Select Flip-Flops state(2) to be connected to PLU output

enumerator kPLU_OUT_SRC_FLIPFLOP_3
Select Flip-Flops state(3) to be connected to PLU output

enum `_plu_interrupt_mask`

The enumerator of PLU Interrupt.

Values:

enumerator `kPLU_OUTPUT_0_INTERRUPT_MASK`

Select PLU output 0 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_1_INTERRUPT_MASK`

Select PLU output 1 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_2_INTERRUPT_MASK`

Select PLU output 2 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_3_INTERRUPT_MASK`

Select PLU output 3 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_4_INTERRUPT_MASK`

Select PLU output 4 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_5_INTERRUPT_MASK`

Select PLU output 5 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_6_INTERRUPT_MASK`

Select PLU output 6 contribute to interrupt/wake-up generation

enumerator `kPLU_OUTPUT_7_INTERRUPT_MASK`

Select PLU output 7 contribute to interrupt/wake-up generation

enum `_plu_wakeint_filter_mode`

Control input of the PLU, add filtering for glitch.

Values:

enumerator `kPLU_WAKEINT_FILTER_MODE_BYPASS`

Select Bypass mode

enumerator `kPLU_WAKEINT_FILTER_MODE_1_CLK_PERIOD`

Filter 1 clock period

enumerator `kPLU_WAKEINT_FILTER_MODE_2_CLK_PERIOD`

Filter 2 clock period

enumerator `kPLU_WAKEINT_FILTER_MODE_3_CLK_PERIOD`

Filter 3 clock period

enum `_plu_wakeint_filter_clock_source`

Clock source for filter mode.

Values:

enumerator `kPLU_WAKEINT_FILTER_CLK_SRC_1MHZ_LPOSC`

Select the 1MHz low-power oscillator as the filter clock

enumerator `kPLU_WAKEINT_FILTER_CLK_SRC_12MHZ_FRO`

Select the 12MHz FRO as the filter clock

enumerator `kPLU_WAKEINT_FILTER_CLK_SRC_ALT`

Select a third clock source

typedef enum `_plu_lut_index` `plu_lut_index_t`

Index of LUT.

```
typedef enum _plu_lut_in_index plu_lut_in_index_t
    Inputs of LUT. 5 input present for each LUT.
typedef enum _plu_lut_input_source plu_lut_input_source_t
    Available sources of LUT input.
typedef enum _plu_output_index plu_output_index_t
    PLU output multiplexer registers.
typedef enum _plu_output_source plu_output_source_t
    Available sources of PLU output.
typedef enum _plu_wakeint_filter_mode plu_wakeint_filter_mode_t
    Control input of the PLU, add filtering for glitch.
typedef enum _plu_wakeint_filter_clock_source plu_wakeint_filter_clock_source_t
    Clock source for filter mode.
typedef struct _plu_wakeint_config plu_wakeint_config_t
    Wake configuration.
struct _plu_wakeint_config
    #include <fsl_plu.h> Wake configuration.
```

Public Members

```
plu_wakeint_filter_mode_t filterMode
    Filter Mode.
plu_wakeint_filter_clock_source_t clockSource
    The clock source for filter mode.
```

2.39 Power Driver

```
enum _power_mode_config
    Values:
    enumerator kPmu__Sleep
    enumerator kPmu__Deep__Sleep
    enumerator kPmu__PowerDown
    enumerator kPmu__Deep__PowerDown
enum pd_bits
    Analog components power modes control during low power modes.
    Values:
    enumerator kPDRUNCFG_PD_DCDC
    enumerator kPDRUNCFG_PD_BIAS
    enumerator kPDRUNCFG_PD_BODCORE
    enumerator kPDRUNCFG_PD_BODVBAT
    enumerator kPDRUNCFG_PD_FRO1M
```

enumerator kPDRUNCFG_PD_FRO192M
enumerator kPDRUNCFG_PD_FRO32K
enumerator kPDRUNCFG_PD_XTAL32K
enumerator kPDRUNCFG_PD_XTAL32M
enumerator kPDRUNCFG_PD_PLL0
enumerator kPDRUNCFG_PD_PLL1
enumerator kPDRUNCFG_PD_USB0_PHY
enumerator kPDRUNCFG_PD_USB1_PHY
enumerator kPDRUNCFG_PD_COMP
enumerator kPDRUNCFG_PD_TEMPSENS
enumerator kPDRUNCFG_PD_GPADC
enumerator kPDRUNCFG_PD_LDOMEM
enumerator kPDRUNCFG_PD_LDODEEPSLEEP
enumerator kPDRUNCFG_PD_LDOUSBHS
enumerator kPDRUNCFG_PD_LDOGPADC
enumerator kPDRUNCFG_PD_LDOXO32M
enumerator kPDRUNCFG_PD_LDOFLASHNV
enumerator kPDRUNCFG_PD_RNG
enumerator kPDRUNCFG_PD_PLL0_SSCG
enumerator kPDRUNCFG_PD_ROM
enumerator kPDRUNCFG_ForceUnsigned

enum _power_bod_vbat_level

BOD VBAT level.

Values:

enumerator kPOWER_BodVbatLevel1000mv
Brown out detector VBAT level 1V
enumerator kPOWER_BodVbatLevel1100mv
Brown out detector VBAT level 1.1V
enumerator kPOWER_BodVbatLevel1200mv
Brown out detector VBAT level 1.2V
enumerator kPOWER_BodVbatLevel1300mv
Brown out detector VBAT level 1.3V
enumerator kPOWER_BodVbatLevel1400mv
Brown out detector VBAT level 1.4V
enumerator kPOWER_BodVbatLevel1500mv
Brown out detector VBAT level 1.5V

enumerator kPOWER__BodVbatLevel1600mv
Brown out detector VBAT level 1.6V

enumerator kPOWER__BodVbatLevel1650mv
Brown out detector VBAT level 1.65V

enumerator kPOWER__BodVbatLevel1700mv
Brown out detector VBAT level 1.7V

enumerator kPOWER__BodVbatLevel1750mv
Brown out detector VBAT level 1.75V

enumerator kPOWER__BodVbatLevel1800mv
Brown out detector VBAT level 1.8V

enumerator kPOWER__BodVbatLevel1900mv
Brown out detector VBAT level 1.9V

enumerator kPOWER__BodVbatLevel2000mv
Brown out detector VBAT level 2V

enumerator kPOWER__BodVbatLevel2100mv
Brown out detector VBAT level 2.1V

enumerator kPOWER__BodVbatLevel2200mv
Brown out detector VBAT level 2.2V

enumerator kPOWER__BodVbatLevel2300mv
Brown out detector VBAT level 2.3V

enumerator kPOWER__BodVbatLevel2400mv
Brown out detector VBAT level 2.4V

enumerator kPOWER__BodVbatLevel2500mv
Brown out detector VBAT level 2.5V

enumerator kPOWER__BodVbatLevel2600mv
Brown out detector VBAT level 2.6V

enumerator kPOWER__BodVbatLevel2700mv
Brown out detector VBAT level 2.7V

enumerator kPOWER__BodVbatLevel2806mv
Brown out detector VBAT level 2.806V

enumerator kPOWER__BodVbatLevel2900mv
Brown out detector VBAT level 2.9V

enumerator kPOWER__BodVbatLevel3000mv
Brown out detector VBAT level 3.0V

enumerator kPOWER__BodVbatLevel3100mv
Brown out detector VBAT level 3.1V

enumerator kPOWER__BodVbatLevel3200mv
Brown out detector VBAT level 3.2V

enumerator kPOWER__BodVbatLevel3300mv
Brown out detector VBAT level 3.3V

enum __power_bod_hyst
BOD Hysteresis control.

Values:

enumerator kPOWER__BodHystLevel25mv

BOD Hysteresis control level 25mv

enumerator kPOWER__BodHystLevel50mv

BOD Hysteresis control level 50mv

enumerator kPOWER__BodHystLevel75mv

BOD Hysteresis control level 75mv

enumerator kPOWER__BodHystLevel100mv

BOD Hysteresis control level 100mv

enum __power__bod__core__level

BOD core level.

Values:

enumerator kPOWER__BodCoreLevel600mv

Brown out detector core level 600mV

enumerator kPOWER__BodCoreLevel650mv

Brown out detector core level 650mV

enumerator kPOWER__BodCoreLevel700mv

Brown out detector core level 700mV

enumerator kPOWER__BodCoreLevel750mv

Brown out detector core level 750mV

enumerator kPOWER__BodCoreLevel800mv

Brown out detector core level 800mV

enumerator kPOWER__BodCoreLevel850mv

Brown out detector core level 850mV

enumerator kPOWER__BodCoreLevel900mv

Brown out detector core level 900mV

enumerator kPOWER__BodCoreLevel950mv

Brown out detector core level 950mV

enum __power__device__reset__cause

Device Reset Causes.

Values:

enumerator kRESET__CAUSE__POR

Power On Reset

enumerator kRESET__CAUSE__PADRESET

Hardware Pin Reset

enumerator kRESET__CAUSE__BODRESET

Brown-out Detector reset (either BODVBAT or BODCORE)

enumerator kRESET__CAUSE__ARMSYSTEMRESET

ARM System Reset

enumerator kRESET__CAUSE__WDTRRESET

Watchdog Timer Reset

enumerator kRESET__CAUSE__SWRRESET

Software Reset

enumerator kRESET_CAUSE_CDOGRESET

Code Watchdog Reset

enumerator kRESET_CAUSE_DPDRESET_WAKEUIPIO

Any of the 4 wake-up pins

enumerator kRESET_CAUSE_DPDRESET_RTC

Real Time Counter (RTC)

enumerator kRESET_CAUSE_DPDRESET_OSTIMER

OS Event Timer (OSTIMER)

enumerator kRESET_CAUSE_DPDRESET_WAKEUIPIO_RTC

Any of the 4 wake-up pins and RTC (it is not possible to distinguish which of these 2 events occurred first)

enumerator kRESET_CAUSE_DPDRESET_WAKEUIPIO_OSTIMER

Any of the 4 wake-up pins and OSTIMER (it is not possible to distinguish which of these 2 events occurred first)

enumerator kRESET_CAUSE_DPDRESET_RTC_OSTIMER

Real Time Counter or OS Event Timer (it is not possible to distinguish which of these 2 events occurred first)

enumerator kRESET_CAUSE_DPDRESET_WAKEUIPIO_RTC_OSTIMER

Any of the 4 wake-up pins (it is not possible to distinguish which of these 3 events occurred first)

enumerator kRESET_CAUSE_NOT_RELEVANT

No reset cause (for example, this code is used when waking up from DEEP-SLEEP low power mode)

enumerator kRESET_CAUSE_NOT_DETERMINISTIC

Unknown Reset Cause. Should be treated like “Hardware Pin Reset” from an application point of view.

enum __power_device_boot_mode

Device Boot Modes.

Values:

enumerator kBOOT_MODE_POWER_UP

All non Low Power Mode wake up (Power On Reset, Pin Reset, BoD Reset, ARM System Reset ...)

enumerator kBOOT_MODE_LP_DEEP_SLEEP

Wake up from DEEP-SLEEP Low Power mode

enumerator kBOOT_MODE_LP_POWER_DOWN

Wake up from POWER-DOWN Low Power mode

enumerator kBOOT_MODE_LP_DEEP_POWER_DOWN

Wake up from DEEP-POWER-DOWN Low Power mode

typedef enum __power_mode_config power_mode_cfg_t

typedef enum pd_bits pd_bit_t

Analog components power modes control during low power modes.

typedef enum __power_bod_vbat_level power_bod_vbat_level_t

BOD VBAT level.

```
typedef enum _power_bod_hyst power_bod_hyst_t
```

BOD Hysteresis control.

```
typedef enum _power_bod_core_level power_bod_core_level_t
```

BOD core level.

```
typedef enum _power_device_reset_cause power_device_reset_cause_t
```

Device Reset Causes.

```
typedef enum _power_device_boot_mode power_device_boot_mode_t
```

Device Boot Modes.

```
static inline void POWER_EnablePD(pd_bit_t en)
```

API to enable PDRUNCFG bit in the Syscon. Note that enabling the bit powers down the peripheral.

Parameters

- *en* – peripheral for which to enable the PDRUNCFG bit

Returns

none

```
static inline void POWER_DisablePD(pd_bit_t en)
```

API to disable PDRUNCFG bit in the Syscon. Note that disabling the bit powers up the peripheral.

Parameters

- *en* – peripheral for which to disable the PDRUNCFG bit

Returns

none

```
static inline void POWER_SetBodVbatLevel(power_bod_vbat_level_t level, power_bod_hyst_t hyst, bool enBodVbatReset)
```

set BOD VBAT level.

Parameters

- *level* – BOD detect level
- *hyst* – BoD Hysteresis control
- *enBodVbatReset* – VBAT brown out detect reset

```
static inline void POWER_EnableDeepSleep(void)
```

API to enable deep sleep bit in the ARM Core.

Returns

none

```
static inline void POWER_DisableDeepSleep(void)
```

API to disable deep sleep bit in the ARM Core.

Returns

none

```
void POWER_CycleCpuAndFlash(void)
```

Shut off the Flash and execute the _WFI(), then power up the Flash after wake-up event This MUST BE EXECUTED outside the Flash: either from ROM or from SRAM. The rest could stay in Flash. But, for consistency, it is preferable to have all functions defined in this file implemented in ROM.

Returns

Nothing

```
void POWER_EnterDeepSleep(uint32_t exclude_from_pd, uint32_t sram_retention_ctrl, uint64_t
                           wakeup_interrupts, uint32_t hardware_wake_ctrl)
```

Configures and enters in DEEP-SLEEP low power mode.

Parameters

- `exclude_from_pd` –
- `sram_retention_ctrl` –
- `wakeup_interrupts` –
- `hardware_wake_ctrl` –

Returns

Nothing

!!! IMPORTANT NOTES :

0 - CPU0 & System CLock frequency is switched to FRO12MHz and is NOT restored back by the API. 1 - CPU0 Interrupt Enable registers (NVIC->ISER) are modified by this function. They are restored back in case of CPU retention or if POWERDOWN is not taken (for instance because an interrupt is pending). 2 - The Non Maskable Interrupt (NMI) is disabled and its configuration before calling this function will be restored back if POWERDOWN is not taken (for instance because an RTC or OSTIMER interrupt is pending). 3 - The HARD FAULT handler should execute from SRAM. (The Hard fault handler should initiate a full chip reset) reset)

```
void POWER_EnterPowerDown(uint32_t exclude_from_pd, uint32_t sram_retention_ctrl,
                           uint64_t wakeup_interrupts, uint32_t cpu_retention_ctrl)
```

Configures and enters in POWERDOWN low power mode.

Parameters

- `exclude_from_pd` –
- `sram_retention_ctrl` –
- `wakeup_interrupts` –
- `cpu_retention_ctrl` – 0 = CPU retention is disable / 1 = CPU retention is enabled, all other values are RESERVED.

Returns

Nothing

!!! IMPORTANT NOTES :

0 - CPU0 & System CLock frequency is switched to FRO12MHz and is NOT restored back by the API. 1 - CPU0 Interrupt Enable registers (NVIC->ISER) are modified by this function. They are restored back in case of CPU retention or if POWERDOWN is not taken (for instance because an interrupt is pending). 2 - The Non Maskable Interrupt (NMI) is disabled and its configuration before calling this function will be restored back if POWERDOWN is not taken (for instance because an RTC or OSTIMER interrupt is pending). 3 - In case of CPU retention, it is the responsibility of the user to make sure that SRAM instance containing the stack used to call this function WILL BE preserved during low power (via parameter “sram_retention_ctrl”) 4 - The HARD FAULT handler should execute from SRAM. (The Hard fault handler should initiate a full chip reset) reset)

```
void POWER_EnterDeepPowerDown(uint32_t exclude_from_pd, uint32_t sram_retention_ctrl,
                               uint64_t wakeup_interrupts, uint32_t wakeup_io_ctrl)
```

Configures and enters in DEEPPowerDOWN low power mode.

Parameters

- `exclude_from_pd` –
- `sram_retention_ctrl` –
- `wakeup_interrupts` –
- `wakeup_io_ctrl` –

Returns

Nothing

!!! IMPORTANT NOTES :

0 - CPU0 & System CLock frequency is switched to FRO12MHz and is NOT restored back by the API. 1 - CPU0 Interrupt Enable registers (NVIC->ISER) are modified by this function. They are restored back if DEEPPowerDown is not taken (for instance because an RTC or OSTIMER interrupt is pending). 2 - The Non Maskable Interrupt (NMI) is disabled and its configuration before calling this function will be restored back if DEEPPowerDown is not taken (for instance because an RTC or OSTIMER interrupt is pending). 3 - The HARD FAULT handler should execute from SRAM. (The Hard fault handler should initiate a full chip reset)

`void POWER_EnterSleep(void)`

Configures and enters in SLEEP low power mode.

Returns

Nothing

`void POWER_SetVoltageForFreq(uint32_t system_freq_hz)`

Power Library API to choose normal regulation and set the voltage for the desired operating frequency.

Parameters

- `system_freq_hz` – - The desired frequency (in Hertz) at which the part would like to operate, note that the voltage and flash wait states should be set before changing frequency

Returns

none

`void POWER_Xtal16mhzCapabankTrim(int32_t pi32_16MfXtalIecLoadpF_x100, int32_t pi32_16MfXtalPPcbParCappF_x100, int32_t pi32_16MfXtalNPcbParCappF_x100)`

Sets board-specific trim values for 16MHz XTAL.

Note: Following default Values can be used: `pi32_32MfXtalIecLoadpF_x100` Load capacitance, pF x 100 : 600 `pi32_32MfXtalPPcbParCappF_x100` PCB +ve parasitic capacitance, pF x 100 : 20 `pi32_32MfXtalNPcbParCappF_x100` PCB -ve parasitic capacitance, pF x 100 : 40

Parameters

- `pi32_16MfXtalIecLoadpF_x100` – Load capacitance, pF x 100. For example, 6pF becomes 600, 1.2pF becomes 120
- `pi32_16MfXtalPPcbParCappF_x100` – PCB +ve parasitic capacitance, pF x 100. For example, 6pF becomes 600, 1.2pF becomes 120
- `pi32_16MfXtalNPcbParCappF_x100` – PCB -ve parasitic capacitance, pF x 100. For example, 6pF becomes 600, 1.2pF becomes 120

Returns

none

```
void POWER_Xtal32khzCapabankTrim(int32_t pi32_32kfXtalIecLoadpF_x100, int32_t
                                pi32_32kfXtalPPcbParCappF_x100, int32_t
                                pi32_32kfXtalNPcbParCappF_x100)
```

Sets board-specific trim values for 32kHz XTAL.

Note: Following default Values can be used: pi32_32kfXtalIecLoadpF_x100 Load capacitance, pF x 100 : 600 pi32_32kfXtalPPcbParCappF_x100 PCB +ve parasitic capacitance, pF x 100 : 40 pi32_32kfXtalNPcbParCappF_x100 PCB -ve parasitic capacitance, pF x 100 : 40

Parameters

- pi32_32kfXtalIecLoadpF_x100 – Load capacitance, pF x 100. For example, 6pF becomes 600, 1.2pF becomes 120
- pi32_32kfXtalPPcbParCappF_x100 – PCB +ve parasitic capacitance, pF x 100. For example, 6pF becomes 600, 1.2pF becomes 120
- pi32_32kfXtalNPcbParCappF_x100 – PCB -ve parasitic capacitance, pF x 100. For example, 6pF becomes 600, 1.2pF becomes 120

Returns

none

```
void POWER_SetXtal16mhzLdo(void)
```

Enables and sets LDO for 16MHz XTAL.

Returns

none

```
void POWER_GetWakeUpCause(power_device_reset_cause_t *p_reset_cause,
                          power_device_boot_mode_t *p_boot_mode, uint32_t
                          *p_wakeupio_cause)
```

Return some key information related to the device reset causes / wake-up sources, for all power modes.

Parameters

- p_reset_cause – : the device reset cause, according to the definition of power_device_reset_cause_t type.
- p_boot_mode – : the device boot mode, according to the definition of power_device_boot_mode_t type.
- p_wakeupio_cause – the wake-up pin sources, according to the definition of register PMC->WAKEIOCAUSE[3:0].

Returns

Nothing

!!! IMPORTANT ERRATA - IMPORTANT ERRATA - IMPORTANT ERRATA !!!

→!

!!! valid ONLY for LPC55S69 (not for LPC55S16 and LPC55S06) !!!

!!! when FALLING EDGE DETECTION is enabled on wake-up pins: !!!

- 1. p_wakeupio_cause is NOT ACCURATE

- 2. Spurious kRESET_CAUSE_DPDRESET_WAKEUIO* event is reported when several wake-up sources are enabled during DEEP-POWER-DOWN (like enabling wake-up on RTC and Falling edge wake-up pins)

```
FSL_POWER_DRIVER_VERSION
```

power driver version 2.0.1.

LOWPOWER_SRAMRETCTRL_RETEN_RAMX0

SRAM instances retention control during low power modes.

Enable SRAMX_0 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAMX1

Enable SRAMX_1 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAMX2

Enable SRAMX_2 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAMX3

Enable SRAMX_3 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM00

Enable SRAM0_0 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM01

Enable SRAM0_1 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM10

Enable SRAM1_0 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM20

Enable SRAM2_0 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM30

Enable SRAM3_0 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM31

Enable SRAM3_1 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM40

Enable SRAM4_0 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM41

Enable SRAM4_1 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM42

Enable SRAM4_2 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM43

Enable SRAM4_3 retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM_USB_HS

Enable SRAM USB HS retention when entering in Low power modes

LOWPOWER_SRAMRETCTRL_RETEN_RAM_PUF

Enable SRAM PUFF retention when entering in Low power modes

WAKEUP_SYS

Low Power Modes Wake up sources.

WAKEUP_SDMA0

[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_GLOBALINT0

[SLEEP, DEEP SLEEP, POWER DOWN]

WAKEUP_GPIO_GLOBALINT1

[SLEEP, DEEP SLEEP, POWER DOWN]

WAKEUP_GPIO_INT0_0
[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_INT0_1
[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_INT0_2
[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_INT0_3
[SLEEP, DEEP SLEEP]

WAKEUP_UTICK
[SLEEP,]

WAKEUP_MRT
[SLEEP,]

WAKEUP_CTIMER0
[SLEEP, DEEP SLEEP]

WAKEUP_CTIMER1
[SLEEP, DEEP SLEEP]

WAKEUP_SCT
[SLEEP,]

WAKEUP_CTIMER3
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM0
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM1
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM2
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM3
[SLEEP, DEEP SLEEP, POWER DOWN]

WAKEUP_FLEXCOMM4
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM5
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM6
[SLEEP, DEEP SLEEP]

WAKEUP_FLEXCOMM7
[SLEEP, DEEP SLEEP]

WAKEUP_ADC
[SLEEP,]

WAKEUP_ACMP_CAPT
[SLEEP, DEEP SLEEP, POWER DOWN]

WAKEUP_USB0_NEEDCLK
[SLEEP, DEEP SLEEP]

WAKEUP_USB0
[SLEEP, DEEP SLEEP]

WAKEUP_RTC_LITE_ALARM_WAKEUP
[SLEEP, DEEP SLEEP, POWER DOWN, DEEP POWER DOWN]

WAKEUP_EZH_ARCH_B
[SLEEP,]

WAKEUP_WAKEUP_MAILBOX
[SLEEP, DEEP SLEEP, POWER DOWN]

WAKEUP_GPIO_INT0_4
[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_INT0_5
[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_INT0_6
[SLEEP, DEEP SLEEP]

WAKEUP_GPIO_INT0_7
[SLEEP, DEEP SLEEP]

WAKEUP_CTIMER2
[SLEEP, DEEP SLEEP]

WAKEUP_CTIMER4
[SLEEP, DEEP SLEEP]

WAKEUP_OS_EVENT_TIMER
[SLEEP, DEEP SLEEP, POWER DOWN, DEEP POWER DOWN]

WAKEUP_SDIO
[SLEEP,]

WAKEUP_USB1
[SLEEP, DEEP SLEEP]

WAKEUP_USB1_NEEDCLK
[SLEEP, DEEP SLEEP]

WAKEUP_SEC_HYPERSVISOR_CALL
[SLEEP,]

WAKEUP_SEC_GPIO_INT0_0
[SLEEP, DEEP SLEEP]

WAKEUP_SEC_GPIO_INT0_1
[SLEEP, DEEP SLEEP]

WAKEUP_PLU
[SLEEP, DEEP SLEEP]

WAKEUP_SEC_VIO

WAKEUP_SHA
[SLEEP,]

WAKEUP_CASPER
[SLEEP,]

WAKEUP_PUFF

[SLEEP,]

WAKEUP_PQ

[SLEEP,]

WAKEUP_SDMA1

[SLEEP, DEEP SLEEP]

WAKEUP_LSPI_HS

[SLEEP, DEEP SLEEP]

WAKEUP_ALLWAKEUPIOS

[, DEEP POWER DOWN]

LOWPOWER_HWWAKE_FORCED

Sleep Postpone.

Force peripheral clocking to stay on during deep-sleep mode.

LOWPOWER_HWWAKE_PERIPHERALS

Wake for Flexcomms. Any Flexcomm FIFO reaching the level specified by its own TXLVL will cause \ peripheral clocking to wake up temporarily while the related status is asserted

LOWPOWER_HWWAKE_SDMA0

Wake for DMA0. DMA0 being busy will cause peripheral clocking to remain running until DMA \ completes. Used in conjunction with LOWPOWER_HWWAKE_PERIPHERALS

LOWPOWER_HWWAKE_SDMA1

Wake for DMA1. DMA0 being busy will cause peripheral clocking to remain running until DMA \ completes. Used in conjunction with LOWPOWER_HWWAKE_PERIPHERALS

LOWPOWER_HWWAKE_ENABLE_FRO192M

Need to be set if FRO192M is disable - via PDCTRL0 - in Deep Sleep mode and any of \ LOWPOWER_HWWAKE_PERIPHERALS, LOWPOWER_HWWAKE_SDMA0 or LOWPOWER_HWWAKE_SDMA1 is set

LOWPOWER_CPURETCTRL_ENA_DISABLE

In POWER DOWN mode, CPU Retention is disabled

LOWPOWER_CPURETCTRL_ENA_ENABLE

In POWER DOWN mode, CPU Retention is enabled

LOWPOWER_WAKEUPIOSRC_PIO0_INDEX

Wake up I/O sources.

Pin P1(1)

LOWPOWER_WAKEUPIOSRC_PIO1_INDEX

Pin P0(28)

LOWPOWER_WAKEUPIOSRC_PIO2_INDEX

Pin P1(18)

LOWPOWER_WAKEUPIOSRC_PIO3_INDEX

Pin P1(30)

LOWPOWER_WAKEUPIOSRC_DISABLE

Wake up is disable

LOWPOWER_WAKEUPIOSRC_RISING

Wake up on rising edge

LOWPOWER_WAKEUPIOSRC_FALLING

Wake up on falling edge

LOWPOWER_WAKEUPIOSRC_RISING_FALLING

Wake up on both rising or falling edges

LOWPOWER_WAKEUIPIO_PIO0_PULLUPDOWN_INDEX

Wake-up I/O 0 pull-up/down configuration index

LOWPOWER_WAKEUIPIO_PIO1_PULLUPDOWN_INDEX

Wake-up I/O 1 pull-up/down configuration index

LOWPOWER_WAKEUIPIO_PIO2_PULLUPDOWN_INDEX

Wake-up I/O 2 pull-up/down configuration index

LOWPOWER_WAKEUIPIO_PIO3_PULLUPDOWN_INDEX

Wake-up I/O 3 pull-up/down configuration index

LOWPOWER_WAKEUIPIO_PIO0_PULLUPDOWN_MASK

Wake-up I/O 0 pull-up/down mask

LOWPOWER_WAKEUIPIO_PIO1_PULLUPDOWN_MASK

Wake-up I/O 1 pull-up/down mask

LOWPOWER_WAKEUIPIO_PIO2_PULLUPDOWN_MASK

Wake-up I/O 2 pull-up/down mask

LOWPOWER_WAKEUIPIO_PIO3_PULLUPDOWN_MASK

Wake-up I/O 3 pull-up/down mask

LOWPOWER_WAKEUIPIO_PULLDOWN

Select pull-down

LOWPOWER_WAKEUIPIO_PULLUP

Select pull-up

LOWPOWER_WAKEUIPIO_PIO0_DISABLEPULLUPDOWN_INDEX

Wake-up I/O 0 pull-up/down disable/enable control index

LOWPOWER_WAKEUIPIO_PIO1_DISABLEPULLUPDOWN_INDEX

Wake-up I/O 1 pull-up/down disable/enable control index

LOWPOWER_WAKEUIPIO_PIO2_DISABLEPULLUPDOWN_INDEX

Wake-up I/O 2 pull-up/down disable/enable control index

LOWPOWER_WAKEUIPIO_PIO3_DISABLEPULLUPDOWN_INDEX

Wake-up I/O 3 pull-up/down disable/enable control index

LOWPOWER_WAKEUIPIO_PIO0_DISABLEPULLUPDOWN_MASK

Wake-up I/O 0 pull-up/down disable/enable mask

LOWPOWER_WAKEUIPIO_PIO1_DISABLEPULLUPDOWN_MASK

Wake-up I/O 1 pull-up/down disable/enable mask

LOWPOWER_WAKEUIPIO_PIO2_DISABLEPULLUPDOWN_MASK

Wake-up I/O 2 pull-up/down disable/enable mask

LOWPOWER_WAKEUIPIO_PIO3_DISABLEPULLUPDOWN_MASK

Wake-up I/O 3 pull-up/down disable/enable mask

LOWPOWER_WAKEUIPIO_PIO0_USEEXTERNALPULLUPDOWN_INDEX

Wake-up I/O 0 use external pull-up/down disable/enable control index

LOWPOWER_WAKEUIO_PIO1_USEEXTERNALPULLUPDOWN_INDEX
Wake-up I/O 1 use external pull-up/down disable/enable control index

LOWPOWER_WAKEUIO_PIO2_USEEXTERNALPULLUPDOWN_INDEX
Wake-up I/O 2 use external pull-up/down disable/enable control index

LOWPOWER_WAKEUIO_PIO3_USEEXTERNALPULLUPDOWN_INDEX
Wake-up I/O 3 use external pull-up/down disable/enable control index

LOWPOWER_WAKEUIO_PIO0_USEEXTERNALPULLUPDOWN_MASK
Wake-up I/O 0 use external pull-up/down \ disable/enable mask, 0: disable, 1: enable

LOWPOWER_WAKEUIO_PIO1_USEEXTERNALPULLUPDOWN_MASK
Wake-up I/O 1 use external pull-up/down \ disable/enable mask, 0: disable, 1: enable

LOWPOWER_WAKEUIO_PIO2_USEEXTERNALPULLUPDOWN_MASK
Wake-up I/O 2 use external pull-up/down \ disable/enable mask, 0: disable, 1: enable

LOWPOWER_WAKEUIO_PIO3_USEEXTERNALPULLUPDOWN_MASK
Wake-up I/O 3 use external pull-up/down \ disable/enable mask, 0: disable, 1: enable

unsigned int fro_calib_Get_Lib_Ver(void)

void Chip_TIMER_Instance_Freq(CTIMER_Type *base, unsigned int ctimerFreq)

void USB_SOF_Event(void)

FSL_FRO_CALIB_DRIVER_VERSION
FRO_CALIB driver version 2.0.0.

FSL_COMPONENT_ID

2.40 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 $\ln(\text{val})$.

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 $\text{inv}(\text{val})$.

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 $\text{sqrt}(\text{val})$.

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 $1/\text{sqrt}(\text{val})$.

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 `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 pi. 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 `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 pi.

`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_SetCoprocesorScaler(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 =
    {
        .a_1 = xxx,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

PQ_BiquadRestoreInternalState(POWERQUAD, 0, &state);
```

(continues on next page)

(continued from previous page)

```
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,
    },
};

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 =
{
```

(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_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 =
{
    .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, &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 =
{
```

(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_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,
        .a_2 = xxx,
        .b_0 = xxx,
        .b_1 = xxx,
        .b_2 = xxx,
    },
};

```

(continues on next page)

(continued from previous page)

```
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.

2.41 PRINCE: PRINCE bus crypto engine

FSL_PRINCE_DRIVER_VERSION

PRINCE driver version 2.6.0.

Current version: 2.6.0

Change log:

- Version 2.0.0
 - Initial version.
- Version 2.1.0
 - Update for the A1 rev. of LPC55Sxx serie.
- Version 2.2.0
 - Add runtime checking of the A0 and A1 rev. of LPC55Sxx serie to support both silicone revisions.
- Version 2.3.0
 - Add support for LPC55S1x and LPC55S2x series
- Version 2.3.0
 - Fix MISRA-2012 issues.
- Version 2.3.1
 - Add support for LPC55S0x series
- Version 2.3.2
 - Fix documentation of enumeration. Extend PRINCE example.
- Version 2.4.0
 - Add support for LPC55S3x series
- Version 2.5.0
 - Add PRINCE_Config() and PRINCE_Reconfig() features.
- Version 2.5.1
 - Fix build error due to renamed symbols
- Version 2.6.0
 - Renamed CSS to ELS

enum _skboot_status

Secure status enumeration.

Values:

enumerator kStatus_SKBOOT_Success

PRINCE Success

enumerator kStatus_SKBOOT_Fail
PRINCE Fail

enumerator kStatus_SKBOOT_InvalidArgument
PRINCE Invalid argument

enumerator kStatus_SKBOOT_KeyStoreMarkerInvalid
PRINCE Invalid marker

enum __secure_bool
Secure boolean enumeration.
Values:

enumerator kSECURE_TRUE
PRINCE true

enumerator kSECURE_FALSE
PRINCE false

enum __prince_region
Prince region.
Values:

enumerator kPRINCE_Region0
PRINCE region 0

enumerator kPRINCE_Region1
PRINCE region 1

enumerator kPRINCE_Region2
PRINCE region 2

enum __prince_lock
Prince lock.
Values:

enumerator kPRINCE_Region0Lock
PRINCE region 0 lock

enumerator kPRINCE_Region1Lock
PRINCE region 1 lock

enumerator kPRINCE_Region2Lock
PRINCE region 2 lock

enumerator kPRINCE_MaskLock
PRINCE mask register lock

enum __prince_flags
Prince flag.
Values:

enumerator kPRINCE_Flag_None
PRINCE Flag None

enumerator kPRINCE_Flag_EraseCheck
PRINCE Flag Erase check

enumerator kPRINCE_Flag_WriteCheck
PRINCE Flag Write check

```
typedef enum _skboot_status skboot__status__t
```

Secure status enumeration.

```
typedef enum _secure_bool secure__bool__t
```

Secure boolean enumeration.

```
typedef enum _prince_region prince__region__t
```

Prince region.

```
typedef enum _prince_lock prince__lock__t
```

Prince lock.

```
typedef enum _prince_flags prince__flags__t
```

Prince flag.

```
static inline void PRINCE__EncryptEnable(PRINCE_Type *base)
```

Enable data encryption.

This function enables PRINCE on-the-fly data encryption.

Parameters

- base – PRINCE peripheral address.

```
static inline void PRINCE__EncryptDisable(PRINCE_Type *base)
```

Disable data encryption.

This function disables PRINCE on-the-fly data encryption.

Parameters

- base – PRINCE peripheral address.

```
static inline bool PRINCE__IsEncryptEnable(PRINCE_Type *base)
```

Is Enable data encryption.

This function test if PRINCE on-the-fly data encryption is enabled.

Parameters

- base – PRINCE peripheral address.

Returns

true if enabled, false if not

```
static inline void PRINCE__SetMask(PRINCE_Type *base, uint64_t mask)
```

Sets PRINCE data mask.

This function sets the PRINCE mask that is used to mask decrypted data.

Parameters

- base – PRINCE peripheral address.
- mask – 64-bit data mask value.

```
static inline void PRINCE__SetLock(PRINCE_Type *base, uint32_t lock)
```

Locks access for specified region registers or data mask register.

This function sets lock on specified region registers or mask register.

Parameters

- base – PRINCE peripheral address.
- lock – registers to lock. This is a logical OR of members of the enumeration `prince_lock_t`

status_t PRINCE_GenNewIV(*prince_region_t* region, *uint8_t* *iv_code, *bool* store, *flash_config_t* *flash_context)

Generate new IV code.

This function generates new IV code and stores it into the persistent memory. Ensure about 800 bytes free space on the stack when calling this routine with the store parameter set to true!

Parameters

- region – PRINCE region index.
- iv_code – IV code pointer used for storing the newly generated 52 bytes long IV code.
- store – flag to allow storing the newly generated IV code into the persistent memory (FFR).
- flash_context – pointer to the flash driver context structure.

Returns

kStatus_Success upon success

Returns

kStatus_Fail otherwise, kStatus_Fail is also returned if the key code for the particular PRINCE region is not present in the keystore (though new IV code has been provided)

status_t PRINCE_LoadIV(*prince_region_t* region, *uint8_t* *iv_code)

Load IV code.

This function enables IV code loading into the PRINCE bus encryption engine.

Parameters

- region – PRINCE region index.
- iv_code – IV code pointer used for passing the IV code.

Returns

kStatus_Success upon success

Returns

kStatus_Fail otherwise

status_t PRINCE_SetEncryptForAddressRange(*prince_region_t* region, *uint32_t* start_address, *uint32_t* length, *flash_config_t* *flash_context, *bool* regenerate_iv)

Allow encryption/decryption for specified address range.

This function sets the encryption/decryption for specified address range. The SR mask value for the selected Prince region is calculated from provided start_address and length parameters. This calculated value is OR'ed with the actual SR mask value and stored into the PRINCE SR_ENABLE register and also into the persistent memory (FFR) to be used after the device reset. It is possible to define several nonadjacent encrypted areas within one Prince region when calling this function repeatedly. If the length parameter is set to 0, the SR mask value is set to 0 and thus the encryption/decryption for the whole selected Prince region is disabled. Ensure about 800 bytes free space on the stack when calling this routine!

Parameters

- region – PRINCE region index.
- start_address – start address of the area to be encrypted/decrypted.
- length – length of the area to be encrypted/decrypted.
- flash_context – pointer to the flash driver context structure.

- `regenerate_iv` – flag to allow IV code regenerating, storing into the persistent memory (FFR) and loading into the PRINCE engine

Returns

`kStatus_Success` upon success

Returns

`kStatus_Fail` otherwise

`status_t` PRINCE_GetRegionSREnable(`PRINCE_Type` *base, *prince_region_t* region, `uint32_t` *sr_enable)

Gets the PRINCE Sub-Region Enable register.

This function gets PRINCE SR_ENABLE register.

Parameters

- base – PRINCE peripheral address.
- region – PRINCE region index.
- sr_enable – Sub-Region Enable register pointer.

Returns

`kStatus_Success` upon success

Returns

`kStatus_InvalidArgument`

`status_t` PRINCE_GetRegionBaseAddress(`PRINCE_Type` *base, *prince_region_t* region, `uint32_t` *region_base_addr)

Gets the PRINCE region base address register.

This function gets PRINCE BASE_ADDR register.

Parameters

- base – PRINCE peripheral address.
- region – PRINCE region index.
- region_base_addr – Region base address pointer.

Returns

`kStatus_Success` upon success

Returns

`kStatus_InvalidArgument`

`status_t` PRINCE_SetRegionIV(`PRINCE_Type` *base, *prince_region_t* region, `const uint8_t` iv[8])

Sets the PRINCE region IV.

This function sets specified AES IV for the given region.

Parameters

- base – PRINCE peripheral address.
- region – Selection of the PRINCE region to be configured.
- iv – 64-bit AES IV in little-endian byte order.

`status_t` PRINCE_SetRegionBaseAddress(`PRINCE_Type` *base, *prince_region_t* region, `uint32_t` region_base_addr)

Sets the PRINCE region base address.

This function configures PRINCE region base address.

Parameters

- base – PRINCE peripheral address.

- `region` – Selection of the PRINCE region to be configured.
- `region_base_addr` – Base Address for region.

`status_t` PRINCE_SetRegionSREnable(`PRINCE_Type` *`base`, `prince_region_t` `region`, `uint32_t` `sr_enable`)

Sets the PRINCE Sub-Region Enable register.

This function configures PRINCE SR_ENABLE register.

Parameters

- `base` – PRINCE peripheral address.
- `region` – Selection of the PRINCE region to be configured.
- `sr_enable` – Sub-Region Enable register value.

`status_t` PRINCE_FlashEraseWithChecker(`flash_config_t` *`config`, `uint32_t` `start`, `uint32_t` `lengthInBytes`, `uint32_t` `key`)

Erases the flash sectors encompassed by parameters passed into function.

This function erases the appropriate number of flash sectors based on the desired start address and length. It deals with the flash erase function complementary to the standard erase API of the IAP1 driver. This implementation additionally checks if the whole encrypted PRINCE subregions are erased at once to avoid secrets revealing. The checker implementation is limited to one contiguous PRINCE-controlled memory area.

Parameters

- `config` – The pointer to the flash driver context structure.
- `start` – The start address of the desired flash memory to be erased. The start address needs to be prince-sburegion-aligned.
- `lengthInBytes` – The length, given in bytes (not words or long-words) to be erased. Must be prince-sburegion-size-aligned.
- `key` – The value used to validate all flash erase APIs.

Returns

`kStatus_FLASH_Success` API was executed successfully.

Returns

`kStatus_FLASH_InvalidArgument` An invalid argument is provided.

Returns

`kStatus_FLASH_AlignmentError` The parameter is not aligned with the specified baseline.

Returns

`kStatus_FLASH_AddressError` The address is out of range.

Returns

`kStatus_FLASH_EraseKeyError` The API erase key is invalid.

Returns

`kStatus_FLASH_CommandFailure` Run-time error during the command execution.

Returns

`kStatus_FLASH_CommandNotSupported` Flash API is not supported.

Returns

`kStatus_FLASH_EccError` A correctable or uncorrectable error during command execution.

Returns

kStatus_FLASH_EncryptedRegionsEraseNotDoneAtOnce Encrypted flash subregions are not erased at once.

status_t PRINCE_FlashProgramWithChecker(*flash_config_t* *config, uint32_t start, uint8_t *src, uint32_t lengthInBytes)

Programs flash with data at locations passed in through parameters.

This function programs the flash memory with the desired data for a given flash area as determined by the start address and the length. It deals with the flash program function complementary to the standard program API of the IAP1 driver. This implementation additionally checks if the whole PRINCE subregions are programmed at once to avoid secrets revealing. The checker implementation is limited to one contiguous PRINCE-controlled memory area.

Parameters

- config – The pointer to the flash driver context structure.
- start – The start address of the desired flash memory to be programmed. Must be prince-sburegion-aligned.
- src – A pointer to the source buffer of data that is to be programmed into the flash.
- lengthInBytes – The length, given in bytes (not words or long-words), to be programmed. Must be prince-sburegion-size-aligned.

Returns

kStatus_FLASH_Success API was executed successfully.

Returns

kStatus_FLASH_InvalidArgument An invalid argument is provided.

Returns

kStatus_FLASH_AlignmentError Parameter is not aligned with the specified baseline.

Returns

kStatus_FLASH_AddressError Address is out of range.

Returns

kStatus_FLASH_AccessError Invalid instruction codes and out-of bounds addresses.

Returns

kStatus_FLASH_CommandFailure Run-time error during the command execution.

Returns

kStatus_FLASH_CommandFailure Run-time error during the command execution.

Returns

kStatus_FLASH_CommandNotSupported Flash API is not supported.

Returns

kStatus_FLASH_EccError A correctable or uncorrectable error during command execution.

Returns

kStatus_FLASH_SizeError Encrypted flash subregions are not programmed at once.

FSL_PRINCE_DRIVER_SUBREGION_SIZE_IN_KB

FSL_PRINCE_DRIVER_MAX_FLASH_ADDR

ALIGN_DOWN(x, a)

2.42 PUF: Physical Unclonable Function

FSL_PUF_DRIVER_VERSION

PUF driver version. Version 2.2.0.

Current version: 2.2.0

Change log:

- 2.0.0
 - Initial version.
- 2.0.1
 - Fixed puf_wait_usec function optimization issue.
- 2.0.2
 - Add PUF configuration structure and support for PUF SRAM controller. Remove magic constants.
- 2.0.3
 - Fix MISRA C-2012 issue.
- 2.1.0
 - Align driver with PUF SRAM controller registers on LPCXpresso55s16.
 - Update initialization logic .
- 2.1.1
 - Fix ARMGCC build warning .
- 2.1.2
 - Update: Add automatic big to little endian swap for user (pre-shared) keys destined to secret hardware bus (PUF key index 0).
- 2.1.3
 - Fix MISRA C-2012 issue.
- 2.1.4
 - Replace register uint32_t ticksCount with volatile uint32_t ticksCount in puf_wait_usec() to prevent optimization out delay loop.
- 2.1.5
 - Use common SDK delay in puf_wait_usec()
- 2.1.6
 - Changed wait time in PUF_Init(), when initialization fails it will try PUF_Powercycle() with shorter time. If this shorter time will also fail, initialization will be tried with worst case time as before.
- 2.2.0
 - Add support for kPUF_KeySlot4.
 - Add new PUF_ClearKey() function, that clears a desired PUF internal HW key register.

enum `_puf_key_index_register`

Values:

enumerator `kPUF_KeyIndex_00`

enumerator `kPUF_KeyIndex_01`

enumerator `kPUF_KeyIndex_02`

enumerator `kPUF_KeyIndex_03`

enumerator `kPUF_KeyIndex_04`

enumerator `kPUF_KeyIndex_05`

enumerator `kPUF_KeyIndex_06`

enumerator `kPUF_KeyIndex_07`

enumerator `kPUF_KeyIndex_08`

enumerator `kPUF_KeyIndex_09`

enumerator `kPUF_KeyIndex_10`

enumerator `kPUF_KeyIndex_11`

enumerator `kPUF_KeyIndex_12`

enumerator `kPUF_KeyIndex_13`

enumerator `kPUF_KeyIndex_14`

enumerator `kPUF_KeyIndex_15`

enum `_puf_min_max`

Values:

enumerator `kPUF_KeySizeMin`

enumerator `kPUF_KeySizeMax`

enumerator `kPUF_KeyIndexMax`

enum `_puf_key_slot`

PUF key slot.

Values:

enumerator `kPUF_KeySlot0`

PUF key slot 0

enumerator `kPUF_KeySlot1`

PUF key slot 1

PUF status return codes.

Values:

enumerator `kStatus_EnrollNotAllowed`

enumerator `kStatus_StartNotAllowed`

typedef enum *`_puf_key_index_register`* `puf_key_index_register_t`

```
typedef enum _puf_min_max puf_min_max_t
typedef enum _puf_key_slot puf_key_slot_t
    PUF key slot.
PUF_GET_KEY_CODE_SIZE_FOR_KEY_SIZE(x)
    Get Key Code size in bytes from key size in bytes at compile time.
PUF_MIN_KEY_CODE_SIZE
PUF_ACTIVATION_CODE_SIZE
KEystore_PUF_DISCHARGE_TIME_FIRST_TRY_MS
KEystore_PUF_DISCHARGE_TIME_MAX_MS
struct puf_config_t
    #include <fsl_puf.h>
```

2.43 Reset Driver

```
enum _SYSCON_RSTn
    Enumeration for peripheral reset control bits.
    Defines the enumeration for peripheral reset control bits in PRESETC-
    TRL/ASYNCPRESETCTRL registers
    Values:
    enumerator kROM_RST_SHIFT_RSTn
        ROM reset control
    enumerator kSRAM1_RST_SHIFT_RSTn
        SRAM1 reset control
    enumerator kSRAM2_RST_SHIFT_RSTn
        SRAM2 reset control
    enumerator kSRAM3_RST_SHIFT_RSTn
        SRAM3 reset control
    enumerator kSRAM4_RST_SHIFT_RSTn
        SRAM4 reset control
    enumerator kFLASH_RST_SHIFT_RSTn
        Flash controller reset control
    enumerator kFMC_RST_SHIFT_RSTn
        Flash accelerator reset control
    enumerator kSPIFI_RST_SHIFT_RSTn
        SPIFI reset control
    enumerator kMUX0_RST_SHIFT_RSTn
        Input mux0 reset control
    enumerator kIOCON_RST_SHIFT_RSTn
        IOCON reset control
    enumerator kGPIO0_RST_SHIFT_RSTn
        GPIO0 reset control
```

enumerator kGPIO1_RST_SHIFT_RSTn
GPIO1 reset control

enumerator kGPIO2_RST_SHIFT_RSTn
GPIO2 reset control

enumerator kGPIO3_RST_SHIFT_RSTn
GPIO3 reset control

enumerator kPINT_RST_SHIFT_RSTn
Pin interrupt (PINT) reset control

enumerator kGINT_RST_SHIFT_RSTn
Grouped interrupt (PINT) reset control.

enumerator kDMA0_RST_SHIFT_RSTn
DMA reset control

enumerator kCRC_RST_SHIFT_RSTn
CRC reset control

enumerator kWWDT_RST_SHIFT_RSTn
Watchdog timer reset control

enumerator kRTC_RST_SHIFT_RSTn
RTC reset control

enumerator kMAILBOX_RST_SHIFT_RSTn
Mailbox reset control

enumerator kADC0_RST_SHIFT_RSTn
ADC0 reset control

enumerator kMRT_RST_SHIFT_RSTn
Multi-rate timer (MRT) reset control

enumerator kOSTIMER0_RST_SHIFT_RSTn
OSTimer0 reset control

enumerator kSCT0_RST_SHIFT_RSTn
SCTimer/PWM 0 (SCT0) reset control

enumerator kSCTIPU_RST_SHIFT_RSTn
SCTIPU reset control

enumerator kUTICK_RST_SHIFT_RSTn
Micro-tick timer 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 kFC4_RST_SHIFT_RSTn
Flexcomm Interface 4 reset control

enumerator kFC5_RST_SHIFT_RSTn
Flexcomm Interface 5 reset control

enumerator kFC6_RST_SHIFT_RSTn
Flexcomm Interface 6 reset control

enumerator kFC7_RST_SHIFT_RSTn
Flexcomm Interface 7 reset control

enumerator kCTIMER2_RST_SHIFT_RSTn
CTimer 2 reset control

enumerator kUSB0D_RST_SHIFT_RSTn
USB0 Device reset control

enumerator kCTIMER0_RST_SHIFT_RSTn
CTimer 0 reset control

enumerator kCTIMER1_RST_SHIFT_RSTn
CTimer 1 reset control

enumerator kPVT_RST_SHIFT_RSTn
PVT reset control

enumerator kEZHA_RST_SHIFT_RSTn
EZHA reset control

enumerator kEZHB_RST_SHIFT_RSTn
EZHB reset control

enumerator kDMA1_RST_SHIFT_RSTn
DMA1 reset control

enumerator kCMP_RST_SHIFT_RSTn
CMP reset control

enumerator kSDIO_RST_SHIFT_RSTn
SDIO reset control

enumerator kUSB1H_RST_SHIFT_RSTn
USBHS Host reset control

enumerator kUSB1D_RST_SHIFT_RSTn
USBHS Device reset control

enumerator kUSB1RAM_RST_SHIFT_RSTn
USB RAM reset control

enumerator kUSB1_RST_SHIFT_RSTn
USBHS reset control

enumerator kFREQME_RST_SHIFT_RSTn
FREQME reset control

enumerator kGPIO4_RST_SHIFT_RSTn
GPIO4 reset control

enumerator kGPIO5_RST_SHIFT_RSTn
GPIO5 reset control

enumerator kAES_RST_SHIFT_RSTn
AES reset control

```
enumerator kOTP_RST_SHIFT_RSTn
    OTP reset control
enumerator kRNG_RST_SHIFT_RSTn
    RNG reset control
enumerator kMUX1_RST_SHIFT_RSTn
    Input mux1 reset control
enumerator kUSB0HMR_RST_SHIFT_RSTn
    USB0HMR reset control
enumerator kUSB0HSL_RST_SHIFT_RSTn
    USB0HSL reset control
enumerator kHASHCRYPT_RST_SHIFT_RSTn
    HASHCRYPT reset control
enumerator kPOWERQUAD_RST_SHIFT_RSTn
    PowerQuad reset control
enumerator kPLULUT_RST_SHIFT_RSTn
    PLU LUT reset control
enumerator kCTIMER3_RST_SHIFT_RSTn
    CTimer 3 reset control
enumerator kCTIMER4_RST_SHIFT_RSTn
    CTimer 4 reset control
enumerator kPUF_RST_SHIFT_RSTn
    PUF reset control
enumerator kCASPER_RST_SHIFT_RSTn
    CASPER reset control
enumerator kCAP0_RST_SHIFT_RSTn
    CASPER reset control
enumerator kOSTIMER1_RST_SHIFT_RSTn
    OSTIMER1 reset control
enumerator kANALOGCTL_RST_SHIFT_RSTn
    ANALOG_CTL reset control
enumerator kHSLSPI_RST_SHIFT_RSTn
    HS LSPI reset control
enumerator kGPIOSEC_RST_SHIFT_RSTn
    GPIO Secure reset control
enumerator kGPIOSECINT_RST_SHIFT_RSTn
    GPIO Secure int reset control
typedef enum _SYSCON_RSTn SYSCON_RSTn_t
    Enumeration for peripheral reset control bits.
    Defines the enumeration for peripheral reset control bits in PRESETC-
    TRL/ASYNCPRESETCTRL registers
typedef SYSCON_RSTn_t reset_ip_name_t
```

`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 2.4.0

`ADC_RSTS`

Array initializers with peripheral reset bits

`AES_RSTS`

`CRC_RSTS`

`CTIMER_RSTS`

`DMA_RSTS_N`

`FLEXCOMM_RSTS`

`GINT_RSTS`

`GPIO_RSTS_N`

`INPUTMUX_RSTS`

`IOCON_RSTS`

`FLASH_RSTS`

`MRT_RSTS`

OTP_RSTS
PINT_RSTS
RNG_RSTS
SDIO_RST
SCT_RSTS
SPIFI_RSTS
USB0D_RST
USB0HMR_RST
USB0HSL_RST
USB1H_RST
USB1D_RST
USB1RAM_RST
UTICK_RSTS
WWDT_RSTS
CAPT_RSTS_N
PLU_RSTS_N
OSTIMER_RSTS
POWERQUAD_RSTS
CASPER_RSTS
HASHCRYPT_RSTS
PUF_RSTS
USB1RAM_RSTS
USB1H_RSTS
USB1D_RSTS
USB0HSL_RSTS
USB0HMR_RSTS
USB0D_RSTS
SDIO_RSTS

2.44 RNG: Random Number Generator

FSL_RNG_DRIVER_VERSION

RNG driver version. Version 2.0.3.

Current version: 2.0.3

Change log:

- Version 2.0.0
 - Initial version
- Version 2.0.1
 - Fix MISRA C-2012 issue.
- Version 2.0.2
 - Add RESET_PeripheralReset function inside RNG_Init and RNG_Deinit functions.
- Version 2.0.3
 - Modified RNG_Init and RNG_GetRandomData functions, added rng_accumulateEntropy and rng_readEntropy functions.
 - These changes are reflecting recommended usage of RNG according to device UM.

void RNG_Init(RNG_Type *base)

Initializes the RNG.

This function initializes the RNG. When called, the RNG module and ring oscillator is enabled.

Parameters

- base – RNG base address

Returns

If successful, returns the kStatus_RNG_Success. Otherwise, it returns an error.

void RNG_Deinit(RNG_Type *base)

Shuts down the RNG.

This function shuts down the RNG.

Parameters

- base – RNG base address.

status_t RNG_GetRandomData(RNG_Type *base, void *data, size_t dataSize)

Gets random data.

This function gets random data from the RNG.

Parameters

- base – RNG base address.
- data – Pointer address used to store random data.
- dataSize – Size of the buffer pointed by the data parameter.

Returns

random data

static inline uint32_t RNG_GetRandomWord(RNG_Type *base)

Returns random 32-bit number.

This function gets random number from the RNG.

Parameters

- base – RNG base address.

Returns

random number

2.45 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.

2.46 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_STATEMSK(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);

2.47 SDIF: SD/MMC/SDIO card interface

FSL_SDIF_DRIVER_VERSION

Driver version 2.0.15.

_sdif_status SDIF status

Values:

enumerator kStatus_SDIF_DescriptorBufferLenError

Set DMA descriptor failed

enumerator kStatus_SDIF_InvalidArgument

invalid argument status

enumerator kStatus_SDIF_SyncCmdTimeout

sync command to CIU timeout status

enumerator kStatus_SDIF_SendCmdFail
send command to card fail

enumerator kStatus_SDIF_SendCmdErrorBufferFull
send command to card fail, due to command buffer full user need to resend this command

enumerator kStatus_SDIF_DMATransferFailWithFBE
DMA transfer data fail with fatal bus error , to do with this error :issue a hard reset/controller reset

enumerator kStatus_SDIF_DMATransferDescriptorUnavailable
DMA descriptor unavailable

enumerator kStatus_SDIF_DataTransferFail
transfer data fail

enumerator kStatus_SDIF_ResponseError
response error

enumerator kStatus_SDIF_DMAAddrNotAlign
DMA address not align

enumerator kStatus_SDIF_BusyTransferring
SDIF transfer busy status

enumerator kStatus_SDIF_DataTransferSuccess
transfer data success

enumerator kStatus_SDIF_SendCmdSuccess
transfer command success

_sdif_capability_flag Host controller capabilities flag mask

Values:

enumerator kSDIF_SupportHighSpeedFlag
Support high-speed

enumerator kSDIF_SupportDmaFlag
Support DMA

enumerator kSDIF_SupportSuspendResumeFlag
Support suspend/resume

enumerator kSDIF_SupportV330Flag
Support voltage 3.3V

enumerator kSDIF_Support4BitFlag
Support 4 bit mode

enumerator kSDIF_Support8BitFlag
Support 8 bit mode

_sdif_reset_type define the reset type

Values:

enumerator kSDIF_ResetController
reset controller,will reset: BIU/CIU interface CIU and state machine,ABORT_READ_DATA,SEND_IRQ_RESPONSE and READ_WAIT bits of control register,START_CMD bit of the command register

enumerator kSDIF__ResetFIFO
reset data FIFO

enumerator kSDIF__ResetDMAInterface
reset DMA interface

enumerator kSDIF__ResetAll
reset all

enum _sdif__bus__width
define the card bus width type

Values:

enumerator kSDIF__Bus1BitWidth
1bit bus width, 1bit mode and 4bit mode share one register bit

enumerator kSDIF__Bus4BitWidth
4bit mode mask

enumerator kSDIF__Bus8BitWidth
support 8 bit mode

_sdif__command__flags define the command flags

Values:

enumerator kSDIF__CmdResponseExpect
command request response

enumerator kSDIF__CmdResponseLengthLong
command response length long

enumerator kSDIF__CmdCheckResponseCRC
request check command response CRC

enumerator kSDIF__DataExpect
request data transfer,either read/write

enumerator kSDIF__DataWriteToCard
data transfer direction

enumerator kSDIF__DataStreamTransfer
data transfer mode :stream/block transfer command

enumerator kSDIF__DataTransferAutoStop
data transfer with auto stop at the end of

enumerator kSDIF__WaitPreTransferComplete
wait pre transfer complete before sending this cmd

enumerator kSDIF__TransferStopAbort
when host issue stop or abort cmd to stop data transfer ,this bit should set so that
cmd/data state-machines of CIU can return to idle correctly

enumerator kSDIF__SendInitialization
send initialization 80 clocks for SD card after power on

enumerator kSDIF__CmdUpdateClockRegisterOnly
send cmd update the CIU clock register only

enumerator kSDIF__CmdtoReadCEATADevice
host is perform read access to CE-ATA device

enumerator kSDIF_CmdExpectCCS
command expect command completion signal signal

enumerator kSDIF_BootModeEnable
this bit should only be set for mandatory boot mode

enumerator kSDIF_BootModeExpectAck
boot mode expect ack

enumerator kSDIF_BootModeDisable
when software set this bit along with START_CMD, CIU terminates the boot operation

enumerator kSDIF_BootModeAlternate
select boot mode ,alternate or mandatory

enumerator kSDIF_CmdVoltageSwitch
this bit set for CMD11 only

enumerator kSDIF_CmdDataUseHoldReg
cmd and data send to card through the HOLD register

_sdif_command_type The command type

Values:

enumerator kCARD_CommandTypeNormal
Normal command

enumerator kCARD_CommandTypeSuspend
Suspend command

enumerator kCARD_CommandTypeResume
Resume command

enumerator kCARD_CommandTypeAbort
Abort command

_sdif_response_type The command response type.

Define the command response type from card to host controller.

Values:

enumerator kCARD_ResponseTypeNone
Response type: none

enumerator kCARD_ResponseTypeR1
Response type: R1

enumerator kCARD_ResponseTypeR1b
Response type: R1b

enumerator kCARD_ResponseTypeR2
Response type: R2

enumerator kCARD_ResponseTypeR3
Response type: R3

enumerator kCARD_ResponseTypeR4
Response type: R4

enumerator kCARD_ResponseTypeR5

Response type: R5

enumerator kCARD_ResponseTypeR5b

Response type: R5b

enumerator kCARD_ResponseTypeR6

Response type: R6

enumerator kCARD_ResponseTypeR7

Response type: R7

_sdif_interrupt_mask define the interrupt mask flags

Values:

enumerator kSDIF_CardDetect

mask for card detect

enumerator kSDIF_ResponseError

command response error

enumerator kSDIF_CommandDone

command transfer over

enumerator kSDIF_DataTransferOver

data transfer over flag

enumerator kSDIF_WriteFIFORequest

write FIFO request

enumerator kSDIF_ReadFIFORequest

read FIFO request

enumerator kSDIF_ResponseCRCError

response CRC error

enumerator kSDIF_DataCRCError

data CRC error

enumerator kSDIF_ResponseTimeout

response timeout

enumerator kSDIF_DataReadTimeout

read data timeout

enumerator kSDIF_DataStarvationByHostTimeout

data starvation by host time out

enumerator kSDIF_FIFOError

indicate the FIFO under run or overrun error

enumerator kSDIF_HardwareLockError

hardware lock write error

enumerator kSDIF_DataStartBitError

start bit error

enumerator kSDIF_AutoCmdDone

indicate the auto command done

enumerator kSDIF__DataEndBitError
end bit error

enumerator kSDIF__SDIOInterrupt
interrupt from the SDIO card

enumerator kSDIF__CommandTransferStatus
command transfer status collection

enumerator kSDIF__DataTransferStatus
data transfer status collection

enumerator kSDIF__DataTransferError

enumerator kSDIF__AllInterruptStatus
all interrupt mask

`_sdif_dma_status` define the internal DMA status flags

Values:

enumerator kSDIF__DMATransFinishOneDescriptor
DMA transfer finished for one DMA descriptor

enumerator kSDIF__DMARecvFinishOneDescriptor
DMA receive finished for one DMA descriptor

enumerator kSDIF__DMAFatalBusError
DMA fatal bus error

enumerator kSDIF__DMADescriptorUnavailable
DMA descriptor unavailable

enumerator kSDIF__DMACardErrorSummary
card error summary

enumerator kSDIF__NormalInterruptSummary
normal interrupt summary

enumerator kSDIF__AbnormalInterruptSummary
abnormal interrupt summary

enumerator kSDIF__DMAAllStatus

`_sdif_dma_descriptor_flag` define the internal DMA descriptor flag

Deprecated:

Do not use this enum anymore, please use `SDIF_DMA_DESCRIPTOR_XXX_FLAG` instead.

Values:

enumerator kSDIF__DisableCompleteInterrupt
disable the complete interrupt flag for the ends in the buffer pointed to by this descriptor

enumerator kSDIF__DMADescriptorDataBufferEnd
indicate this descriptor contain the last data buffer of data

enumerator `kSDIF_DMADescriptorDataBufferStart`

indicate this descriptor contain the first data buffer of data,if first buffer size is 0,next descriptor contain the begin of the data

enumerator `kSDIF_DMASecondAddrChained`

indicate that the second addr in the descriptor is the next descriptor addr not the data buffer

enumerator `kSDIF_DMADescriptorEnd`

indicate that the descriptor list reached its final descriptor

enumerator `kSDIF_DMADescriptorOwnByDMA`

indicate the descriptor is own by SD/MMC DMA

enum `_sdif_dma_mode`

define the internal DMA mode

Values:

enumerator `kSDIF_ChainDMAMode`

enumerator `kSDIF_DualDMAMode`

typedef enum `_sdif_bus_width` `sdif_bus_width_t`

define the card bus width type

typedef enum `_sdif_dma_mode` `sdif_dma_mode_t`

define the internal DMA mode

typedef struct `_sdif_dma_descriptor` `sdif_dma_descriptor_t`

define the internal DMA descriptor

typedef struct `_sdif_dma_config` `sdif_dma_config_t`

Defines the internal DMA configure structure.

typedef struct `_sdif_data` `sdif_data_t`

Card data descriptor.

typedef struct `_sdif_command` `sdif_command_t`

Card command descriptor.

Define card command-related attribute.

typedef struct `_sdif_transfer` `sdif_transfer_t`

Transfer state.

typedef struct `_sdif_config` `sdif_config_t`

Data structure to initialize the sdif.

typedef struct `_sdif_capability` `sdif_capability_t`

SDIF capability information. Defines a structure to get the capability information of SDIF.

typedef struct `_sdif_transfer_callback` `sdif_transfer_callback_t`

sdif callback functions.

typedef struct `_sdif_handle` `sdif_handle_t`

sdif handle

Defines the structure to save the sdif state information and callback function. The detail interrupt status when send command or transfer data can be obtained from `interruptFlags` field by using mask defined in `sdif_interrupt_flag_t`;

Note: All the fields except interruptFlags and transferredWords must be allocated by the user.

typedef *status_t* (*sdif_transfer_function_t)(SDIF_Type *base, *sdif_transfer_t* *content)
sdif transfer function.

typedef struct *_sdif_host* sdif_host_t
sdif host descriptor

void SDIF_Init(SDIF_Type *base, *sdif_config_t* *config)
SDIF module initialization function.
Configures the SDIF according to the user configuration.

Parameters

- base – SDIF peripheral base address.
- config – SDIF configuration information.

void SDIF_Deinit(SDIF_Type *base)
SDIF module deinit function. user should call this function follow with IP reset.

Parameters

- base – SDIF peripheral base address.

bool SDIF_SendCardActive(SDIF_Type *base, uint32_t timeout)
SDIF send initialize 80 clocks for SD card after initial.

Parameters

- base – SDIF peripheral base address.
- timeout – timeout value

static inline void SDIF_EnableCardClock(SDIF_Type *base, bool enable)
SDIF module enable/disable card0 clock.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

static inline void SDIF_EnableCard1Clock(SDIF_Type *base, bool enable)
SDIF module enable/disable card1 clock.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

static inline void SDIF_EnableLowPowerMode(SDIF_Type *base, bool enable)
SDIF module enable/disable module disable the card clock to enter low power mode when card is idle,for SDIF cards, if interrupts must be detected, clock should not be stopped.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

static inline void SDIF_EnableCard1LowPowerMode(SDIF_Type *base, bool enable)

SDIF module enable/disable module disable the card clock to enter low power mode when card is idle, for SDIF cards, if interrupts must be detected, clock should not be stopped.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

static inline void SDIF_EnableCardPower(SDIF_Type *base, bool enable)

enable/disable the card0 power. once turn power on, software should wait for regulator/switch ramp-up time before trying to initialize card.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag.

static inline void SDIF_EnableCard1Power(SDIF_Type *base, bool enable)

enable/disable the card1 power. once turn power on, software should wait for regulator/switch ramp-up time before trying to initialize card.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag.

void SDIF_SetCardBusWidth(SDIF_Type *base, *sdif_bus_width_t* type)

set card0 data bus width

Parameters

- base – SDIF peripheral base address.
- type – data bus width type

void SDIF_SetCard1BusWidth(SDIF_Type *base, *sdif_bus_width_t* type)

set card1 data bus width

Parameters

- base – SDIF peripheral base address.
- type – data bus width type

static inline uint32_t SDIF_DetectCardInsert(SDIF_Type *base, bool data3)

SDIF module detect card0 insert status function.

Parameters

- base – SDIF peripheral base address.
- data3 – indicate use data3 as card insert detect pin

Return values

1 – card is inserted 0 card is removed

static inline uint32_t SDIF_DetectCard1Insert(SDIF_Type *base, bool data3)

SDIF module detect card1 insert status function.

Parameters

- base – SDIF peripheral base address.
- data3 – indicate use data3 as card insert detect pin

Return values

1 – card is inserted 0 card is removed

uint32_t SDIF_SetCardClock(SDIF_Type *base, uint32_t srcClock_Hz, uint32_t target_HZ)

Sets the card bus clock frequency.

Parameters

- base – SDIF peripheral base address.
- srcClock_Hz – SDIF source clock frequency united in Hz.
- target_HZ – card bus clock frequency united in Hz.

Returns

The nearest frequency of busClock_Hz configured to SD bus.

bool SDIF_Reset(SDIF_Type *base, uint32_t mask, uint32_t timeout)

reset the different block of the interface.

Parameters

- base – SDIF peripheral base address.
- mask – indicate which block to reset.
- timeout – timeout value, set to wait the bit self clear

Returns

reset result.

static inline uint32_t SDIF_GetCardWriteProtect(SDIF_Type *base)

get the card write protect status

Parameters

- base – SDIF peripheral base address.

static inline void SDIF_AssertHardwareReset(SDIF_Type *base)

toggle state on hardware reset PIN This is used which card has a reset PIN typically.

Parameters

- base – SDIF peripheral base address.

status_t SDIF_SendCommand(SDIF_Type *base, sdif_command_t *cmd, uint32_t timeout)

send command to the card

This api include polling the status of the bit START_COMMAND, if 0 used as timeout value, then this function will return directly without polling the START_CMD status.

Parameters

- base – SDIF peripheral base address.
- cmd – configuration collection
- timeout – the timeout value of polling START_CMD auto clear status.

Returns

command excute status

static inline void SDIF_EnableGlobalInterrupt(SDIF_Type *base, bool enable)

SDIF enable/disable global interrupt.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

static inline void SDIF_EnableInterrupt(SDIF_Type *base, uint32_t mask)
SDIF enable interrupt.

Parameters

- base – SDIF peripheral base address.
- mask – mask

static inline void SDIF_DisableInterrupt(SDIF_Type *base, uint32_t mask)
SDIF disable interrupt.

Parameters

- base – SDIF peripheral base address.
- mask – mask

static inline uint32_t SDIF_GetInterruptStatus(SDIF_Type *base)
SDIF get interrupt status.

Parameters

- base – SDIF peripheral base address.

static inline uint32_t SDIF_GetEnabledInterruptStatus(SDIF_Type *base)
SDIF get enabled interrupt status.

Parameters

- base – SDIF peripheral base address.

static inline void SDIF_ClearInterruptStatus(SDIF_Type *base, uint32_t mask)
SDIF clear interrupt status.

Parameters

- base – SDIF peripheral base address.
- mask – mask to clear

void SDIF_TransferCreateHandle(SDIF_Type *base, *sdif_handle_t* *handle,
sdif_transfer_callback_t *callback, void *userData)

Creates the SDIF handle. register call back function for interrupt and enable the interrupt.

Parameters

- base – SDIF peripheral base address.
- handle – SDIF handle pointer.
- callback – Structure pointer to contain all callback functions.
- userData – Callback function parameter.

static inline void SDIF_EnableDmaInterrupt(SDIF_Type *base, uint32_t mask)
SDIF enable DMA interrupt.

Parameters

- base – SDIF peripheral base address.
- mask – mask to set

static inline void SDIF_DisableDmaInterrupt(SDIF_Type *base, uint32_t mask)
SDIF disable DMA interrupt.

Parameters

- base – SDIF peripheral base address.
- mask – mask to clear

static inline uint32_t SDIF_GetInternalDMAStatus(SDIF_Type *base)

SDIF get internal DMA status.

Parameters

- base – SDIF peripheral base address.

Returns

the internal DMA status register

static inline uint32_t SDIF_GetEnabledDMAInterruptStatus(SDIF_Type *base)

SDIF get enabled internal DMA interrupt status.

Parameters

- base – SDIF peripheral base address.

Returns

the internal DMA status register

static inline void SDIF_ClearInternalDMAStatus(SDIF_Type *base, uint32_t mask)

SDIF clear internal DMA status.

Parameters

- base – SDIF peripheral base address.
- mask – mask to clear

status_t SDIF_InternalDMAConfig(SDIF_Type *base, *sdif_dma_config_t* *config, const uint32_t *data, uint32_t dataSize)

SDIF internal DMA config function.

Parameters

- base – SDIF peripheral base address.
- config – DMA configuration collection
- data – buffer pointer
- dataSize – buffer size

static inline void SDIF_EnableInternalDMA(SDIF_Type *base, bool enable)

SDIF internal DMA enable.

Parameters

- base – SDIF peripheral base address.
- enable – internal DMA enable or disable flag.

static inline void SDIF_SendReadWait(SDIF_Type *base)

SDIF send read wait to SDIF card function.

Parameters

- base – SDIF peripheral base address.

bool SDIF_AbortReadData(SDIF_Type *base, uint32_t timeout)

SDIF abort the read data when SDIF card is in suspend state Once assert this bit,data state machine will be reset which is waiting for the next blocking data,used in SDIO card suspend sequence,should call after suspend cmd send.

Parameters

- base – SDIF peripheral base address.
- timeout – timeout value to wait this bit self clear which indicate the data machine reset to idle

static inline void SDIF_EnableCEATAInterrupt(SDIF_Type *base, bool enable)

SDIF enable/disable CE-ATA card interrupt this bit should set together with the card register.

Parameters

- base – SDIF peripheral base address.
- enable – enable/disable flag

status_t SDIF_TransferNonBlocking(SDIF_Type *base, *sdif_handle_t* *handle, *sdif_dma_config_t* *dmaConfig, *sdif_transfer_t* *transfer)

SDIF transfer function data/cmd in a non-blocking way this API should be use in interrupt mode, when use this API user must call SDIF_TransferCreateHandle first, all status check through interrupt.

Parameters

- base – SDIF peripheral base address.
- handle – handle
- dmaConfig – config structure This parameter can be config as:
 - a. NULL In this condition, polling transfer mode is selected
 - b. available DMA config In this condition, DMA transfer mode is selected
- transfer – transfer configuration collection

status_t SDIF_TransferBlocking(SDIF_Type *base, *sdif_dma_config_t* *dmaConfig, *sdif_transfer_t* *transfer)

SDIF transfer function data/cmd in a blocking way.

Parameters

- base – SDIF peripheral base address.
- dmaConfig – config structure
 - a. NULL In this condition, polling transfer mode is selected
 - b. available DMA config In this condition, DMA transfer mode is selected
- transfer – transfer configuration collection

status_t SDIF_ReleaseDMADescriptor(SDIF_Type *base, *sdif_dma_config_t* *dmaConfig)

SDIF release the DMA descriptor to DMA engine this function should be called when DMA descriptor unavailable status occurs.

Parameters

- base – SDIF peripheral base address.
- dmaConfig – DMA config pointer

void SDIF_GetCapability(SDIF_Type *base, *sdif_capability_t* *capability)

SDIF return the controller capability.

Parameters

- base – SDIF peripheral base address.
- capability – capability pointer

static inline uint32_t SDIF_GetControllerStatus(SDIF_Type *base)

SDIF return the controller status.

Parameters

- base – SDIF peripheral base address.

static inline void SDIF_SendCCSD(SDIF_Type *base, bool withAutoStop)

SDIF send command complete signal disable to CE-ATA card.

Parameters

- base – SDIF peripheral base address.
- withAutoStop – auto stop flag

void SDIF_ConfigClockDelay(uint32_t target_HZ, uint32_t divider)

SDIF config the clock delay This function is used to config the cclk_in delay to sample and driver the data ,should meet the min setup time and hold time, and user need to config this parameter according to your board setting.

Parameters

- target_HZ – freq work mode
- divider – not used in this function anymore, use DELAY value instead of phase directly.

SDIF_CLOCK_RANGE_NEED_DELAY

SDIOCLKCTRL setting Below clock delay setting should depend on specific platform, so it can be redefined when timing mismatch issue occur. Such as: response error/CRC error and so on.

clock range value which need to add delay to avoid timing issue

SDIF_HIGHSPEED_SAMPLE_DELAY

High speed mode clk_sample fixed delay.

12 * 250ps = 3ns

SDIF_HIGHSPEED_DRV_DELAY

High speed mode clk_drv fixed delay.

31 * 250ps = 7.75ns

SDIF_HIGHSPEED_SAMPLE_PHASE_SHIFT

High speed mode clk_sample phase shift.

SDIF_HIGHSPEED_DRV_PHASE_SHIFT

High speed mode clk_drv phase shift.

SDIF_DEFAULT_MODE_SAMPLE_DELAY

default mode sample fixed delay

12 * 250ps = 3ns

SDIF_DEFAULT_MODE_DRV_DELAY

31 * 250ps = 7.75ns

SDIF_INTERNAL_DMA_ADDR_ALIGN

SDIF internal DMA descriptor address and the data buffer address align.

SDIF_DMA_DESCRIPTOR_DISABLE_COMPLETE_INT_FLAG

SDIF DMA descriptor flag.

SDIF_DMA_DESCRIPTOR_DATA_BUFFER_END_FLAG

SDIF_DMA_DESCRIPTOR_DATA_BUFFER_START_FLAG

SDIF_DMA_DESCRIPTOR_SECOND_ADDR_CHAIN_FLAG

SDIF_DMA_DESCRIPTOR_DESCRIPTOR_END_FLAG

SDIF_DMA_DESCRIPTOR_OWN_BY_DMA_FLAG

struct _sdif_dma_descriptor

#include <fsl_sdif.h> define the internal DMA descriptor

Public Members

uint32_t dmaDesAttribute

internal DMA attribute control and status

uint32_t dmaDataBufferSize

internal DMA transfer buffer size control

const uint32_t *dmaDataBufferAddr0

internal DMA buffer 0 addr ,the buffer size must be 32bit aligned

const uint32_t *dmaDataBufferAddr1

internal DMA buffer 1 addr ,the buffer size must be 32bit aligned

struct _sdif_dma_config

#include <fsl_sdif.h> Defines the internal DMA configure structure.

Public Members

bool enableFixBurstLen

fix burst len enable/disable flag,When set, the AHB will use only SINGLE, INCR4, INCR8 or INCR16 during start of normal burst transfers. When reset, the AHB will use SINGLE and INCR burst transfer operations

sdif_dma_mode_t mode

define the DMA mode

uint8_t dmaDesSkipLen

define the descriptor skip length ,the length between two descriptor this field is special for dual DMA mode

uint32_t *dmaDesBufferStartAddr

internal DMA descriptor start address

uint32_t dmaDesBufferLen

internal DMA buffer descriptor buffer len ,user need to pay attention to the dma descriptor buffer length if it is bigger enough for your transfer

struct _sdif_data

#include <fsl_sdif.h> Card data descriptor.

Public Members

bool streamTransfer

indicate this is a stream data transfer command

bool enableAutoCommand12

indicate if auto stop will send when data transfer over

bool enableIgnoreError

indicate if enable ignore error when transfer data

```
size_t blockSize
    Block size, take care when configure this parameter
uint32_t blockCount
    Block count
uint32_t *rxData
    data buffer to receive
const uint32_t *txData
    data buffer to transfer
struct __sdif_command
    #include <fsl_sdif.h> Card command descriptor.
    Define card command-related attribute.

Public Members

uint32_t index
    Command index
uint32_t argument
    Command argument
uint32_t response[4U]
    Response for this command
uint32_t type
    define the command type
uint32_t responseType
    Command response type
uint32_t flags
    Cmd flags
uint32_t responseErrorFlags
    response error flags, need to check the flags when receive the cmd response
struct __sdif_transfer
    #include <fsl_sdif.h> Transfer state.
```

Public Members

```
sdif_data_t *data
    Data to transfer
sdif_command_t *command
    Command to send
struct __sdif_config
    #include <fsl_sdif.h> Data structure to initialize the sdif.
```

Public Members

```
uint8_t responseTimeout
    command response timeout value
```

uint32_t cardDetDebounce_Clock

define the debounce clock count which will used in card detect logic,typical value is 5-25ms

uint32_t dataTimeout

data timeout value

struct _sdif_capability

#include <fsl_sdif.h> SDIF capability information. Defines a structure to get the capability information of SDIF.

Public Members

uint32_t sdVersion

support SD card/sdio version

uint32_t mmcVersion

support emmc card version

uint32_t maxBlockLength

Maximum block length united as byte

uint32_t maxBlockCount

Maximum byte count can be transfered

uint32_t flags

Capability flags to indicate the support information

struct _sdif_transfer_callback

#include <fsl_sdif.h> sdif callback functions.

Public Members

void (*cardInserted)(SDIF_Type *base, void *userData)

card insert call back

void (*cardRemoved)(SDIF_Type *base, void *userData)

card remove call back

void (*SDIOInterrupt)(SDIF_Type *base, void *userData)

SDIO card interrupt occurs

void (*DMADesUnavailable)(SDIF_Type *base, void *userData)

DMA descriptor unavailable

void (*CommandReload)(SDIF_Type *base, void *userData)

command buffer full,need re-load

void (*TransferComplete)(SDIF_Type *base, void *handle, *status_t* status, void *userData)

Transfer complete callback

struct _sdif_handle

#include <fsl_sdif.h> sdif handle

Defines the structure to save the sdif state information and callback function. The detail interrupt status when send command or transfer data can be obtained from interruptFlags field by using mask defined in sdif_interrupt_flag_t;

Note: All the fields except interruptFlags and transferredWords must be allocated by the user.

Public Members*sdif_data_t* *volatile data

Data to transfer

sdif_command_t *volatile command

Command to send

volatile uint32_t transferredWords

Words transferred by polling way

sdif_transfer_callback_t callback

Callback function

void *userData

Parameter for transfer complete callback

struct _sdif_host

#include <fsl_sdif.h> sdif host descriptor

Public Members

SDIF_Type *base

sdif peripheral base address

uint32_t sourceClock_Hz

sdif source clock frequency united in Hz

sdif_config_t config

sdif configuration

sdif_transfer_function_t transfer

sdif transfer function

sdif_capability_t capability

sdif capability information

2.48 skboot_authenticate

enum _skboot_status

SKBOOT return status.

Values:

enumerator kStatus_SKBOOT_Success

SKBOOT return success status.

enumerator kStatus_SKBOOT_Fail

SKBOOT return fail status.

enumerator kStatus_SKBOOT_InvalidArgument

SKBOOT return invalid argument status.

enumerator kStatus_SKBOOT_KeyStoreMarkerInvalid

SKBOOT return Keystore invalid Marker status.

enumerator kStatus_SKBOOT_HashcryptFinishedWithStatusSuccess

SKBOOT return Hashcrypt finished with the success status.

enumerator kStatus_SKBOOT_HashcryptFinishedWithStatusFail
SKBOOT return Hashcrypt finished with the fail status.

enum *_secure_bool*

Secure bool flag.

Values:

enumerator kSECURE_TRUE
Secure true flag.

enumerator kSECURE_FALSE
Secure false flag.

enumerator kSECURE_CALLPROTECT_SECURITY_FLAGS
Secure call protect the security flag.

enumerator kSECURE_CALLPROTECT_IS_APP_READY
Secure call protect the app is ready flag.

enumerator kSECURE_TRACKER_VERIFIED
Secure tracker verified flag.

typedef enum *_skboot_status* skboot_status_t
SKBOOT return status.

typedef enum *_secure_bool* secure_bool_t
Secure bool flag.

skboot_status_t skboot_authenticate(const uint8_t *imageStartAddr, *secure_bool_t* *isSignVerified)

Authenticate entry function with ARENA allocator init.

This is called by ROM boot or by ROM API *g_skbootAuthenticateInterface*

void HASH_IRQHandler(void)

Interface for image authentication API.

2.49 SPI: Serial Peripheral Interface Driver

2.50 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

2.51 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 maximum 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

2.52 SYSCTL: I2S bridging and signal sharing Configuration

void SYSCTL_Init(SYSCTL_Type *base)

SYSCTL initial.

Parameters

- base – Base address of the SYSCTL peripheral.

void SYSCTL_Deinit(SYSCTL_Type *base)

SYSCTL deinit.

Parameters

- base – Base address of the SYSCTL peripheral.

void SYSCTL_SetFlexcommShareSet(SYSCTL_Type *base, uint32_t flexCommIndex, uint32_t sckSet, uint32_t wsSet, uint32_t dataInSet, uint32_t dataOutSet)

SYSCTL share set configure for flexcomm.

Parameters

- base – Base address of the SYSCTL peripheral.
- flexCommIndex – index of flexcomm, reference `_sysctl_share_src`
- sckSet – share set for sck, reference `_sysctl_share_set_index`
- wsSet – share set for ws, reference `_sysctl_share_set_index`

- dataInSet – share set for data in, reference `_sysctl_share_set_index`
- dataOutSet – share set for data out, reference `_sysctl_dataout_mask`

```
void SYSCTL_SetShareSet(SYSCTL_Type *base, uint32_t flexCommIndex,  
                        sysctl_fcctrlsel_signal_t signal, uint32_t set)
```

SYSCTL share set configure for separate signal.

Parameters

- base – Base address of the SYSCTL peripheral
- flexCommIndex – index of flexcomm,reference `_sysctl_share_src`
- signal – FCCTRLSEL signal shift
- set – share set for sck, reference `_sysctl_share_set_index`

```
void SYSCTL_SetShareSetSrc(SYSCTL_Type *base, uint32_t setIndex, uint32_t sckShareSrc,  
                           uint32_t wsShareSrc, uint32_t dataInShareSrc, uint32_t  
                           dataOutShareSrc)
```

SYSCTL share set source configure.

Parameters

- base – Base address of the SYSCTL peripheral
- setIndex – index of share set, reference `_sysctl_share_set_index`
- sckShareSrc – sck source for this share set,reference `_sysctl_share_src`
- wsShareSrc – ws source for this share set,reference `_sysctl_share_src`
- dataInShareSrc – data in source for this share set,reference `_sysctl_share_src`
- dataOutShareSrc – data out source for this share set,reference `_sysctl_dataout_mask`

```
void SYSCTL_SetShareSignalSrc(SYSCTL_Type *base, uint32_t setIndex,  
                              sysctl_sharedctrlset_signal_t signal, uint32_t shareSrc)
```

SYSCTL sck source configure.

Parameters

- base – Base address of the SYSCTL peripheral
- setIndex – index of share set, reference `_sysctl_share_set_index`
- signal – FCCTRLSEL signal shift
- shareSrc – sck source fro this share set,reference `_sysctl_share_src`

FSL_SYSCTL_DRIVER_VERSION

Group sysctl driver version for SDK.

Version 2.0.5.

enum `_sysctl_share_set_index`

SYSCTL share set.

Values:

enumerator `kSYSCTL_ShareSet0`
share set 0

enumerator `kSYSCTL_ShareSet1`
share set 1

enum _sysctl_fcctrlsel_signal
SYSCTL flexcomm signal.

Values:

enumerator kSYSCTL_FlexcommSignalSCK
SCK signal

enumerator kSYSCTL_FlexcommSignalWS
WS signal

enumerator kSYSCTL_FlexcommSignalDataIn
Data in signal

enumerator kSYSCTL_FlexcommSignalDataOut
Data out signal

enum _sysctl_share_src
SYSCTL flexcomm index.

Values:

enumerator kSYSCTL_Flexcomm0
share set 0

enumerator kSYSCTL_Flexcomm1
share set 1

enumerator kSYSCTL_Flexcomm2
share set 2

enumerator kSYSCTL_Flexcomm4
share set 4

enumerator kSYSCTL_Flexcomm5
share set 5

enumerator kSYSCTL_Flexcomm6
share set 6

enumerator kSYSCTL_Flexcomm7
share set 7

enum _sysctl_dataout_mask
SYSCTL shared data out mask.

Values:

enumerator kSYSCTL_Flexcomm0DataOut
share set 0

enumerator kSYSCTL_Flexcomm1DataOut
share set 1

enumerator kSYSCTL_Flexcomm2DataOut
share set 2

enumerator kSYSCTL_Flexcomm4DataOut
share set 4

enumerator kSYSCTL_Flexcomm5DataOut
share set 5

```
enumerator kSYSCTL_Flexcomm6DataOut
    share set 6
enumerator kSYSCTL_Flexcomm7DataOut
    share set 7
enum _sysctl_sharedctrlset_signal
    SYSCTL flexcomm signal.
Values:
enumerator kSYSCTL_SharedCtrlSignalSCK
    SCK signal
enumerator kSYSCTL_SharedCtrlSignalWS
    WS signal
enumerator kSYSCTL_SharedCtrlSignalDataIn
    Data in signal
enumerator kSYSCTL_SharedCtrlSignalDataOut
    Data out signal
typedef enum _sysctl_fcctrlsel_signal sysctl_fcctrlsel_signal_t
    SYSCTL flexcomm signal.
typedef enum _sysctl_sharedctrlset_signal sysctl_sharedctrlset_signal_t
    SYSCTL flexcomm signal.
```

2.53 USART: Universal Synchronous/Asynchronous Receiver/Transmitter Driver

2.54 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

2.55 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_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 `remainBytes` 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_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_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_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_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.

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 __unnamed32__`

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.

2.56 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.

2.57 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 = 0xFFFFFfU;
config->timeoutValue = 0xFFFFFfU;
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 `0xFFFFF` (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 2.2.0

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

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

- [Zephyr](#)
- [Apache Mynewt](#)
- [Apache NuttX](#)
- [RIOT](#)
- [Mbed OS](#)
- [Espressif](#)
- [Cypress/Infineon](#)

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

MCUboot How-tos

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

- [Zephyr](#)
- [Apache Mynewt](#)
- [Apache NuttX](#)
- [RIOT](#)
- [Mbed OS](#)
- [Espressif](#)
- [Cypress/Infineon](#)

There are also instructions for the *Simulator*.

Roadmap

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

Source files

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

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

Joining the project

Developers are welcome!

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

- [Our developer mailing list](#)
- [Our Discord channel](#) [Get your invite](#)

3.2 File System

3.2.1 FatFs

MCUXpresso SDK : mcuxsdk-middleware-fatfs

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

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

Visit [FatFs - Documentation](#) to review details on the contents in this sub-repo.

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

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

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

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

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

Changelog FatFs

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

The format is based on [Keep a Changelog](#)

[R0.15_rev0]

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

[R0.14b_rev1]

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

[R0.14b_rev0]

- Upgraded to version 0.14b

[R0.14a_rev0]

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

[R0.14_rev0]

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

[R0.13c_rev0]

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

[R0.13b_rev0]

- Upgraded to version 0.13b

[R0.13a_rev0]

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

[R0.12c_rev1]

- Add NAND disk support.

[R0.12c_rev0]

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

[R0.12b_rev0]

- Upgraded to version 0.12b.

[R0.11a]

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

3.3 Motor Control

3.3.1 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.

3.4 MultiCore

3.4.1 Multicore SDK

Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

Multicore SDK (MCSDK) Release Notes

Overview These are the release notes for the NXP Multicore Software Development Kit (MCSDK) version 25.06.00.

This software package contains components for efficient work with multicore devices as well as for the multiprocessor communication.

What is new

- eRPC [CHANGELOG](#)
- RPPMsg-Lite [CHANGELOG](#)
- MCMgr [CHANGELOG](#)
- Supported evaluation boards (multicore examples):
 - LPCXpresso55S69
 - FRDM-K32L3A6
 - MIMXRT1170-EVKB
 - MIMXRT1160-EVK
 - MIMXRT1180-EVK
 - MCX-N5XX-EVK
 - MCX-N9XX-EVK
 - FRDM-MCXN947
 - MIMXRT700-EVK
 - KW47-EVK
 - KW47-LOC
 - FRDM-MCXW72
 - MCX-W72-EVK
- Supported evaluation boards (multiprocessor examples):
 - LPCXpresso55S36
 - FRDM-K22F
 - FRDM-K32L2B
 - MIMXRT685-EVK
 - MIMXRT1170-EVKB
 - MIMXRT1180
 - FRDM-MCXN236
 - FRDM-MCXC242
 - FRDM-MCXC444
 - MCX-N9XX-EVK
 - FRDM-MCXN947
 - MIMXRT700-EVK

Development tools The Multicore SDK (MCSDK) was compiled and tested with development tools referred in: [Development tools](#)

Release contents This table describes the release contents. Not all MCUXpresso SDK packages contain the whole set of these components.

Deliverable	Location
Multicore SDK location	<MCUXpressoSDK_install_dir>/middleware/multicore/
Documentation	<MCSDK_dir>/mcuxsdk-doc/
Embedded Remote Procedure Call component	<MCSDK_dir>/erpc/
Multicore Manager component	<MCSDK_dir>/mcmgr/
RPMsg-Lite	<MCSDK_dir>/rpmsg_lite/
Multicore demo applications	<MCUXpressoSDK_install_dir>/examples/multicore_examples/
Multiprocessor demo applications	<MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/

Multicore SDK release overview Together, the Multicore SDK (MCSDK) and the MCUXpresso SDK (SDK) form a framework for the development of software for NXP multicore devices. The MCSDK release consists of the following elementary software components for multicore:

- Embedded Remote Procedure Call (eRPC)
- Multicore Manager (MCMGR) - included just in SDK for multicore devices
- Remote Processor Messaging - Lite (RPMsg-Lite) - included just in SDK for multicore devices

The MCSDK is also accompanied with documentation and several multicore and multiprocessor demo applications.

Demo applications The multicore demo applications demonstrate the usage of the MCSDK software components on supported multicore development boards.

The following multicore demo applications are located together with other MCUXpresso SDK examples in

the <MCUXpressoSDK_install_dir>/examples/multicore_examples subdirectories.

- erpc_matrix_multiply_mu
- erpc_matrix_multiply_mu_rtos
- erpc_matrix_multiply_rpmsg
- erpc_matrix_multiply_rpmsg_rtos
- erpc_two_way_rpc_rpmsg_rtos
- freertos_message_buffers
- hello_world
- multicore_manager
- rpmsg_lite_pingpong
- rpmsg_lite_pingpong_rtos
- rpmsg_lite_pingpong_tzm

The eRPC multicore component can be leveraged for inter-processor communication and remote procedure calls between SoCs / development boards.

The following multiprocessor demo applications are located together with other MCUXpresso SDK examples in

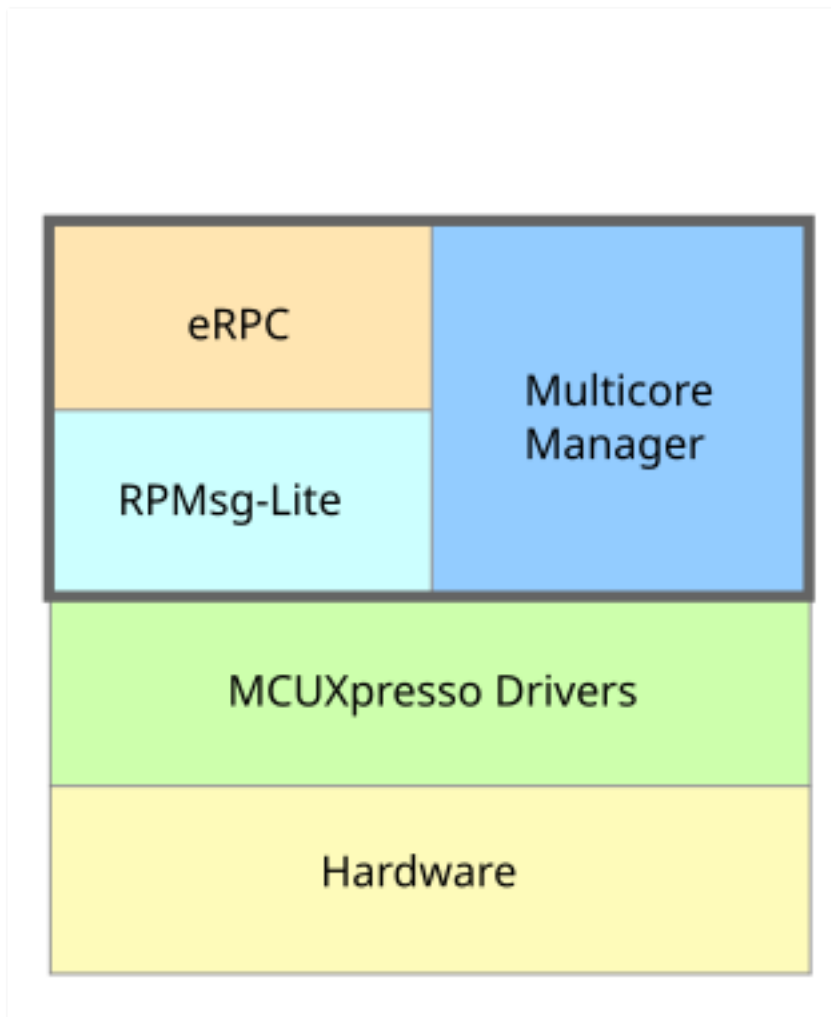
the <MCUXpressoSDK_install_dir>/examples/multiprocessor_examples subdirectories.

- erpc_client_matrix_multiply_spi
- erpc_server_matrix_multiply_spi
- erpc_client_matrix_multiply_uart
- erpc_server_matrix_multiply_uart
- erpc_server_dac_adc
- erpc_remote_control

Getting Started with Multicore SDK (MCSDK)

Overview Multicore Software Development Kit (MCSDK) is a Software Development Kit that provides comprehensive software support for NXP dual/multicore devices. The MCSDK is combined with the MCUXpresso SDK to make the software framework for easy development of multicore applications.

The following figure highlights the layers and main software components of the MCSDK.

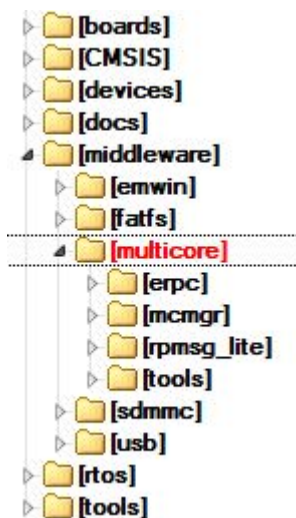


All the MCSDK-related files are located in `<MCUXpressoSDK_install_dir>/middleware/multicore` folder.

For supported toolchain versions, see the *Multicore SDK v25.06.00 Release Notes* (document MCS-DKRN). For the latest version of this and other MCSDK documents, visit www.nxp.com.

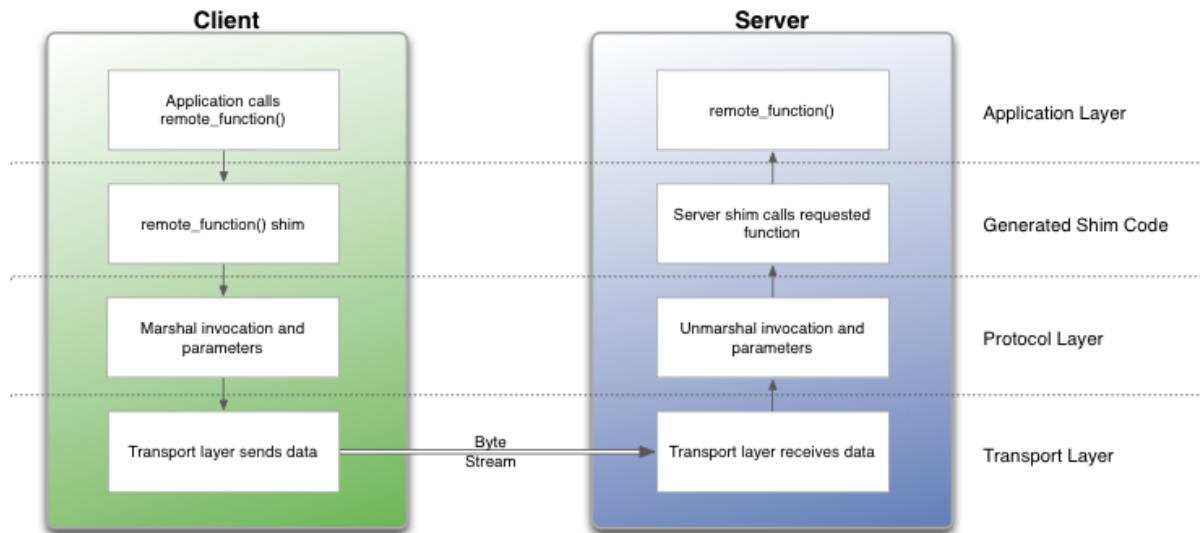
Multicore SDK (MCSDK) components The MCSDK consists of the following software components:

- **Embedded Remote Procedure Call (eRPC):** This component is a combination of a library and code generator tool that implements a transparent function call interface to remote services (running on a different core).
- **Multicore Manager (MCMGR):** This library maintains information about all cores and starts up secondary/auxiliary cores.
- **Remote Processor Messaging - Lite (RPMsg-Lite):** Inter-Processor Communication library.



Embedded Remote Procedure Call (eRPC) The Embedded Remote Procedure Call (eRPC) is the RPC system created by NXP. The RPC is a mechanism used to invoke a software routine on a remote system via a simple local function call.

When a remote function is called by the client, the function's parameters and an identifier for the called routine are marshaled (or serialized) into a stream of bytes. This byte stream is transported to the server through a communications channel (IPC, TPC/IP, UART, and so on). The server unmarshals the parameters, determines which function was invoked, and calls it. If the function returns a value, it is marshaled and sent back to the client.



RPC implementations typically use a combination of a tool (erpcgen) and IDL (interface definition language) file to generate source code to handle the details of marshaling a function's parameters and building the data stream.

Main eRPC features:

- Scalable from BareMetal to Linux OS - configurable memory and threading policies.
- Focus on embedded systems - intrinsic support for C, modular, and lightweight implementation.
- Abstracted transport interface - RPMsg is the primary transport for multicore, UART, or SPI-based solutions can be used for multichip.

The eRPC library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc` folder. For detailed information about the eRPC, see the documentation available in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc` folder.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems.

The main MCMGR features:

- Maintains information about all cores in system.
- Secondary/auxiliary cores startup and shutdown.
- Remote core monitoring and event handling.

The MCMGR library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr` folder. For detailed information about the MCMGR library, see the documentation available in the `<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/doc` folder.

Remote Processor Messaging Lite (RPMsg-Lite) RPMsg-Lite is a lightweight implementation of the RPMsg protocol. The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system. Compared to the legacy OpenAMP implementation, RPMsg-Lite offers a code size reduction, API simplification, and improved modularity.

The main RPMsg protocol features:

- Shared memory interprocessor communication.
- Virtio-based messaging bus.
- Application-defined messages sent between endpoints.

- Portable to different environments/platforms.
- Available in upstream Linux OS.

The RPMsg-Lite library is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg-lite` folder. For detailed information about the RPMsg-Lite, see the RPMsg-Lite User's Guide located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/doc` folder.

MCSDK demo applications Multicore and multiprocessor example applications are stored together with other MCUXpresso SDK examples, in the dedicated multicore subfolder.

Location		Folder
Multicore projects	example	<code><MCUXpressoSDK_install_dir>/examples/multicore_examples/<application_name>/</code>
Multiprocessor projects	example	<code><MCUXpressoSDK_install_dir>/examples/multiprocessor_examples/<application_name>/</code>

See the *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) and *Getting Started with MCUXpresso SDK for XXX Derivatives* documents for more information about the MCUXpresso SDK example folder structure and the location of individual files that form the example application projects. These documents also contain information about building, running, and debugging multicore demo applications in individual supported IDEs. Each example application also contains a readme file that describes the operation of the example and required setup steps.

Inter-Processor Communication (IPC) levels The MCSDK provides several mechanisms for Inter-Processor Communication (IPC). Particular ways and levels of IPC are described in this chapter.

IPC using low-level drivers

The NXP multicore SoCs are equipped with peripheral modules dedicated for data exchange between individual cores. They deal with the Mailbox peripheral for LPC parts and the Messaging Unit (MU) peripheral for Kinetis and i.MX parts. The common attribute of both modules is the ability to provide a means of IPC, allowing multiple CPUs to share resources and communicate with each other in a simple manner.

The most lightweight method of IPC uses the MCUXpresso SDK low-level drivers for these peripherals. Using the Mailbox/MU driver API functions, it is possible to pass a value from core to core via the dedicated registers (could be a scalar or a pointer to shared memory) and also to trigger inter-core interrupts for notifications.

For details about individual driver API functions, see the MCUXpresso SDK API Reference Manual of the specific multicore device. The MCUXpresso SDK is accompanied with the RPMsg-Lite documentation that shows how to use this API in multicore applications.

Messaging mechanism

On top of Mailbox/MU drivers, a messaging system can be implemented, allowing messages to send between multiple endpoints created on each of the CPUs. The RPMsg-Lite library of the MCSDK provides this ability and serves as the preferred MCUXpresso SDK messaging library. It implements ring buffers in shared memory for messages exchange without the need of a locking mechanism.

The RPMsg-Lite provides the abstraction layer and can be easily ported to different multicore platforms and environments (Operating Systems). The advantages of such a messaging system are ease of use (there is no need to study behavior of the used underlying hardware) and smooth application code portability between platforms due to unified messaging API.

However, this costs several kB of code and data memory. The MCUXpresso SDK is accompanied by the RPMsg-Lite documentation and several multicore examples. You can also obtain the latest RPMsg-Lite code from the GitHub account github.com/nxp-mcuxpresso/rpmsg-lite.

Remote procedure calls

To facilitate the IPC even more and to allow the remote functions invocation, the remote procedure call mechanism can be implemented. The eRPC of the MCSDK serves for these purposes and allows the ability to invoke a software routine on a remote system via a simple local function call. Utilizing different transport layers, it is possible to communicate between individual cores of multicore SoCs (via RPMsg-Lite) or between separate processors (via SPI, UART, or TCP/IP). The eRPC is mostly applicable to the MPU parts with enough of memory resources like i.MX parts.

The eRPC library allows you to export existing C functions without having to change their prototypes (in most cases). It is accompanied by the code generator tool that generates the shim code for serialization and invocation based on the IDL file with definitions of data types and remote interfaces (API).

If the communicating peer is running as a Linux OS user-space application, the generated code can be either in C/C++ or Python.

Using the eRPC simplifies the access to services implemented on individual cores. This way, the following types of applications running on dedicated cores can be easily interfaced:

- Communication stacks (USB, Thread, Bluetooth Low Energy, Zigbee)
- Sensor aggregation/fusion applications
- Encryption algorithms
- Virtual peripherals

The eRPC is publicly available from the following GitHub account: github.com/EmbeddedRPC/erpc. Also, the MCUXpresso SDK is accompanied by the eRPC code and several multicore and multiprocessor eRPC examples.

The mentioned IPC levels demonstrate the scalability of the Multicore SDK library. Based on application needs, different IPC techniques can be used. It depends on the complexity, required speed, memory resources, system design, and so on. The MCSDK brings users the possibility for quick and easy development of multicore and multiprocessor applications.

Changelog Multicore SDK

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

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[25.06.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.14.0
 - eRPC generator (erpcgen) v1.14.0
 - Multicore Manager (MCMgr) v5.0.0
 - RPMsg-Lite v5.2.0

[25.03.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0

- eRPC generator (erpcgen) v1.13.0
- Multicore Manager (MCMgr) v4.1.7
- RPPMsg-Lite v5.1.4

[24.12.00]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.6
 - RPPMsg-Lite v5.1.3

[2.16.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.13.0
 - eRPC generator (erpcgen) v1.13.0
 - Multicore Manager (MCMgr) v4.1.5
 - RPPMsg-Lite v5.1.2

[2.15.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.12.0
 - eRPC generator (erpcgen) v1.12.0
 - Multicore Manager (MCMgr) v4.1.5
 - RPPMsg-Lite v5.1.1

[2.14.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.11.0
 - eRPC generator (erpcgen) v1.11.0
 - Multicore Manager (MCMgr) v4.1.4
 - RPPMsg-Lite v5.1.0

[2.13.0_imxrt1180a0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.10.0
 - eRPC generator (erpcgen) v1.10.0
 - Multicore Manager (MCMgr) v4.1.3
 - RPPMsg-Lite v5.0.0

[2.13.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.10.0
 - eRPC generator (erpcgen) v1.10.0
 - Multicore Manager (MCMgr) v4.1.3
 - RPSMsg-Lite v5.0.0

[2.12.0_imx93]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.1
 - eRPC generator (erpcgen) v1.9.1
 - Multicore Manager (MCMgr) v4.1.2
 - RPSMsg-Lite v4.0.1

[2.12.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.1
 - eRPC generator (erpcgen) v1.9.1
 - Multicore Manager (MCMgr) v4.1.2
 - RPSMsg-Lite v4.0.0

[2.11.1]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.0
 - eRPC generator (erpcgen) v1.9.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.2.1

[2.11.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.9.0
 - eRPC generator (erpcgen) v1.9.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.2.0

[2.10.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.8.1
 - eRPC generator (erpcgen) v1.8.1
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.1.2

[2.9.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.8.0
 - eRPC generator (erpcgen) v1.8.0
 - Multicore Manager (MCMgr) v4.1.1
 - RPSMsg-Lite v3.1.1

[2.8.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.4
 - eRPC generator (erpcgen) v1.7.4
 - Multicore Manager (MCMgr) v4.1.0
 - RPSMsg-Lite v3.1.0

[2.7.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.3
 - eRPC generator (erpcgen) v1.7.3
 - Multicore Manager (MCMgr) v4.1.0
 - RPSMsg-Lite v3.0.0

[2.6.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.2
 - eRPC generator (erpcgen) v1.7.2
 - Multicore Manager (MCMgr) v4.0.3
 - RPSMsg-Lite v2.2.0

[2.5.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.1
 - eRPC generator (erpcgen) v1.7.1
 - Multicore Manager (MCMgr) v4.0.2
 - RPMsg-Lite v2.0.2

[2.4.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.7.0
 - eRPC generator (erpcgen) v1.7.0
 - Multicore Manager (MCMgr) v4.0.1
 - RPMsg-Lite v2.0.1

[2.3.1]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.6.0
 - eRPC generator (erpcgen) v1.6.0
 - Multicore Manager (MCMgr) v4.0.0
 - RPMsg-Lite v1.2.0

[2.3.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.5.0
 - eRPC generator (erpcgen) v1.5.0
 - Multicore Manager (MCMgr) v3.0.0
 - RPMsg-Lite v1.2.0

[2.2.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.4.0
 - eRPC generator (erpcgen) v1.4.0
 - Multicore Manager (MCMgr) v2.0.1
 - RPMsg-Lite v1.1.0

[2.1.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.3.0
 - eRPC generator (erpcgen) v1.3.0

[2.0.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.2.0
 - eRPC generator (erpcgen) v1.2.0
 - Multicore Manager (MCMgr) v2.0.0
 - RPMMsg-Lite v1.0.0

[1.1.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.1.0
 - Multicore Manager (MCMgr) v1.1.0
 - Open-AMP / RPMMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev01

[1.0.0]

- Multicore SDK component versions:
 - embedded Remote Procedure Call (eRPC) v1.0.0
 - Multicore Manager (MCMgr) v1.0.0
 - Open-AMP / RPMMsg based on SHA1 ID 44b5f3c0a6458f3cf80 rev00

Multicore SDK Components

RPMSG-Lite

MCUXpresso SDK : mcuxsdk-middleware-rpmsg-lite

Overview This repository is for MCUXpresso SDK RPMSG-Lite middleware delivery and it contains RPMSG-Lite component 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](#) for the complete delivery of MCUXpresso SDK to be able to build and run RPMSG-Lite examples that are based on mcux-sdk-middleware-rpmsg-lite component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

Visit [RPMSG-Lite - 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 We welcome and encourage the community to submit patches directly to the rpmsg-lite project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

RPMSG-Lite This documentation describes the RPMsg-Lite component, which is a lightweight implementation of the Remote Processor Messaging (RPMsg) protocol. The RPMsg protocol defines a standardized binary interface used to communicate between multiple cores in a heterogeneous multicore system.

Compared to the RPMsg implementation of the Open Asymmetric Multi Processing (OpenAMP) framework (<https://github.com/OpenAMP/open-amp>), the RPMsg-Lite offers a code size reduction, API simplification, and improved modularity. On smaller Cortex-M0+ based systems, it is recommended to use RPMsg-Lite.

The RPMsg-Lite is an open-source component developed by NXP Semiconductors and released under the BSD-compatible license.

For Further documentation, please look at doxygen documentation at: <https://nxp-mcuxpresso.github.io/rpmsg-lite/>

For overview please read RPMSG-Lite VirtIO Overview.

For RPMSG-Lite Design Considerations please read RPMSG-Lite Design Considerations.

Motivation to create RPMsg-Lite There are multiple reasons why RPMsg-Lite was developed. One reason is the need for the small footprint of the RPMsg protocol-compatible communication component, another reason is the simplification of extensive API of OpenAMP RPMsg implementation.

RPMsg protocol was not documented, and its only definition was given by the Linux Kernel and legacy OpenAMP implementations. This has changed with [1] which is a standardization protocol allowing multiple different implementations to coexist and still be mutually compatible.

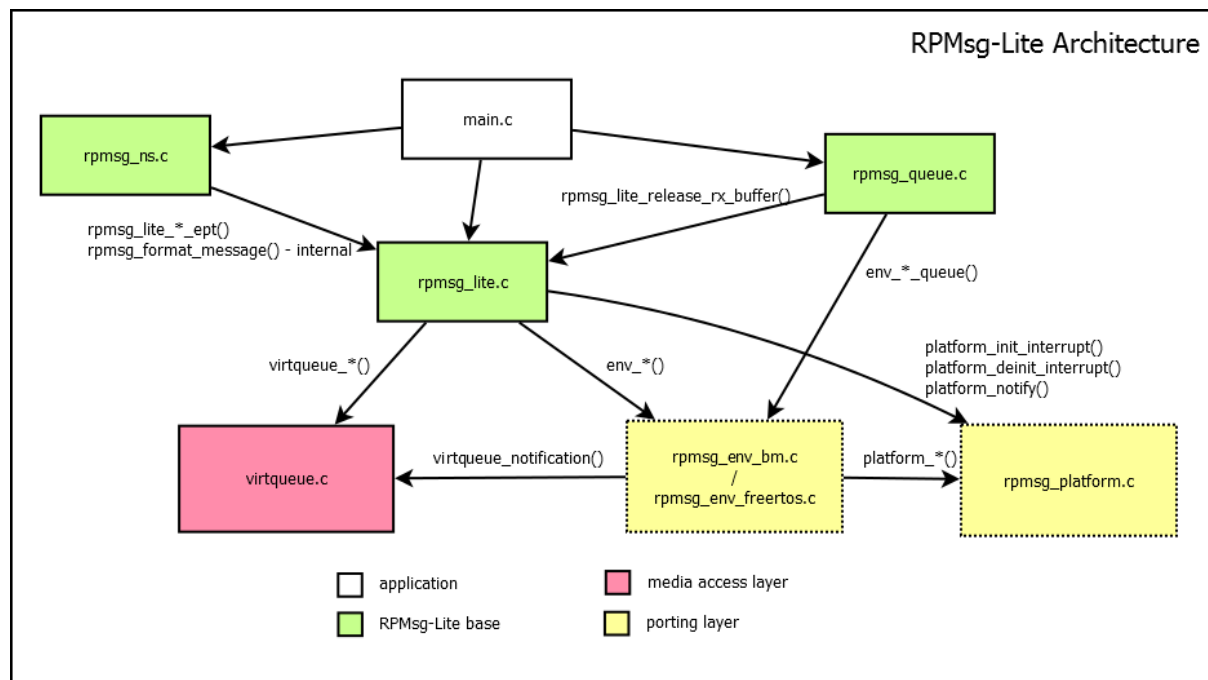
Small MCU-based systems often do not implement dynamic memory allocation. The creation of static API in RPMsg-Lite enables another reduction of resource usage. Not only does the dynamic allocation adds another 5 KB of code size, but also communication is slower and less deterministic, which is a property introduced by dynamic memory. The following table shows some rough comparison data between the OpenAMP RPMsg implementation and new RPMsg-Lite implementation:

Component / Configuration	Flash [B]	RAM [B]
OpenAMP RPMsg / Release (reference)	5547	456 + dynamic
RPMsg-Lite / Dynamic API, Release	3462	56 + dynamic
Relative Difference [%]	~62.4%	~12.3%
RPMsg-Lite / Static API (no malloc), Release	2926	352
Relative Difference [%]	~52.7%	~77.2%

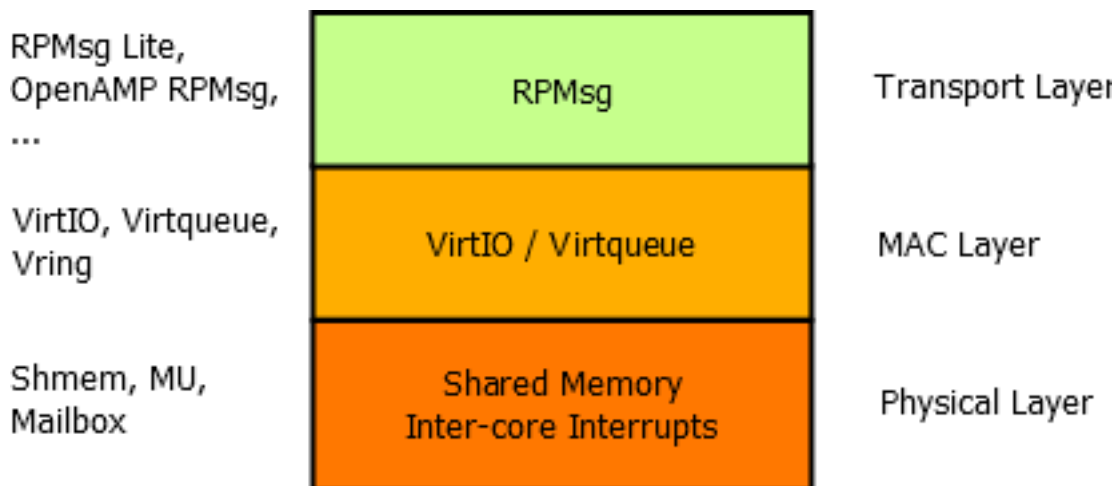
Implementation The implementation of RPMsg-Lite can be divided into three sub-components, from which two are optional. The core component is situated in `rpmsg_lite.c`. Two optional components are used to implement a blocking receive API (in `rpmsg_queue.c`) and dynamic “named” endpoint creation and deletion announcement service (in `rpmsg_ns.c`).

The actual “media access” layer is implemented in `virtqueue.c`, which is one of the few files shared with the OpenAMP implementation. This layer mainly defines the shared memory model, and internally defines used components such as `vring` or `virtqueue`.

The porting layer is split into two sub-layers: the environment layer and the platform layer. The first sublayer is to be implemented separately for each environment. (The bare metal environment already exists and is implemented in `rpmsg_env_bm.c`, and the FreeRTOS environment is implemented in `rpmsg_env_freertos.c` etc.) Only the source file, which matches the used environment, is included in the target application project. The second sublayer is implemented in `rpmsg_platform.c` and defines low-level functions for interrupt enabling, disabling, and triggering mainly. The situation is described in the following figure:



RPMsg-Lite core sub-component This subcomponent implements a blocking send API and callback-based receive API. The RPMsg protocol is part of the transport layer. This is realized by using so-called endpoints. Each endpoint can be assigned a different receive callback function. However, it is important to notice that the callback is executed in an interrupt environment in current design. Therefore, certain actions like memory allocation are discouraged to execute in the callback. The following figure shows the role of RPMsg in an ISO/OSI-like layered model:



Queue sub-component (optional) This subcomponent is optional and requires implementation of the `env_*_queue()` functions in the environment porting layer. It uses a blocking receive API, which is common in RTOS-environments. It supports both copy and nocopy blocking receive functions.

Name Service sub-component (optional) This subcomponent is a minimum implementation of the name service which is present in the Linux Kernel implementation of RPMsg. It allows the communicating node both to send announcements about “named” endpoint (in other words, channel) creation or deletion and to receive these announcement taking any user-defined action

in an application callback. The endpoint address used to receive name service announcements is arbitrarily fixed to be 53 (0x35).

Usage The application should put the `/rpmsg_lite/lib/include` directory to the include path and in the application, include either the `rpmsg_lite.h` header file, or optionally also include the `rpmsg_queue.h` and/or `rpmsg_ns.h` files. Both porting sublayers should be provided for you by NXP, but if you plan to use your own RTOS, all you need to do is to implement your own environment layer (in other words, `rpmsg_env_myrtos.c`) and to include it in the project build.

The initialization of the stack is done by calling the `rpmsg_lite_master_init()` on the master side and the `rpmsg_lite_remote_init()` on the remote side. This initialization function must be called prior to any RPMsg-Lite API call. After the init, it is wise to create a communication endpoint, otherwise communication is not possible. This can be done by calling the `rpmsg_lite_create_ept()` function. It optionally accepts a last argument, where an internal context of the endpoint is created, just in case the `RL_USE_STATIC_API` option is set to 1. If not, the stack internally calls `env_alloc()` to allocate dynamic memory for it. In case a callback-based receiving is to be used, an ISR-callback is registered to each new endpoint with user-defined callback data pointer. If a blocking receive is desired (in case of RTOS environment), the `rpmsg_queue_create()` function must be called before calling `rpmsg_lite_create_ept()`. The queue handle is passed to the endpoint creation function as a callback data argument and the callback function is set to `rpmsg_queue_rx_cb()`. Then, it is possible to use `rpmsg_queue_receive()` function to listen on a queue object for incoming messages. The `rpmsg_lite_send()` function is used to send messages to the other side.

The RPMsg-Lite also implements no-copy mechanisms for both sending and receiving operations. These methods require specifics that have to be considered when used in an application.

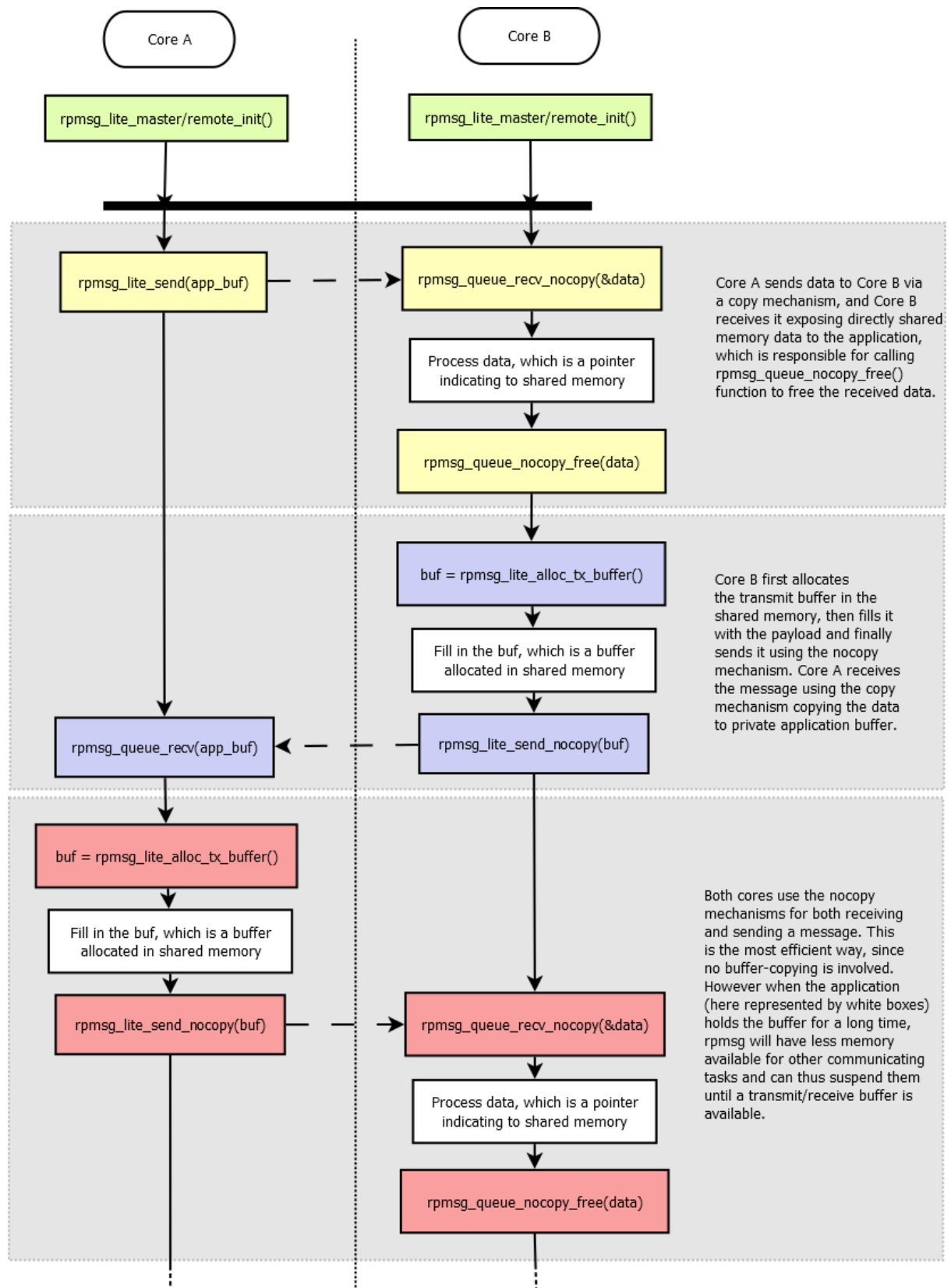
no-copy-send mechanism: This mechanism allows sending messages without the cost for copying data from the application buffer to the RPMsg/virtio buffer in the shared memory. The sequence of no-copy sending steps to be performed is as follows:

- Call the `rpmsg_lite_alloc_tx_buffer()` function to get the virtio buffer and provide the buffer pointer to the application.
- Fill the data to be sent into the pre-allocated virtio buffer. Ensure that the filled data does not exceed the buffer size (provided as the `rpmsg_lite_alloc_tx_buffer()` size output parameter).
- Call the `rpmsg_lite_send_nocopy()` function to send the message to the destination endpoint. Consider the cache functionality and the virtio buffer alignment. See the `rpmsg_lite_send_nocopy()` function description below.

no-copy-receive mechanism: This mechanism allows reading messages without the cost for copying data from the virtio buffer in the shared memory to the application buffer. The sequence of no-copy receiving steps to be performed is as follows:

- Call the `rpmsg_queue_rcv_nocopy()` function to get the virtio buffer pointer to the received data.
- Read received data directly from the shared memory.
- Call the `rpmsg_queue_nocopy_free()` function to release the virtio buffer and to make it available for the next data transfer.

The user is responsible for destroying any RPMsg-Lite objects he has created in case of deinitialization. In order to do this, the function `rpmsg_queue_destroy()` is used to destroy a queue, `rpmsg_lite_destroy_ept()` is used to destroy an endpoint and finally, `rpmsg_lite_deinit()` is used to deinitialize the RPMsg-Lite intercore communication stack. Deinitialize all endpoints using a queue before deinitializing the queue. Otherwise, you are actively invalidating the used queue handle, which is not allowed. RPMsg-Lite does not check this internally, since its main aim is to be lightweight.



Examples RPMsg_Lite multicore examples are part of NXP MCUXpressoSDK packages. Visit <https://mcuxpresso.nxp.com> to configure, build and download these packages. To get the board list with multicore support (RPMsg_Lite included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded,

see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for RPMMsg_Lite multicore examples with 'rpmmsg_lite_' name prefix.

Another way of getting NXP MCUXpressoSDK RPMMsg_Lite multicore examples is using the [mcuxsdk-manifests](#) Github repo. Follow the description how to use the West tool to clone and update the mcuxsdk-manifests repo in [readme section](#). Once done the armgcc rpmmsg_lite examples can be found in

mcuxsdk/examples/_<board_name>/multicore_examples

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the RPMMsg_Lite examples use the 'rpmmsg_lite_' name prefix.

Notes

Environment layers implementation Several environment layers are provided in lib/rpmmsg_lite/porting/environment folder. Not all of them are fully tested however. Here is the list of environment layers that passed testing:

- rpmmsg_env_bm.c
- rpmmsg_env_freertos.c
- rpmmsg_env_xos.c
- rpmmsg_env_threadx.c

The rest of environment layers has been created and used in some experimental projects, it has been running well at the time of creation but due to the lack of unit testing there is no guarantee it is still fully functional.

Shared memory configuration It is important to correctly initialize/configure the shared memory for data exchange in the application. The shared memory must be accessible from both the master and the remote core and it needs to be configured as Non-Cacheable memory. Dedicated shared memory section in linker file is also a good practise, it is recommended to use linker files from MCUXpressoSDK packages for NXP devices based applications. It needs to be ensured no other application part/component is unintentionally accessing this part of memory.

Configuration options The RPMMsg-Lite can be configured at the compile time. The default configuration is defined in the rpmmsg_default_config.h header file. This configuration can be customized by the user by including rpmmsg_config.h file with custom settings. The following table summarizes all possible RPMMsg-Lite configuration options.

Config- uration option	De- fault value	Usage
RL_MS_PE (1)		Delay in milliseconds used in non-blocking API functions for polling.
RL_BUFEE (496)		Size of the buffer payload, it must be equal to (240, 496, 1008, ...) [$2^n - 16$]
RL_BUFEE (2)		Number of the buffers, it must be power of two (2, 4, ...)
RL_API_H (1)		Zero-copy API functions enabled/disabled.
RL_USE_S' (0)		Static API functions (no dynamic allocation) enabled/disabled.
RL_USE_D (0)		Memory cache management of shared memory. Use in case of data cache is enabled for shared memory.
RL_CLEAF (0)		Clearing used buffers before returning back to the pool of free buffers enabled/disabled.
RL_USE_M (0)		When enabled IPC interrupts are managed by the Multicore Manager (IPC interrupts router), when disabled RPMsg-Lite manages IPC interrupts by itself.
RL_USE_E (0)		When enabled the environment layer uses its own context. Required for some environments (QNX). The default value is 0 (no context, saves some RAM).
RL_DEBU (0)		When enabled buffer pointers passed to <code>rpmsg_lite_send_nocopy()</code> and <code>rpmsg_lite_release_rx_buffer()</code> functions (enabled by <code>RL_API_HAS_ZEROCOPY</code> config) are checked to avoid passing invalid buffer pointer. The default value is 0 (disabled). Do not use in RPMsg-Lite to Linux configuration.
RL_ALLO (0)		When enabled the opposite side is notified each time received buffers are consumed and put into the queue of available buffers. Enable this option in RPMsg-Lite to Linux configuration to allow unblocking of the Linux blocking send. The default value is 0 (RPMsg-Lite to RPMsg-Lite communication).
RL_ALLO (0)		It allows to define custom shared memory configuration and replacing the shared memory related global settings from <code>rpmsg_config.h</code> . This is useful when multiple instances are running in parallel but different shared memory arrangement (vring size & alignment, buffers size & count) is required. The default value is 0 (all RPMsg-Lite instances use the same shared memory arrangement as defined by common config macros).
RL_ASSERT	see <code>rpmsg</code>	Assert implementation.

How to format rpmsg-lite code To format code, use the application developed by Google, named *clang-format*. This tool is part of the [llvm](#) project. Currently, the clang-format 10.0.0 version is used for rpmsg-lite. The set of style settings used for clang-format is defined in the `.clang-format` file, placed in a root of the rpmsg-lite directory where Python script `run_clang_format.py` can be executed. This script executes the application named *clang-format.exe*. You need to have the path of this application in the OS's environment path, or you need to change the script.

References

[1] M. Novak, M. Cingel, **Lockless Shared Memory Based Multicore Communication Protocol**
Copyright © 2016 Freescale Semiconductor, Inc. Copyright © 2016-2025 NXP

Changelog RPMSG-Lite All notable changes to this project will be documented in this file. The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Fixed

- Fixed CERT-C INT31-C violation in platform_notify function in rpmsg_platform.c for imxrt700_m33, imxrt700_hifi4, imxrt700_hifi1 platforms

v5.2.0

Added

- Add MCXL20 porting layer and unit testing
- New utility macro RL_CALCULATE_BUFFER_COUNT_DOWN_SAFE to safely determine maximum buffer count within shared memory while preventing integer underflow.
- RT700 platform add support for MCMGR in DSPs

Changed

- Change rpmsg_platform.c to support new MCMGR API
- Improved input validation in initialization functions to properly handle insufficient memory size conditions.
- Refactored repeated buffer count calculation pattern for better code maintainability.
- To make sure that remote has already registered IRQ there is required App level IPC mechanism to notify master about it

Fixed

- Fixed env_wait_for_link_up function to handle timeout in link state checks for baremetal and qnx environment, RL_BLOCK mode can be used to wait indefinitely.
- Fixed CERT-C INT31-C violation by adding compile-time check to ensure RL_PLATFORM_HIGHEST_LINK_ID remains within safe range for 16-bit casting in virtqueue ID creation.
- Fixed CERT-C INT30-C violations by adding protection against unsigned integer underflow in shared memory calculations, specifically in shmem_length - (uint32_t)RL_VRING_OVERHEAD and shmem_length - 2U * shmem_config.vring_size expressions.
- Fixed CERT INT31-C violation in platform_interrupt_disable() and similar functions by replacing unsafe cast from uint32_t to int32_t with a return of 0 constant.
- Fixed unsigned integer underflow in rpmsg_lite_alloc_tx_buffer() where subtracting header size from buffer size could wrap around if buffer was too small, potentially leading to incorrect buffer sizing.
- Fixed CERT-C INT31-C violation in rpmsg_lite.c where size parameter was cast from uint32_t to uint16_t without proper validation.
 - Applied consistent masking approach to both size and flags parameters: (uint16_t)(value & 0xFFFFU).
 - This fix prevents potential data loss when size values exceed 65535.

- Fixed CERT INT31-C violation in `env_memset` functions by explicitly converting `int32_t` values to unsigned char using bit masking. This prevents potential data loss or misinterpretation when passing values outside the unsigned char range (0-255) to the standard `memset()` function.
- Fixed CERT-C INT31-C violations in RPMsg-Lite environment porting: Added validation checks for signed-to-unsigned integer conversions to prevent data loss and misinterpretation.
 - `rpmsg_env_freertos.c`: Added validation before converting `int32_t` to `UBaseType_t`.
 - `rpmsg_env_qnx.c`: Fixed format string and added validation before assigning to `mqstat` fields.
 - `rpmsg_env_threadx.c`: Added validation to prevent integer overflow and negative values.
 - `rpmsg_env_xos.c`: Added range checking before casting to `uint16_t`.
 - `rpmsg_env_zephyr.c`: Added validation before passing values to `k_msgq_init`.
- Fixed a CERT INT31-C compliance issue in `env_get_current_queue_size()` function where an unsigned queue count was cast to a signed `int32_t` without proper validation, which could lead to lost or misinterpreted data if queue size exceeded `INT32_MAX`.
- Fixed CERT INT31-C violation in `rpmsg_platform.c` where `memcmp()` return value (signed int) was compared with unsigned constant without proper type handling.
- Fixed CERT INT31-C violation in `rpmsg_platform.c` where casting from `uint32_t` to `uint16_t` could potentially result in data loss. Changed length variable type from `uint16_t` to `uint32_t` to properly handle memory address differences without truncation.
- Fixed potential integer overflow in `env_sleep_msec()` function in ThreadX environment implementation by rearranging calculation order in the sleep duration formula.
- Fixed CERT-C INT31-C violation in RPMsg-Lite where bitwise NOT operations on integer constants were performed in signed integer context before being cast to unsigned. This could potentially lead to misinterpreted data on `imx943` platform.
- Added `RL_MAX_BUFFER_COUNT` (32768U) and `RL_MAX_VRING_ALIGN` (65536U) limit to ensure alignment values cannot contribute to integer overflow
- Fixed CERT INT31-C violation in `vring_need_event()`, added cast to `uint16_t` for each operand.

v5.1.4 - 27-Mar-2025

Added

- Add KW43B43 porting layer

Changed

- Doxygen bump to version 1.9.6

v5.1.3 - 13-Jan-2025

Added

- Memory cache management of shared memory. Enable with `#define RL_USE_DCACHE (1)` in `rpmsg_config.h` in case of data cache is used.
- Cmake/Kconfig support added.
- Porting layers for imx95, imxrt700, mcmxw71x, mcmxw72x, kw47b42 added.

v5.1.2 - 08-Jul-2024**Changed**

- Zephyr-related changes.
- Minor Misra corrections.

v5.1.1 - 19-Jan-2024**Added**

- Test suite provided.
- Zephyr support added.

Changed

- Minor changes in platform and env. layers, minor test code updates.

v5.1.0 - 02-Aug-2023**Added**

- RPMsg-Lite: Added aarch64 support.

Changed

- RPMsg-Lite: Increased the queue size to $(2 * RL\_BUFFER\_COUNT)$ to cover zero copy cases.
- Code formatting using LLVM16.

Fixed

- Resolved issues in ThreadX env. layer implementation.

v5.0.0 - 19-Jan-2023**Added**

- Timeout parameter added to `rpmsg_lite_wait_for_link_up` API function.

Changed

- Improved debug check buffers implementation - instead of checking the pointer fits into shared memory check the presence in the VirtIO ring descriptors list.
- VRING_SIZE is set based on number of used buffers now (as calculated in vring_init) - updated for all platforms that are not communicating to Linux rpmsg counterpart.

Fixed

- Fixed wrong RL_VRING_OVERHEAD macro comment in platform.h files
- Misra corrections.

v4.0.0 - 20-Jun-2022

Added

- Added support for custom shared memory arrangement per the RPMsg_Lite instance.
- Introduced new rpmsg_lite_wait_for_link_up() API function - this allows to avoid using busy loops in rtos environments, GitHub PR [#21](#).

Changed

- Adjusted rpmsg_lite_is_link_up() to return RL_TRUE/RL_FALSE.

v3.2.0 - 17-Jan-2022

Added

- Added support for i.MX8 MP multicore platform.

Changed

- Improved static allocations - allow OS-specific objects being allocated statically, GitHub PR [#14](#).
- Aligned rpmsg_env_xos.c and some platform layers to latest static allocation support.

Fixed

- Minor Misra and typo corrections, GitHub PR [#19](#), [#20](#).

v3.1.2 - 16-Jul-2021

Added

- Addressed MISRA 21.6 rule violation in rpmsg_env.h (use SDK's PRINTF in MCUXpressoSDK examples, otherwise stdio printf is used).
- Added environment layers for XOS.
- Added support for i.MX RT500, i.MX RT1160 and i.MX RT1170 multicore platforms.

Fixed

- Fixed incorrect description of the `rpmsg_lite_get_endpoint_from_addr` function.

Changed

- Updated `RL_BUFFER_COUNT` documentation (issue [#10](#)).
- Updated `imxrt600_hifi4` platform layer.

v3.1.1 - 15-Jan-2021**Added**

- Introduced `RL_ALLOW_CONSUMED_BUFFERS_NOTIFICATION` config option to allow opposite side notification sending each time received buffers are consumed and put into the queue of available buffers.
- Added environment layers for Threadx.
- Added support for i.MX8QM multicore platform.

Changed

- Several MISRA C-2012 violations addressed.

v3.1.0 - 22-Jul-2020**Added**

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed (7.4).
- Fixed missing lock in `rpmsg_lite_rx_callback()` for QNX env.
- Correction of `rpmsg_lite_instance` structure members description.
- Address -Waddress-of-packed-member warnings in GCC9.

Changed

- Clang update to v10.0.0, code re-formatted.

v3.0.0 - 20-Dec-2019**Added**

- Added support for several new multicore platforms.

Fixed

- MISRA C-2012 violations fixed, incl. data types consolidation.
- Code formatted.

v2.2.0 - 20-Mar-2019

Added

- Added configuration macro `RL_DEBUG_CHECK_BUFFERS`.
- Several MISRA violations fixed.
- Added environment layers for QNX and Zephyr.
- Allow environment context required for some environment (controlled by the `RL_USE_ENVIRONMENT_CONTEXT` configuration macro).
- Data types consolidation.

v1.1.0 - 28-Apr-2017

Added

- Supporting i.MX6SX and i.MX7D MPU platforms.
- Supporting LPC5411x MCU platform.
- Baremetal and FreeRTOS support.
- Support of copy and zero-copy transfer.
- Support of static API (without dynamic allocations).

Multicore Manager

MCUXpresso SDK : `mcuxsdk-middleware-mcmgr` (Multicore Manager)

Overview This repository is for MCUXpresso SDK Multicore Manager middleware delivery and it contains Multicore Manager component 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](#) for the complete delivery of MCUXpresso SDK to be able to build and run Multicore Manager examples that are based on `mcux-sdk-middleware-mcmgr` component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

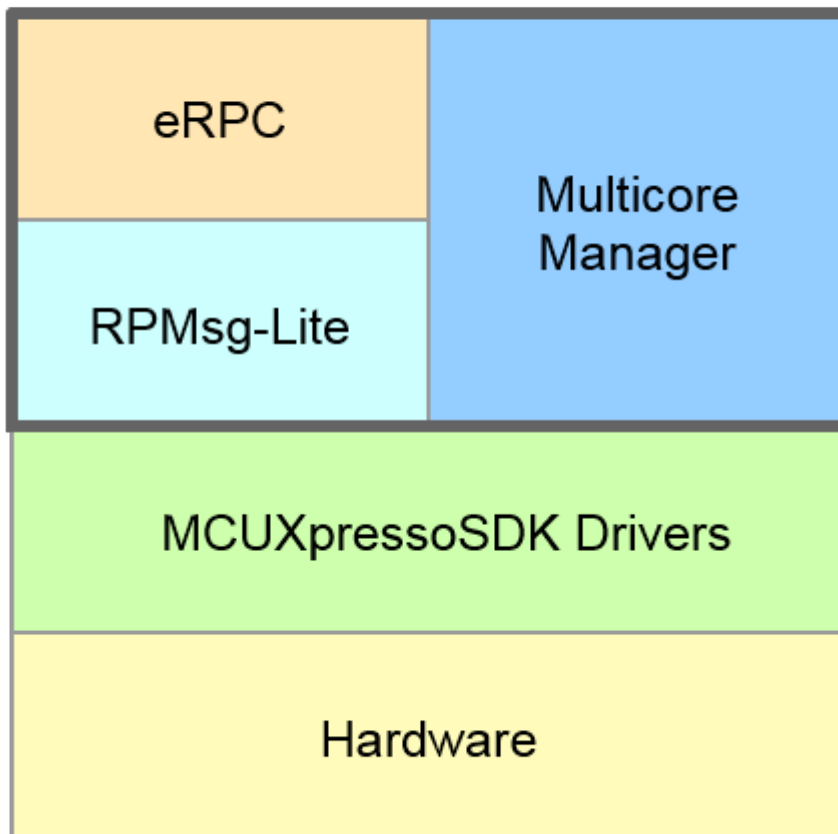
Visit [Multicore Manager - Documentation](#) to review details on the contents in this sub-repo.

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

Contribution We welcome and encourage the community to submit patches directly to the mcmgr project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

Multicore Manager (MCMGR) The Multicore Manager (MCMGR) software library provides a number of services for multicore systems. This library is distributed as a part of the Multicore SDK (MCSDK). Together, the MCSDK and the MCUXpresso SDK (SDK) form a framework for development of software for NXP multicore devices.

The MCMGR component is located in the <MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr directory.



The Multicore Manager provides the following major functions:

- Maintains information about all cores in system.
- Secondary/auxiliary core(s) startup and shutdown.
- Remote core monitoring and event handling.

For Further API documentation, please look at doxygen documentation at: <https://nxp-mcuxpresso.github.io/mcux-mcmgr/>

Usage of the MCMGR software component The main use case of MCMGR is the secondary/auxiliary core start. This functionality is performed by the public API function.

Example of MCMGR usage to start secondary core:

```
#include "mcmgr.h"

void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();

    /* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass "1" as startup data,
    ↪starting synchronously. */
    MCMGR_StartCore(kMCMGR_Core1, CORE1_BOOT_ADDRESS, 1, kMCMGR_Start_Synchronous);
    .
    .
    .
    /* Stop secondary core execution. */
    MCMGR_StopCore(kMCMGR_Core1);
}
```

Some platforms allow stopping and re-starting the secondary core application again, using the MCMGR_StopCore / MCMGR_StartCore API calls. It is necessary to ensure the initially loaded image is not corrupted before re-starting, especially if it deals with the RAM target. Cache coherence has to be considered/ensured as well.

Another important MCMGR feature is the ability for remote core monitoring and handling of events such as reset, exception, and application events. Application-specific callback functions for events are registered by the MCMGR_RegisterEvent() API. Triggering these events is done using the MCMGR_TriggerEvent() API. mcmgr_event_type_t enums all possible event types.

An example of MCMGR usage for remote core monitoring and event handling. Code for the primary side:

```
#include "mcmgr.h"

#define APP_RPMSG_READY_EVENT_DATA (1)
#define APP_NUMBER_OF_CORES (2)
#define APP_SECONDARY_CORE kMCMGR_Core1

/* Callback function registered via the MCMGR_RegisterEvent() and triggered by MCMGR_TriggerEvent()
↪called on the secondary core side */
void RPSGRemoteReadyEventHandler(mcmgr_core_t coreNum, uint16_t eventData, void *context)
{
    uint16_t *data = &((uint16_t *)context)[coreNum];

    *data = eventData;
}

void main()
{
    uint16_t RPSGRemoteReadyEventData[NUMBER_OF_CORES] = {0};

    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();
```

(continues on next page)

(continued from previous page)

```

/* Initialize MCMGR, install generic event handlers */
MCMGR_Init();

/* Register the application event before starting the secondary core */
MCMGR_RegisterEvent(kMCMGR_RemoteApplicationEvent, RPSMsgRemoteReadyEventHandler, (void_)
↳*)RPSMsgRemoteReadyEventData);

/* Boot secondary core application from the CORE1_BOOT_ADDRESS, pass rpsmsg_lite_base address_
↳as startup data, starting synchronously. */
MCMGR_StartCore(APP_SECONDARY_CORE, CORE1_BOOT_ADDRESS, (uint32_t)rpsmsg_lite_
↳base, kMCMGR_Start_Synchronous);

/* Wait until the secondary core application signals the rpsmsg remote has been initialized and is ready to_
↳communicate. */
while(APP_RPSMSG_READY_EVENT_DATA != RPSMsgRemoteReadyEventData[APP_SECONDARY_
↳CORE]) {};
.
.
.
}

```

Code for the secondary side:

```

#include "mcmgr.h"

#define APP_RPSMSG_READY_EVENT_DATA (1)

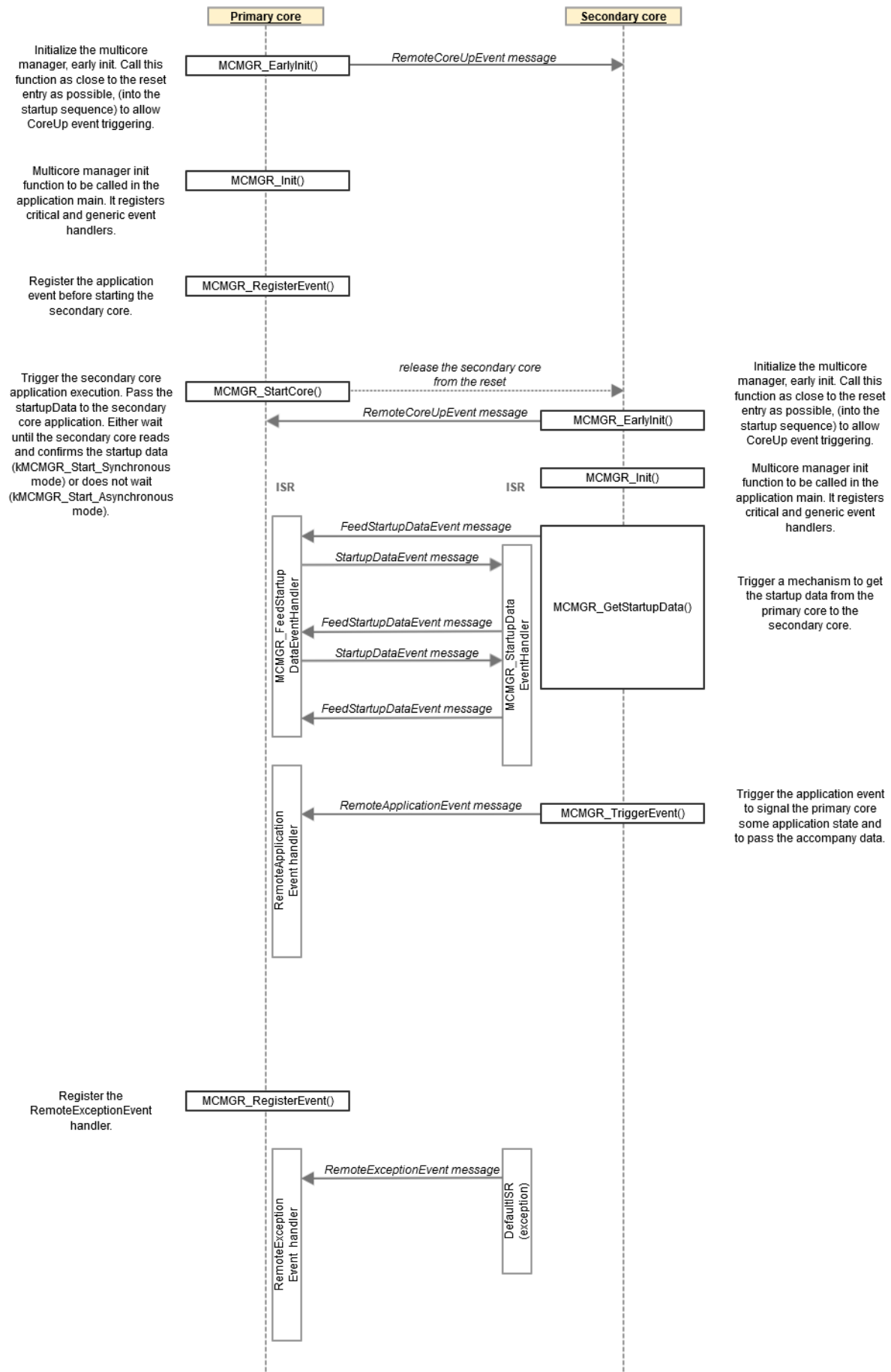
void main()
{
    /* Initialize MCMGR - low level multicore management library.
       Call this function as close to the reset entry as possible,
       (into the startup sequence) to allow CoreUp event triggering. */
    MCMGR_EarlyInit();

    /* Initialize MCMGR, install generic event handlers */
    MCMGR_Init();
    .
    .
    .

    /* Signal the to other core that we are ready by triggering the event and passing the APP_RPSMSG_
    ↳READY_EVENT_DATA */
    MCMGR_TriggerEvent(kMCMGR_Core0, kMCMGR_RemoteApplicationEvent, APP_RPSMSG_
    ↳READY_EVENT_DATA);
    .
    .
    .
}

```

MCMGR Data Exchange Diagram The following picture shows how the handshakes are supposed to work between the two cores in the MCMGR software.



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

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Added

Fixed

- Added CX flag into CMakeLists.txt to allow c++ build compatibility.

v5.0.0

Added

- Added MCMGR_BUSY_POLL_COUNT macro to prevent infinite polling loops in MCMGR operations.
- Implemented timeout mechanism for all polling loops in MCMGR code.
- Added support to handle more then two cores. Breaking API change by adding parameter `coreNum` specifying core number in functions bellow.
 - MCMGR_GetStartupData(uint32_t *startupData, mcmgr_core_t coreNum)
 - MCMGR_TriggerEvent(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
 - MCMGR_TriggerEventForce(mcmgr_event_type_t type, uint16_t eventData, mcmgr_core_t coreNum)
 - typedef void (*mcmgr_event_callback_t)(uint16_t data, void *context, mcmgr_core_t coreNum);

When registering the event with function `MCMGR_RegisterEvent()` user now needs to provide `callbackData` pointer to array of elements per every core in system (see README.md for example). In case of systems with only two cores the `coreNum` in callback can be ignored as events can arrive only from one core. Please see Porting guide for more details: [Porting-GuideTo_v5.md](#)

- Updated all porting files to support new MCMGR API.
- Added new platform specific include file `mcmgr_platform.h`. It will contain common platform specific macros that can be then used in `mcmgr` and application. e.g. platform core count `MCMGR_CORECOUNT` 4.
- Move all header files to new `inc` directory.
- Added new platform-specific include files `inc/platform/<platform_name>/mcmgr_platform.h`.

Added

- Add MCXL20 porting layer and unit testing

v4.1.7

Fixed

- `mcmgr_stop_core_internal()` function now returns `kStatus_MCMGR_NotImplemented` status code instead of `kStatus_MCMGR_Success` when device does not support stop of secondary core. Ports affected: kw32w1, kw45b41, kw45b42, mcxw716, mcxw727.

[v4.1.6]

Added

- Multicore Manager moved to standalone repository.
- Add porting layers for imxrt700, mcmxw727, kw47b42.
- New `MCMGR_ProcessDeferredRxIsr()` API added.

[v4.1.5]

Added

- Add notification into `MCMGR_EarlyInit` and `mcmgr_early_init_internal` functions to avoid using uninitialized data in their implementations.

[v4.1.4]

Fixed

- Avoid calling tx isr callbacks when respective Messaging Unit Transmit Interrupt Enable flag is not set in the CR/TCR register.
- Messaging Unit RX and status registers are cleared after the initialization.

[v4.1.3]

Added

- Add porting layers for imxrt1180.

Fixed

- `mu_isr()` updated to avoid calling tx isr callbacks when respective Transmit Interrupt Enable flag is not set in the CR/TCR register.
- `mcmgr_mu_internal.c` code adaptation to new supported SoCs.

[v4.1.2]

Fixed

- Update `mcmgr_stop_core_internal()` implementations to set core state to `kMCMGR_ResetCoreState`.

[v4.1.0]

Fixed

- Code adjustments to address MISRA C-2012 Rules

[v4.0.3]

Fixed

- Documentation updated to describe handshaking in a graphic form.
- Minor code adjustments based on static analysis tool findings

[v4.0.2]

Fixed

- Align porting layers to the updated MCUXpressoSDK feature files.

[v4.0.1]

Fixed

- Code formatting, removed unused code

[v4.0.0]

Added

- Add new MCMGR_TriggerEventForce() API.

[v3.0.0]

Removed

- Removed MCMGR_LoadApp(), MCMGR_MapAddress() and MCMGR_SignalReady()

Modified

- Modified MCMGR_GetStartupData()

Added

- Added MCMGR_EarlyInit(), MCMGR_RegisterEvent() and MCMGR_TriggerEvent()
- Added the ability for remote core monitoring and event handling

[v2.0.1]

Fixed

- Updated to be Misra compliant.

[v2.0.0]

Added

- Support for lpcxpresso54114 board.

[v1.1.0]

Fixed

- Ported to KSDK 2.0.0.

[v1.0.0]

Added

- Initial release.

eRPC

MCUXpresso SDK : mcuxsdk-middleware-erpc

Overview This repository is for MCUXpresso SDK eRPC middleware delivery and it contains eRPC component 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](#) for the complete delivery of MCUXpresso SDK to be able to build and run eRPC examples that are based on mcux-sdk-middleware-erpc component.

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

Visit [eRPC - 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 We welcome and encourage the community to submit patches directly to the eRPC project placed on github. Contributing can be managed via pull-requests. Before a pull-request is created the code should be tested and properly formatted.

eRPC

- [*MCUXpresso SDK : mcuxsdk-middleware-erpc*](#)

- [*Overview*](#)
- [*Documentation*](#)
- [*Setup*](#)
- [*Contribution*](#)

- [*eRPC*](#)

- [*About*](#)
- [*Releases*](#)
 - * [*Edge releases*](#)
- [*Documentation*](#)
- [*Examples*](#)
- [*References*](#)
- [*Directories*](#)
- [*Building and installing*](#)
 - * [*Requirements*](#)
 - [*Windows*](#)
 - [*Mac OS X*](#)
 - * [*Building*](#)
 - [*CMake and KConfig*](#)
 - [*Make*](#)
 - * [*Installing for Python*](#)
- [*Known issues and limitations*](#)
- [*Code providing*](#)

About

eRPC (Embedded RPC) is an open source Remote Procedure Call (RPC) system for multichip embedded systems and heterogeneous multicore SoCs.

Unlike other modern RPC systems, such as the excellent [Apache Thrift](#), eRPC distinguishes itself by being designed for tightly coupled systems, using plain C for remote functions, and having a small code size (<5kB). It is not intended for high performance distributed systems over a network.

eRPC does not force upon you any particular API style. It allows you to export existing C functions, without having to change their prototypes. (There are limits, of course.) And although the

internal infrastructure is written in C++, most users will be able to use only the simple C setup APIs shown in the examples below.

A code generator tool called `erpcgen` is included. It accepts input IDL files, having an `.erpc` extension, that have definitions of your data types and remote interfaces, and generates the shim code that handles serialization and invocation. `erpcgen` can generate either C/C++ or Python code.

Example `.erpc` file:

```
// Define a data type.
enum LEDName { kRed, kGreen, kBlue }

// An interface is a logical grouping of functions.
interface IO {
    // Simple function declaration with an empty reply.
    set_led(LEDName whichLed, bool onOrOff) -> void
}
```

Client side usage:

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* init eRPC client IO service */
    initIO_client(client_manager);

    // Now we can call the remote function to turn on the green LED.
    set_led(kGreen, true);

    /* deinit objects */
    deinitIO_client();
    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}
```

```
void example_client(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_client_t client_manager;

    /* Init eRPC client infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    client_manager = erpc_client_init(transport, message_buffer_factory);

    /* scope for client service */
    {
        /* init eRPC client IO service */
        IO_client client(client_manager);

        // Now we can call the remote function to turn on the green LED.
        client.set_led(kGreen, true);
    }

    /* deinit objects */
```

(continues on next page)

(continued from previous page)

```

    erpc_client_deinit(client_manager);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}

```

Server side usage:

```

// Implement the remote function.
void set_led(LEDName whichLed, bool onOrOff) {
    // implementation goes here
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    erpc_service_t service = create_IO_service();

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, service);

    // Run the server.
    erpc_server_run();

    /* deinit objects */
    destroy_IO_service(service);
    erpc_server_deinit(server);
    erpc_mbf_dynamic_deinit(message_buffer_factory);
    erpc_transport_tcp_deinit(transport);
}

```

```

// Implement the remote function.
class IO : public IO_interface
{
    /* eRPC call definition */
    void set_led(LEDName whichLed, bool onOrOff) override {
        // implementation goes here
    }
}

void example_server(void) {
    erpc_transport_t transport;
    erpc_mbf_t message_buffer_factory;
    erpc_server_t server;
    IO IOImpl;
    IO_service io(&IOImpl);

    /* Init eRPC server infrastructure */
    transport = erpc_transport_cmsis_uart_init(Driver_USART0);
    message_buffer_factory = erpc_mbf_dynamic_init();
    server = erpc_server_init(transport, message_buffer_factory);

    /* add custom service implementation to the server */
    erpc_add_service_to_server(server, &io);

    /* poll for requests */
}

```

(continues on next page)

(continued from previous page)

```
erpc_status_t err = server.run();

/* deinit objects */
erpc_server_deinit(server);
erpc_mbf_dynamic_deinit(message_buffer_factory);
erpc_transport_tcp_deinit(transport);
}
```

A number of transports are supported, and new transport classes are easy to write.

Supported transports can be found in *erpc/erpc_c/transport* folder. E.g:

- CMSIS UART
- NXP Kinetis SPI and DSPI
- POSIX and Windows serial port
- TCP/IP (mostly for testing)
- [NXP RPMsg-Lite / RPMsg TTY](#)
- SPIdev Linux
- USB CDC
- NXP Messaging Unit

eRPC is available with an unrestrictive BSD 3-clause license. See the [LICENSE file](#) for the full license text.

Releases [eRPC releases](#)

Edge releases Edge releases can be found on [eRPC CircleCI](#) webpage. Choose build of interest, then platform target and choose ARTIFACTS tab. Here you can find binary application from chosen build.

Documentation [Documentation](#) is in the wiki section.

[eRPC Infrastructure documentation](#)

Examples *Example IDL* is available in the *examples/* folder.

Plenty of eRPC multicore and multiprocessor examples can be also found in NXP MCUXpressoSDK packages. Visit <https://mcuxpresso.nxp.com> to configure, build and download these packages.

To get the board list with multicore support (eRPC included) use filtering based on Middleware and search for 'multicore' string. Once the selected package with the multicore middleware is downloaded, see

<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples for eRPC multicore examples (RPMsg_Lite or Messaging Unit transports used) or

<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples for eRPC multiprocessor examples (UART or SPI transports used).

eRPC examples use the 'erpc_' name prefix.

Another way of getting NXP MCUXpressoSDK eRPC multicore and multiprocessor examples is using the [mcux-sdk](#) Github repo. Follow the description how to use the West tool to clone and

update the mcuxsdk repo in [readme Overview section](#). Once done the armgcc eRPC examples can be found in

mcuxsdk/examples/<board_name>/multicore_examples or in

mcuxsdk/examples/<board_name>/multiprocessor_examples folders.

You can use the evkmimxrt1170 as the board_name for instance. Similar to MCUXpressoSDK packages the eRPC examples use the 'erpc_' name prefix.

References This section provides links to interesting erpc-based projects, articles, blogs or guides:

- [erpc \(EmbeddedRPC\) getting started notes](#)
- [ERPC Linux Local Environment Construction and Use](#)
- [The New Wio Terminal eRPC Firmware](#)

Directories *doc* - Documentation.

doxygen - Configuration and support files for running Doxygen over the eRPC C++ infrastructure and erpcgen code.

erpc_c - Holds C/C++ infrastructure for eRPC. This is the code you will include in your application.

erpc_python - Holds Python version of the eRPC infrastructure.

erpcgen - Holds source code for erpcgen and makefiles or project files to build erpcgen on Windows, Linux, and OS X.

erpcsniffer - Holds source code for erpcsniffer application.

examples - Several example IDL files.

mk - Contains common makefiles for building eRPC components.

test - Client/server tests. These tests verify the entire communications path from client to server and back.

utilities - Holds utilities which bring additional benefit to eRPC apps developers.

Building and installing These build instructions apply to host PCs and embedded Linux. For bare metal or RTOS embedded environments, you should copy the *erpc_c* directory into your application sources.

CMake and KConfig build:

It builds a static library of the eRPC C/C++ infrastructure, the *erpcgen* executable, and optionally the unit tests and examples.

CMake is compatible with gcc and clang. On Windows local MingGW downloaded by *script* can be used.

Make build:

It builds a static library of the eRPC C/C++ infrastructure, the *erpcgen* executable, and optionally the unit tests.

The makefiles are compatible with gcc or clang on Linux, OS X, and Cygwin. A Windows build of *erpcgen* using Visual Studio is also available in the *erpcgen/VisualStudio_v14* directory. There is also an Xcode project file in the *erpcgen* directory, which can be used to build *erpcgen* for OS X.

Requirements erPC now support building **erpcgen**, **erpc_lib**, **tests** and **C examples** using CMake.

Requirements when using CMake:

- **CMake** (minimal version 3.20.0)
- Generator - **Make, Ninja, ...**
- **C/C++ compiler** - **GCC, CLANG, ...**
- **Bison** - <https://www.gnu.org/software/bison/>
- **Flex** - <https://github.com/westes/flex/>

Requirements when using Make:

- **Make**
- **C/C++ compiler - GCC, CLANG, ...**
- **Binson** - <https://www.gnu.org/software/bison/>
- **Flex** - <https://github.com/westes/flex/>

Windows Related steps to build **erpcgen** using **Visual Studio** are described in `erpcgen/VisualStudio_v14/readme_erpcgen.txt`.

To install MinGW, Bison, Flex locally on Windows:

```
./install_dependencies.ps1
* \ \ \

#### Linux

\ \ \ bash
./install_dependencies.sh
```

Mandatory for case, when build for different architecture is needed

- gcc-multilib, g++-multilib

Mac OS X

```
./install_dependencies.sh
```

Building

CMake and KConfig eRPC use CMake and KConfig to configurate and build eRPC related targets. KConfig can be edited by *prj.conf* or *menuconfig* when building.

Generate project, config and build. In *erpc/* execute:

```
cmake -B ./build # in erpc/build generate cmake project
cmake --build ./build --target menuconfig # Build menuconfig and configure erpcgen, erpc_lib, tests and examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

****CMake will use the system's default compilers and generator**

If you want to use Windows and locally installed MinGW, use *CMake preset* :

```
cmake --preset mingw64 # Generate project in ./build using mingw64's make and compilers
cmake --build ./build --target menuconfig # Build menuconfig and configurate erpcgen, erpc_lib, tests and ↵
↵examples
cmake --build ./build # Build all selected target from prj.conf/menuconfig
```

Make To build the library and erpcgen, run from the repo root directory:

```
make
```

To install the library, erpcgen, and include files, run:

```
make install
```

You may need to sudo the make install.

By default this will install into /usr/local. If you want to install elsewhere, set the PREFIX environment variable. Example for installing into /opt:

```
make install PREFIX=/opt
```

List of top level Makefile targets:

- erpc: build the liberpc.a static library
- erpcgen: build the erpcgen tool
- erpcsniffer: build the sniffer tool
- test: build the unit tests under the *test* directory
- all: build all of the above
- install: install liberpc.a, erpcgen, and include files

eRPC code is validated with respect to the C++ 11 standard.

Installing for Python To install the Python infrastructure for eRPC see instructions in the *erpc python readme*.

Known issues and limitations

- Static allocations controlled by the ERPC_ALLOCATION_POLICY config macro are not fully supported yet, i.e. not all erpc objects can be allocated statically now. It deals with the ongoing process and the full static allocations support will be added in the future.

Code providing Repository on Github contains two main branches: **main** and **develop**. Code is developed on **develop** branch. Release version is created via merging **develop** branch into **main** branch.

Copyright 2014-2016 Freescale Semiconductor, Inc.

Copyright 2016-2025 NXP

eRPC Getting Started

Overview This *Getting Started User Guide* shows software developers how to use Remote Procedure Calls (RPC) in embedded multicore microcontrollers (eRPC).

The eRPC documentation is located in the `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/doc` folder.

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
 1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
 2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
 3. Add infrastructure files.
 4. Add user code for client and server applications.
 5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

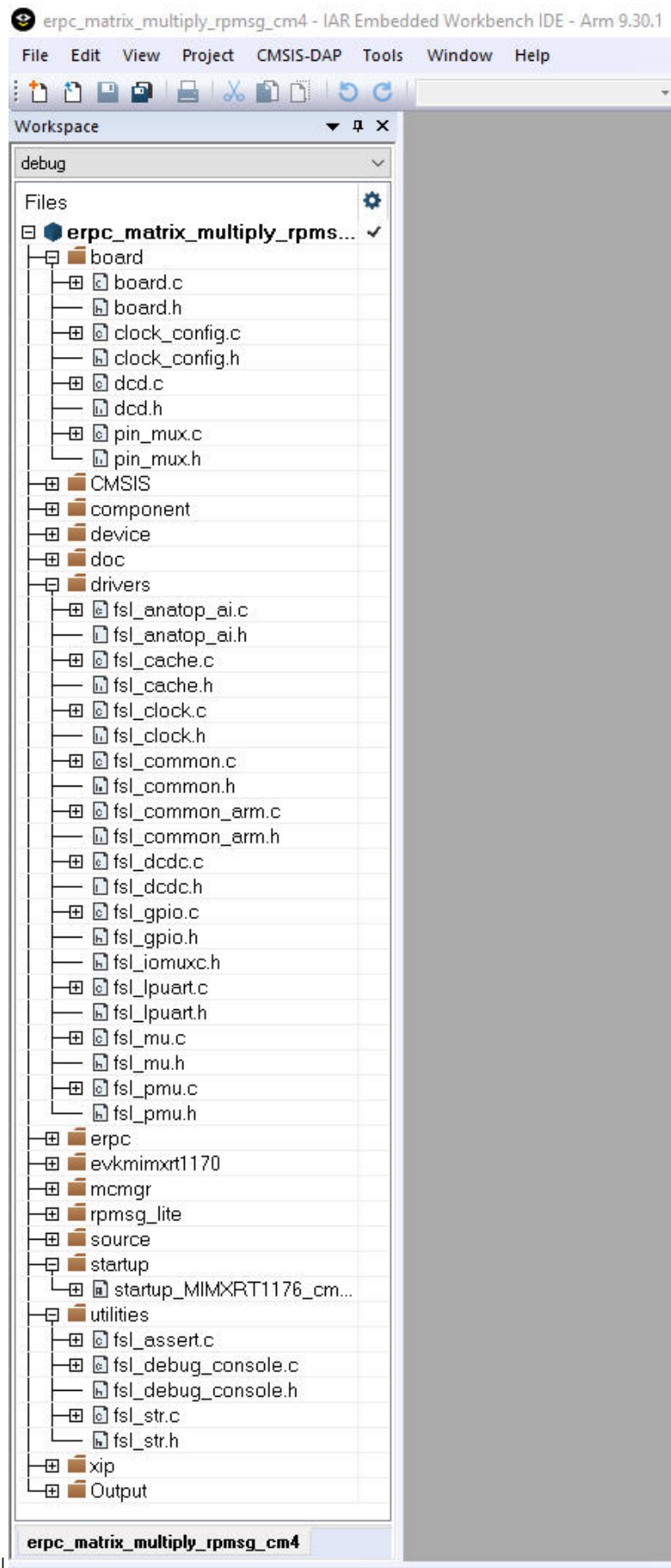
Multicore server application The “Matrix multiply” eRPC server project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmmsg/cm4/iar/`

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



|

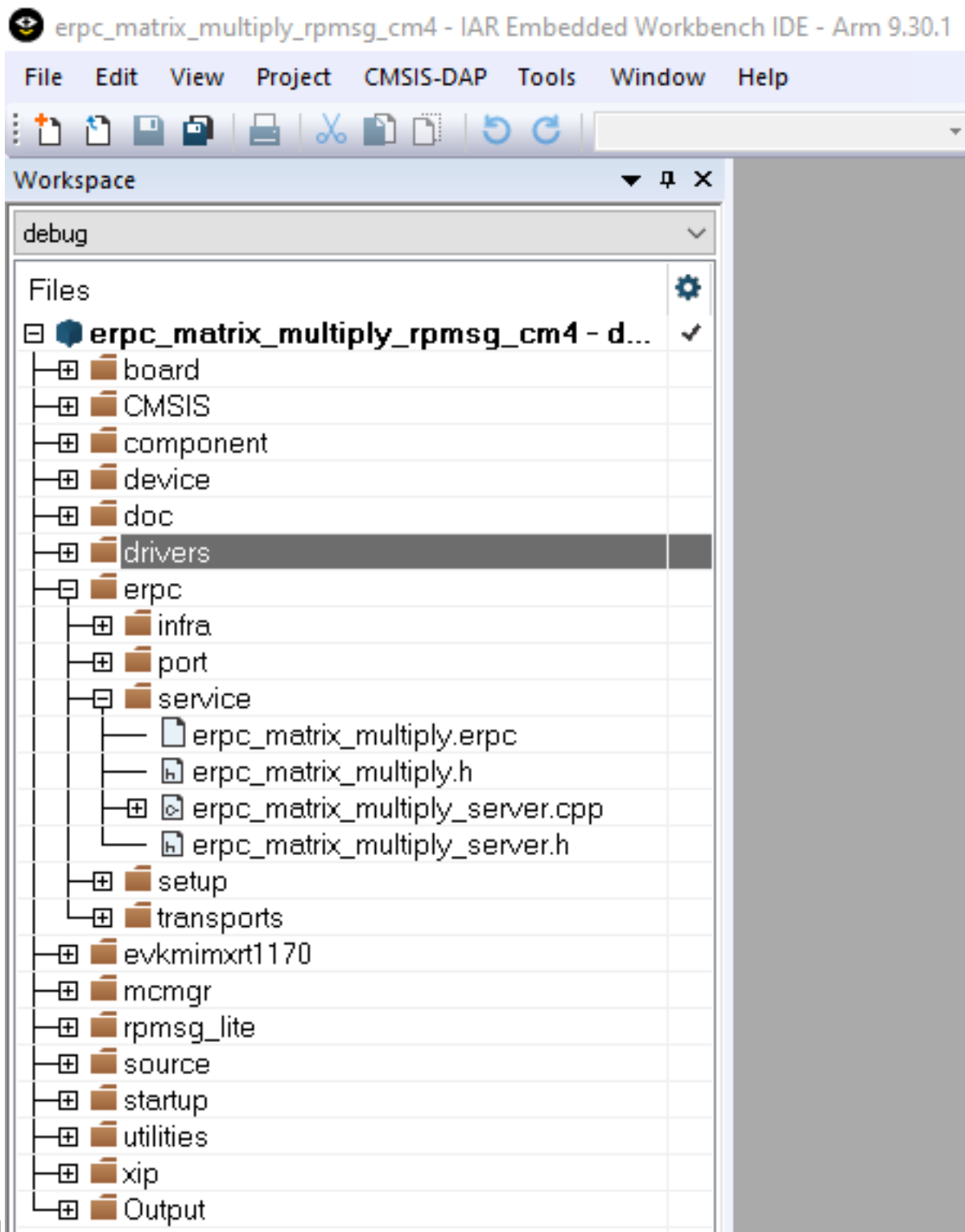
Parent topic: Multicore server application

Server related generated files The server-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__server.h
- erpc__matrix__multiply__server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/



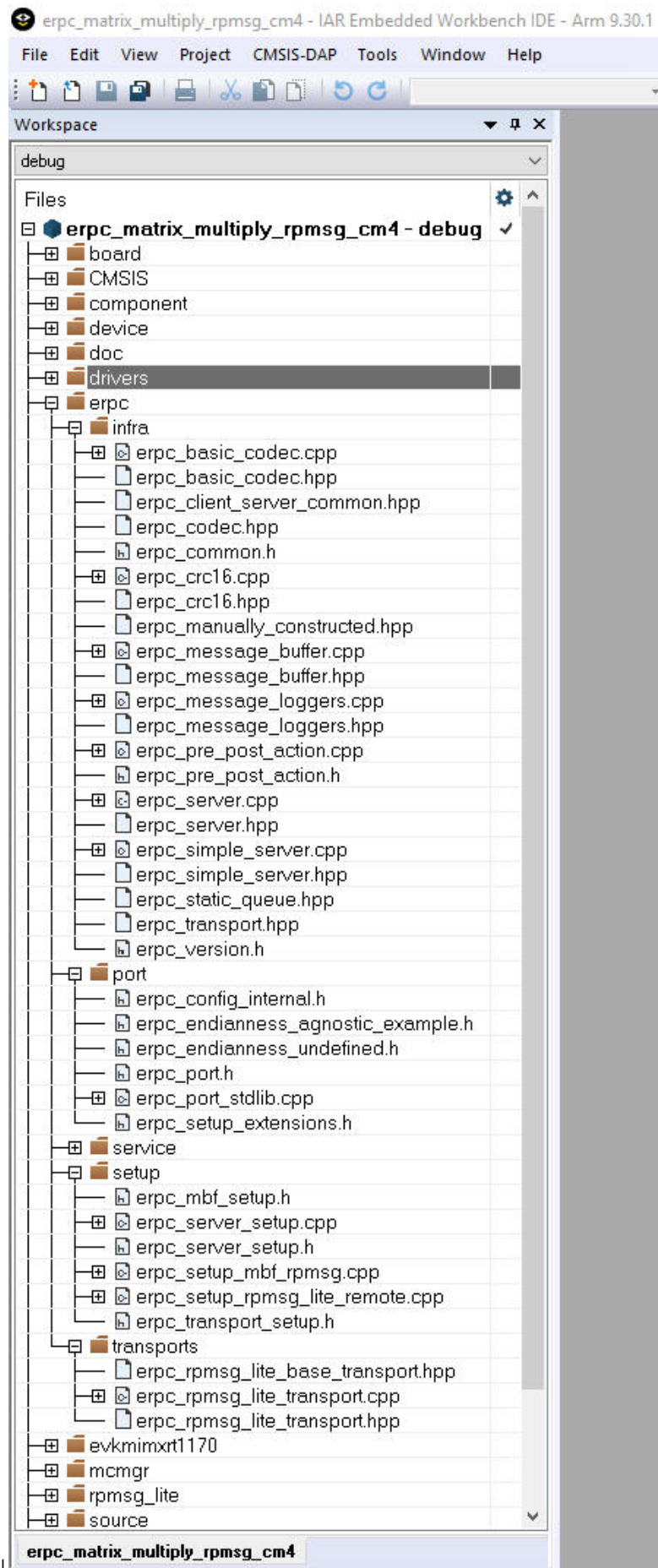
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
 - Four files, `erpc_server.hpp`, `erpc_server.cpp`, `erpc_simple_server.hpp`, and `erpc_simple_server.cpp`, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
 - Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
 - The `erpc_common.hpp` file is used for common eRPC definitions, typedefs, and enums.
 - The `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
 - Message buffer files are used for storing serialized data: `erpc_message_buffer.h` and `erpc_message_buffer.cpp`.
 - The `erpc_transport.h` file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
 - `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
 - `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
 - `erpc_config_internal.h` internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
 - The `erpc_server_setup.h` and `erpc_server_setup.cpp` files need to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
 - The `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_remote.cpp` files need to be added into the project in order to allow the C-wrapped function for transport layer setup.
 - The `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files need to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
 - RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files need to be added into the server project.



|

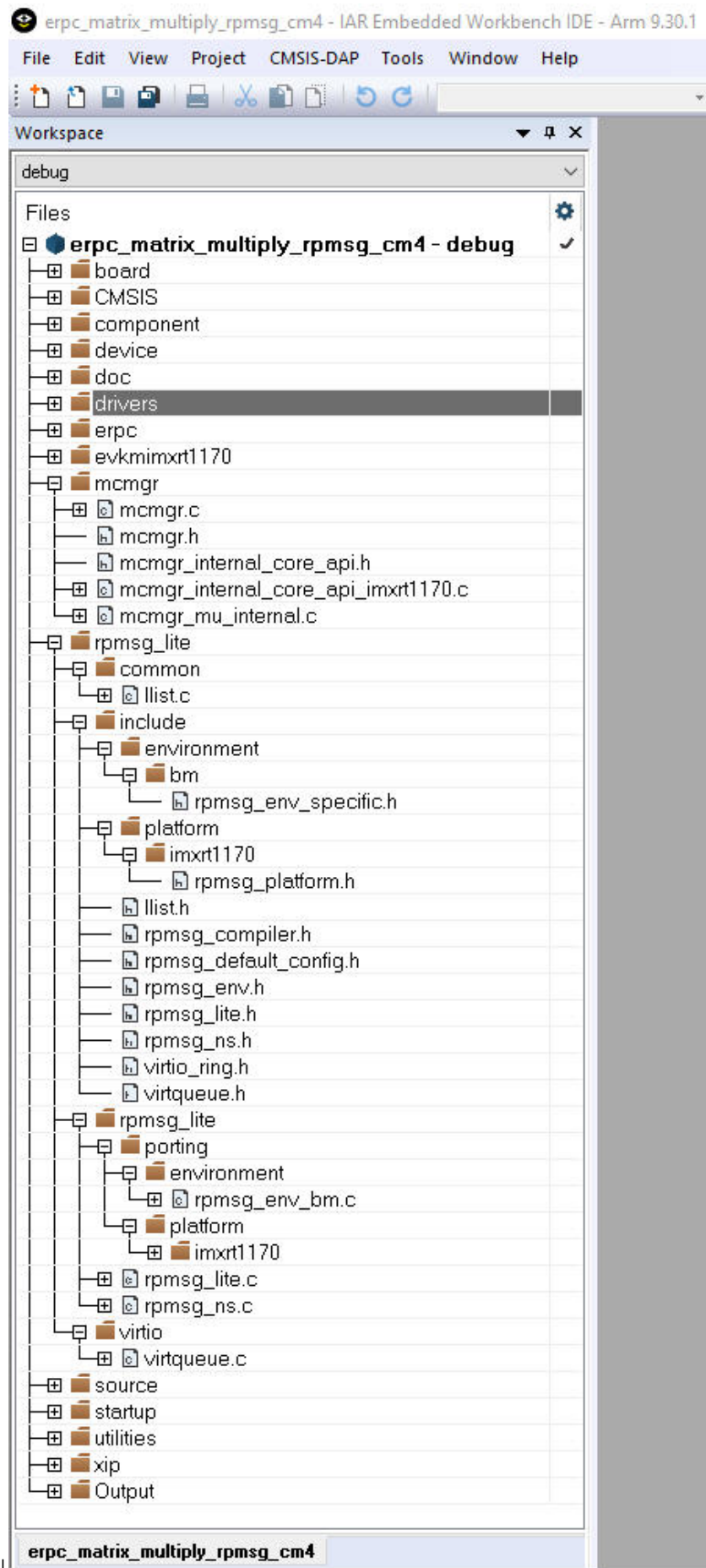
Parent topic: Multicore server application

Server multicore infrastructure files Because of the RPSMsg-Lite (transport layer), it is also necessary to include RPSMsg-Lite related files, which are in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpsmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic: Multicore server application

Server user code The server's user code is stored in the `main_core1.c` file, located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4`

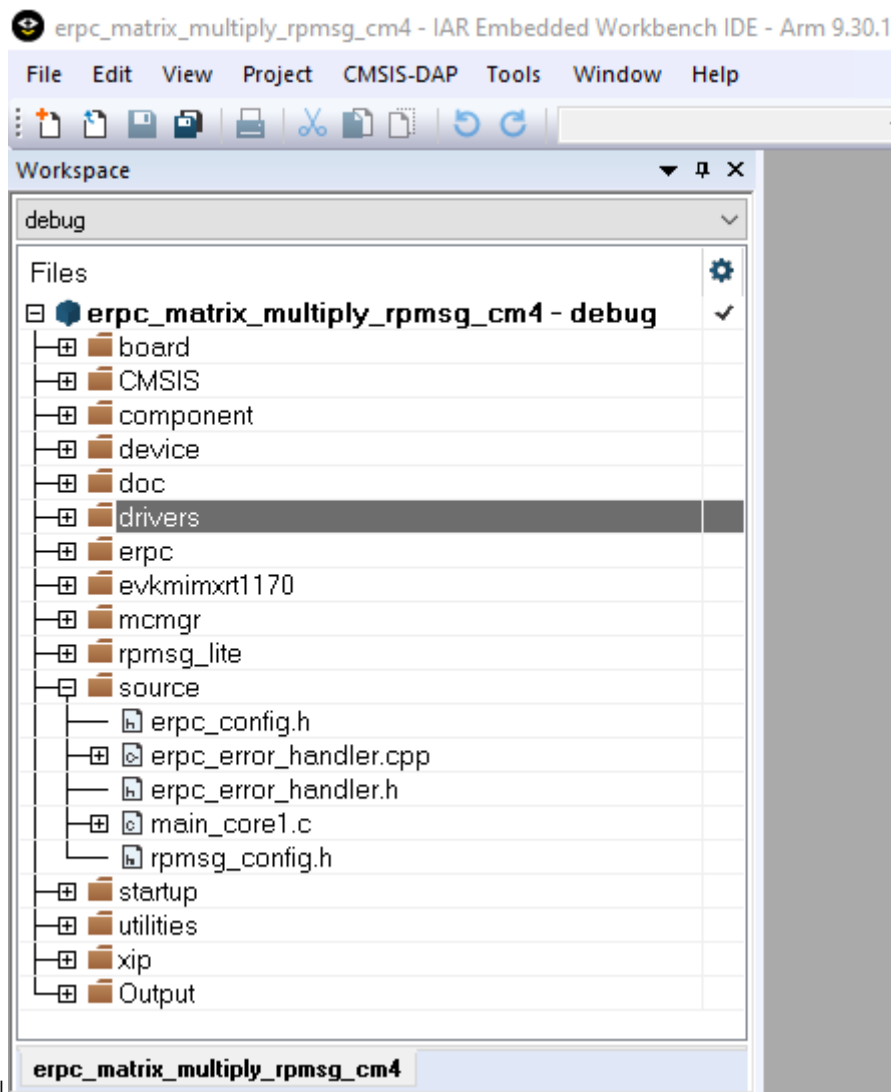
The `main_core1.c` file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the `erpc_error_handler.h` and `erpc_error_handler.cpp` files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
    ...
}
int main()
{
    ...
    /* RPSMsg-Lite transport layer initialization */
    erpc_transport_t transport;
    transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
    ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_rpmsg_init(transport);
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server);
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg` folder.



Parent topic:Multicore server application

Parent topic:[Create an eRPC application](#)

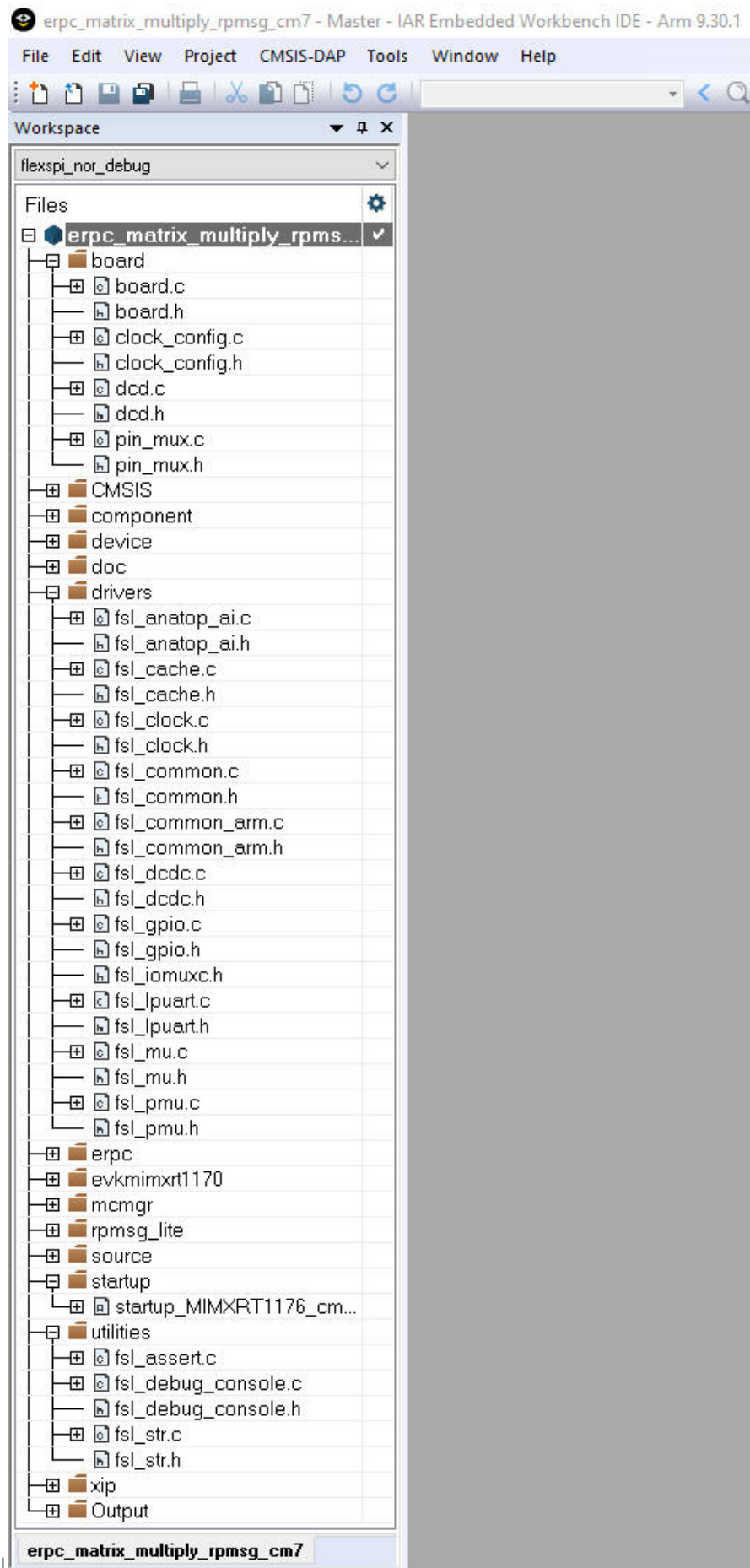
Multicore client application The “Matrix multiply” eRPC client project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar`

Project files for the eRPC client have the `_cm7` suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



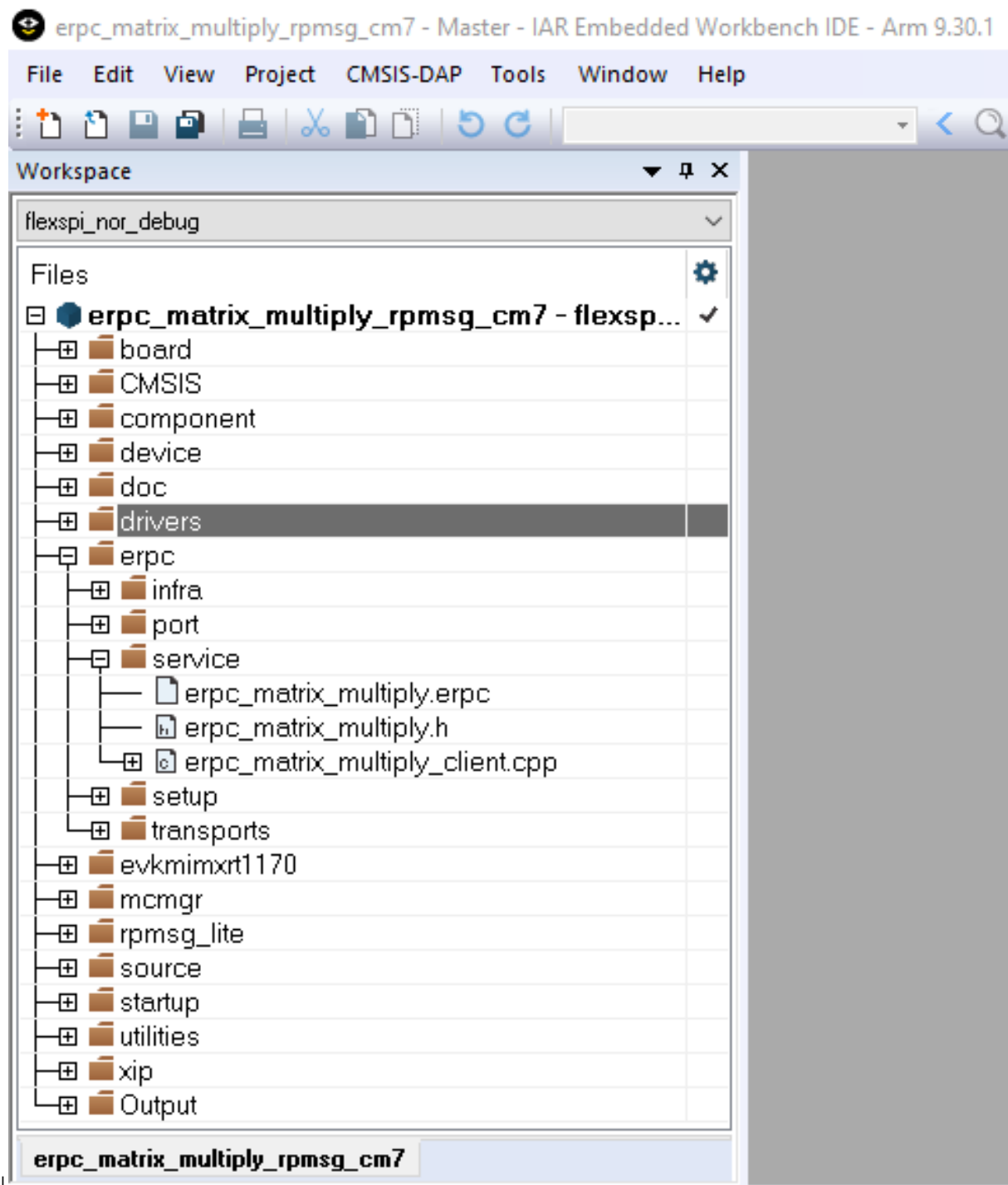
|

Parent topic: Multicore client application

Client-related generated files The client-related generated files are:

- `erpc__matric__multiply.h`
- `erpc__matrix__multiply__client.cpp`

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/` folder.



Parent topic: Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, `erpc_client_manager.h` and `erpc_client_manager.cpp`, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- `erpc_common.h` file is used for common eRPC definitions, typedefs, and enums.
- `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: `erpc_message_buffer.hpp` and `erpc_message_buffer.cpp`.
- `erpc_transport.hpp` file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

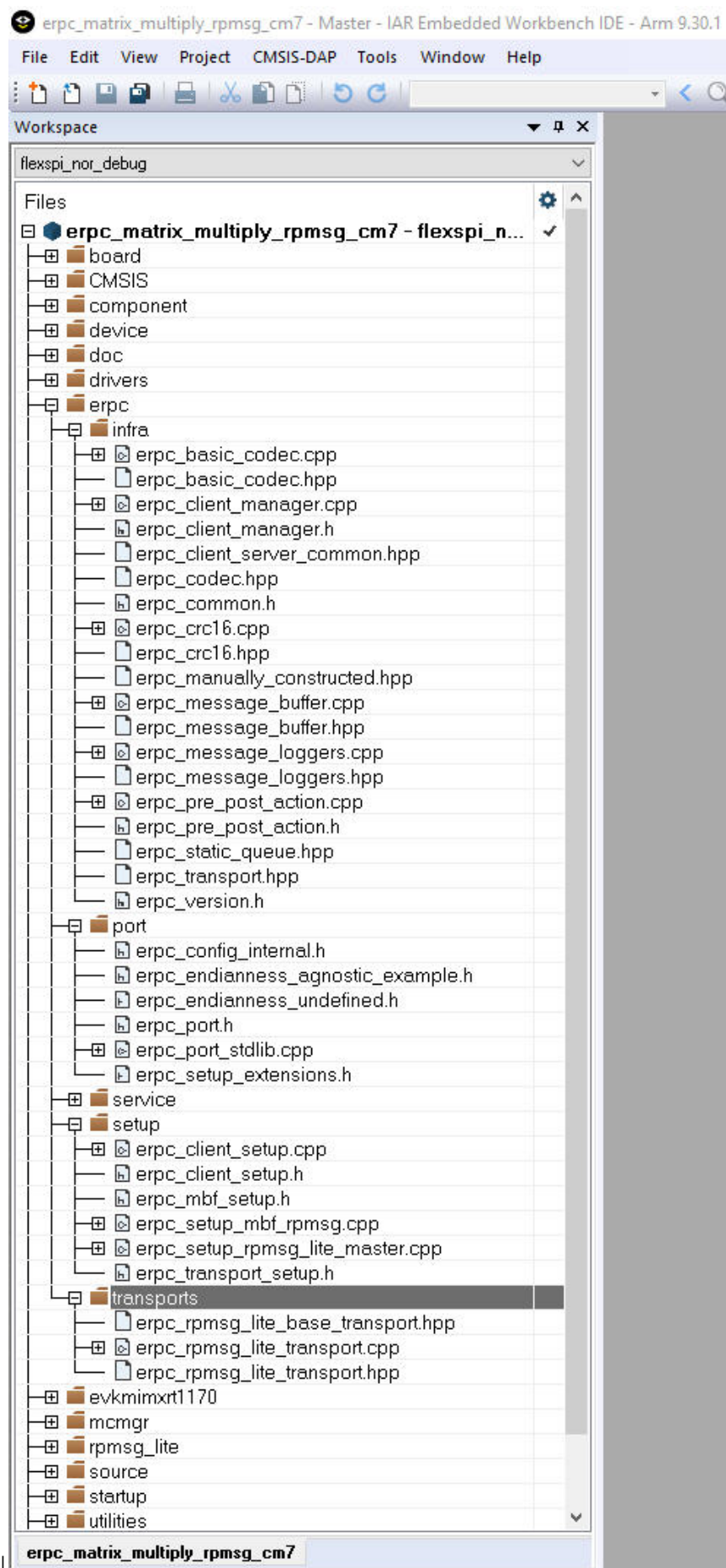
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- `erpc_client_setup.h` and `erpc_client_setup.cpp` files needs to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
- `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_master.cpp` files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files needs to be added into the client project.



|

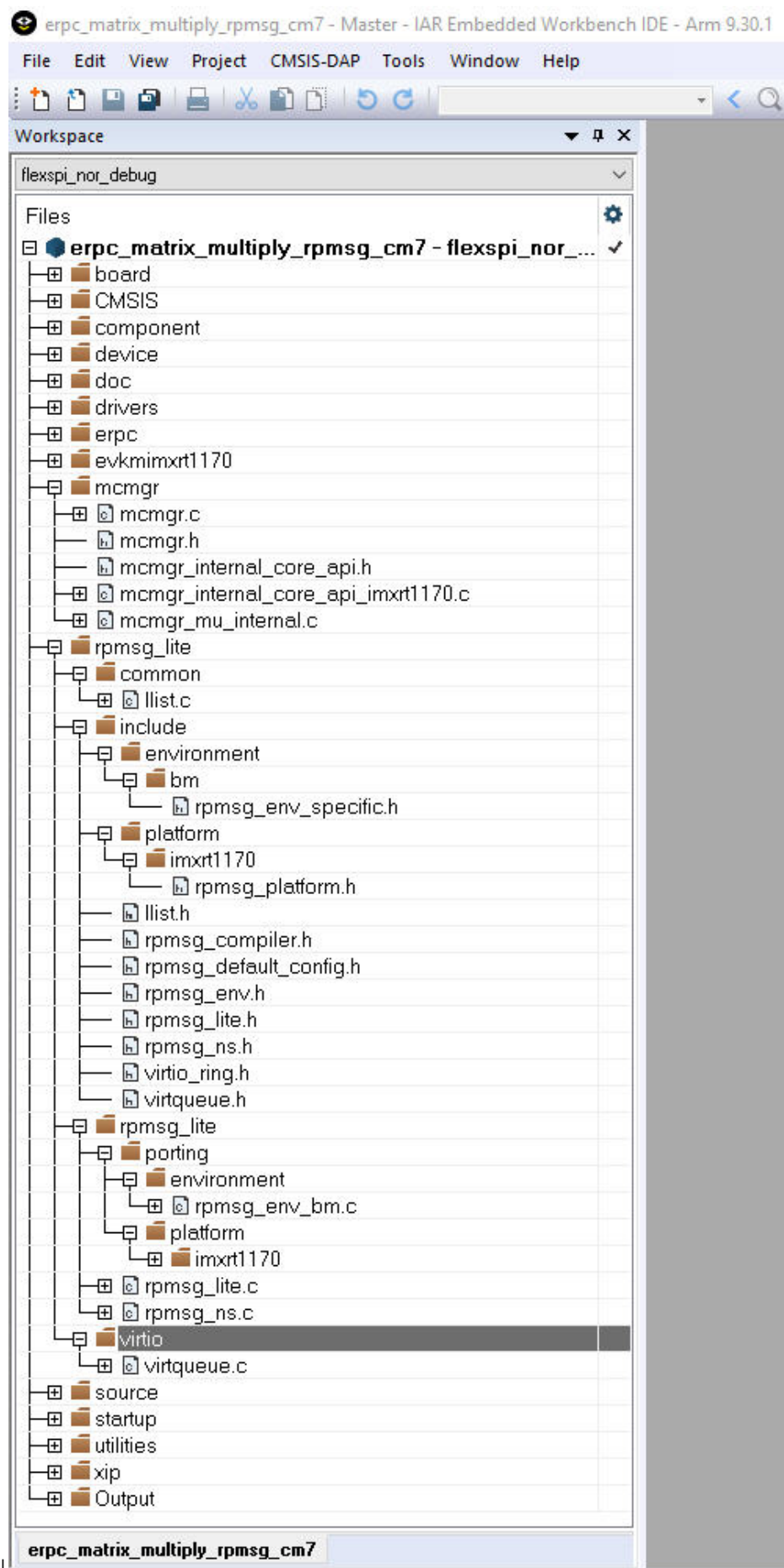
Parent topic: Multicore client application

Client multicore infrastructure files Because of the RPMsg-Lite (transport layer), it is also necessary to include RPMsg-Lite related files, which are in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/



Parent topic:Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

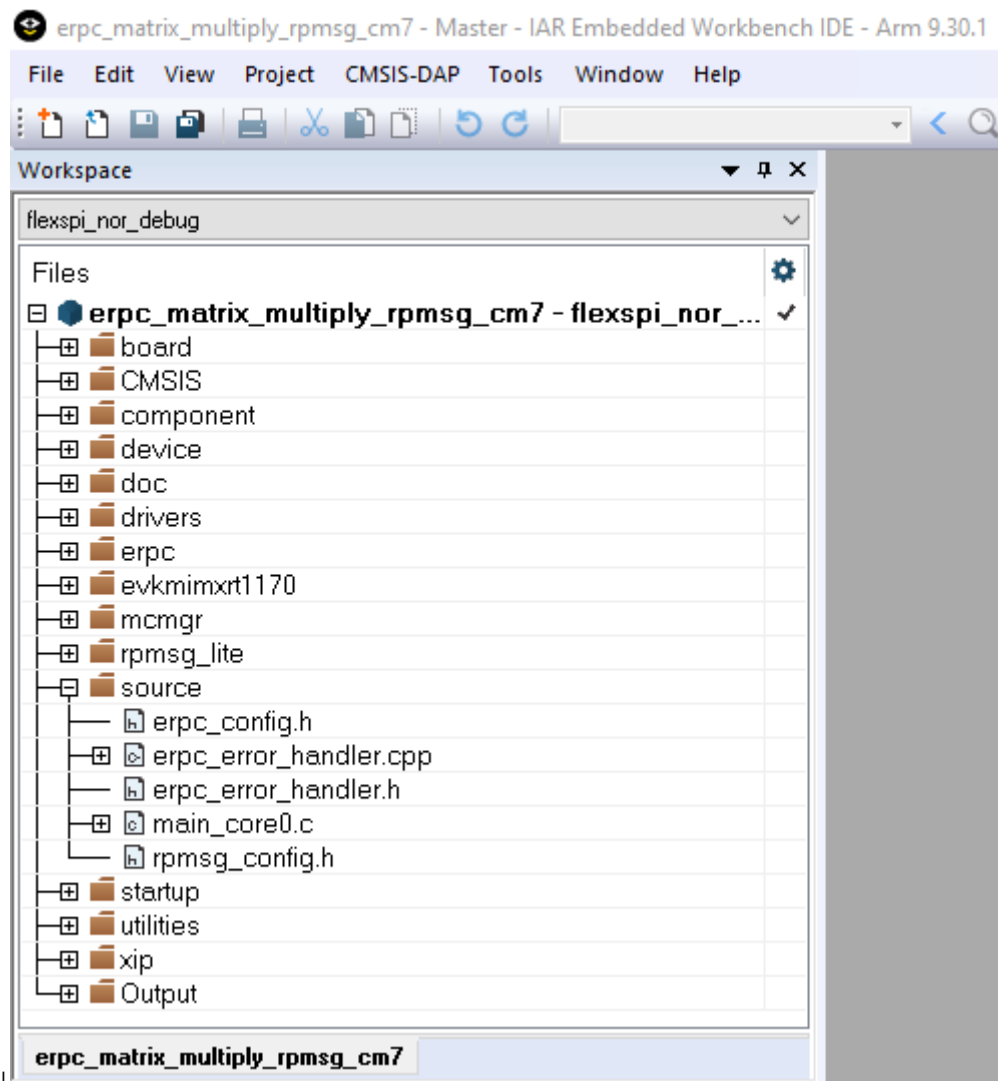
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



Parent topic:Multicore client application

Parent topic:[Create an eRPC application](#)

Multiprocessor server application The “Matrix multiply” eRPC server project for multiprocessor applications is located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>` folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMs-Lite). The RPMs-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/ folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
    ...
}
int main()
{
    ...
    /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
    ↪operations */
    erpc_transport_t transport;
    transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_dynamic_init();
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server)
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Parent topic:Multiprocessor server application

Multiprocessor client application The “Matrix multiply” eRPC client project for multiprocessor applications is located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/ folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPSMsg-Lite). The RPSMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.hpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.hpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.cpp

Client user code The client's user code is stored in the `main_client.c` file, located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/` folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

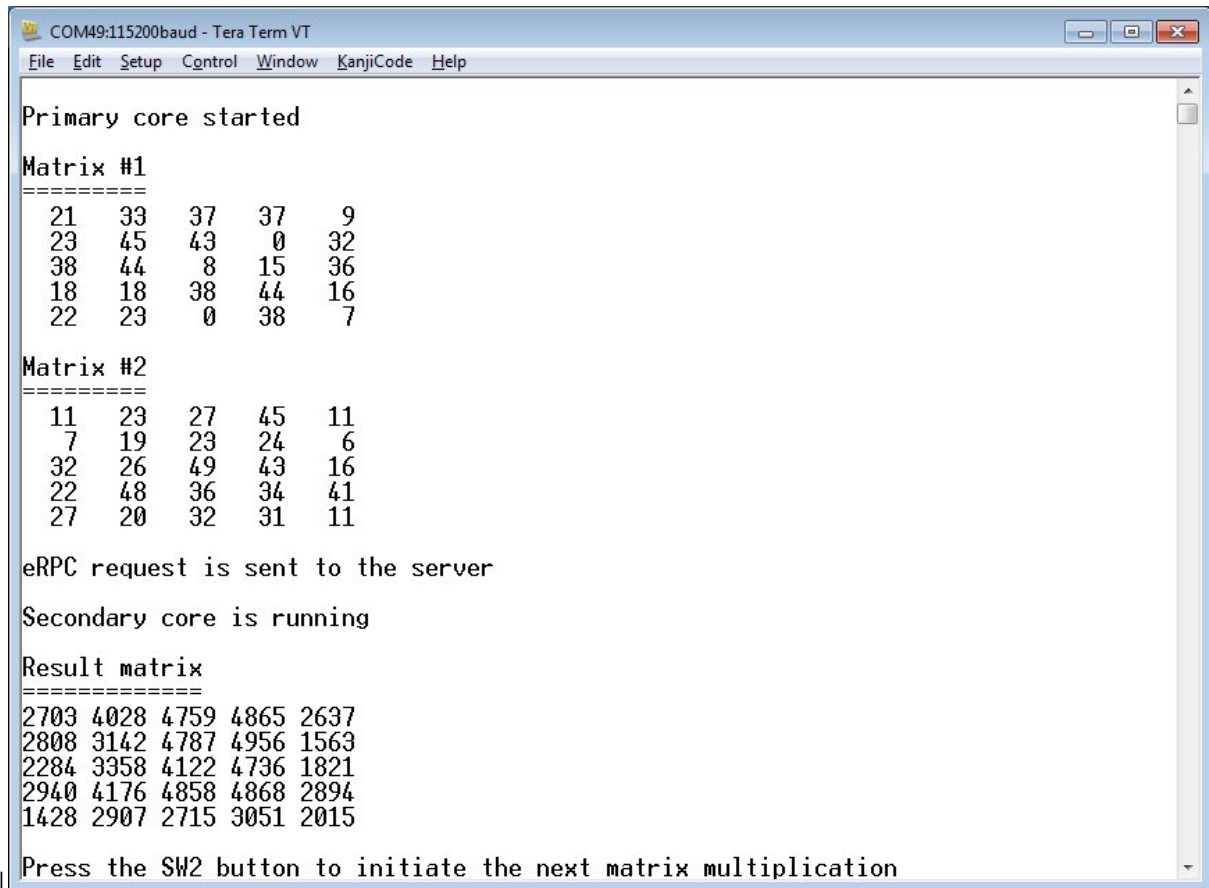
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Parent topic:Multiprocessor client application

Parent topic:Multiprocessor server application

Parent topic:[Create an eRPC application](#)

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the <MCUXpressoSDK_install_dir>/docs folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



```

COM49:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

Primary core started

Matrix #1
=====
 21  33  37  37   9
 23  45  43   0  32
 38  44   8  15  36
 18  18  38  44  16
 22  23   0  38   7

Matrix #2
=====
 11  23  27  45  11
  7  19  23  24   6
 32  26  49  43  16
 22  48  36  34  41
 27  20  32  31  11

eRPC request is sent to the server

Secondary core is running

Result matrix
=====
2703 4028 4759 4865 2637
2808 3142 4787 4956 1563
2284 3358 4122 4736 1821
2940 4176 4858 4868 2894
1428 2907 2715 3051 2015

Press the SW2 button to initiate the next matrix multiplication

```

For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:[Create an eRPC application](#)

Parent topic:[eRPC example](#)

eRPC example This section shows how to create an example eRPC application called “Matrix multiply”, which implements one eRPC function (matrix multiply) with two function parameters (two matrices). The client-side application calls this eRPC function, and the server side performs the multiplication of received matrices. The server side then returns the result.

For example, use the NXP MIMXRT1170-EVK board as the target dual-core platform, and the IAR Embedded Workbench for ARM (EWARM) as the target IDE for developing the eRPC example.

- The primary core (CM7) runs the eRPC client.
- The secondary core (CM4) runs the eRPC server.
- RPMsg-Lite (Remote Processor Messaging Lite) is used as the eRPC transport layer.

The “Matrix multiply” application can be also run in the multi-processor setup. In other words, the eRPC client running on one SoC communicates with the eRPC server that runs on another SoC, utilizing different transport channels. It is possible to run the board-to-PC example (PC as the eRPC server and a board as the eRPC client, and vice versa) and also the board-to-board example. These multiprocessor examples are prepared for selected boards only.

| Multicore application source and project files | `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore/`
| Multiprocessor application source and project files | `<MCUXpressoSDK_install_dir>/boards/<board_name>/multi`
`<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<tr`
| | eRPC source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/` | | RPLite
source files | `<MCUXpressoSDK_install_dir>/middleware/multicore/rplite/`

Designing the eRPC application The matrix multiply application is based on calling single eRPC function that takes 2 two-dimensional arrays as input and returns matrix multiplication results as another 2 two-dimensional array. The IDL file syntax supports arrays with the dimension length set by the number only (in the current eRPC implementation). Because of this, a variable is declared in the IDL dedicated to store information about matrix dimension length, and to allow easy maintenance of the user and server code.

For a simple use of the two-dimensional array, the alias name (new type definition) for this data type has been declared in the IDL. Declaring this alias name ensures that the same data type can be used across the client and server applications.

Parent topic: [eRPC example](#)

Creating the IDL file The created IDL file is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/`

The created IDL file contains the following code:

```
program erpc_matrix_multiply
/*! This const defines the matrix size. The value has to be the same as the
Matrix array dimension. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
const int32 matrix_size = 5;
/*! This is the matrix array type. The dimension has to be the same as the
matrix size const. Do not forget to re-generate the erpc code once the
matrix size is changed in the erpc file */
type Matrix = int32[matrix_size][matrix_size];
interface MatrixMultiplyService {
erpcMatrixMultiply(in Matrix matrix1, in Matrix matrix2, out Matrix result_matrix) ->
void
}
```

Details:

- The IDL file starts with the program name (*erpc_matrix_multiply*), and this program name is used in the naming of all generated outputs.
- The declaration and definition of the constant variable named *matrix_size* follows next. The *matrix_size* variable is used for passing information about the length of matrix dimensions to the client/server user code.
- The alias name for the two-dimensional array type (*Matrix*) is declared.
- The interface group *MatrixMultiplyService* is located at the end of the IDL file. This interface group contains only one function declaration *erpcMatrixMultiply*.
- As shown above, the function’s declaration contains three parameters of *Matrix* type: *matrix1* and *matrix2* are input parameters, while *result_matrix* is the output parameter. Additionally, the returned data type is declared as *void*.

When writing the IDL file, the following order of items is recommended:

1. Program name at the top of the IDL file.
2. New data types and constants declarations.
3. Declarations of interfaces and functions at the end of the IDL file.

Parent topic: [eRPC example](#)

Using the eRPC generator tool | Windows OS | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x64`
 | Linux OS | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86`
 | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Linux_x86`
 | | Mac OS | `<MCUXpressoSDK_install_dir>/middleware/multicore/tools/erpcgen/Mac`

The files for the “Matrix multiply” example are pre-generated and already a part of the application projects. The following section describes how they have been created.

- The easiest way to create the shim code is to copy the erpcgen application to the same folder where the IDL file (*.erpc) is located; then run the following command:

```
erpcgen <IDL_file>.erpc
```

- In the “Matrix multiply” example, the command should look like:

```
erpcgen erpc_matrix_multiply.erpc
```

Additionally, another method to create the shim code is to execute the eRPC application using input commands:

- “-?”/”—help” – Shows supported commands.
- “-o <filePath>”/”—output<filePath>” – Sets the output directory.

For example,

```
<path_to_erpcgen>/erpcgen -o <path_to_output>  
<path_to_IDL>/<IDL_file_name>.erpc
```

For the “Matrix multiply” example, when the command is executed from the default erpcgen location, it looks like:

```
erpcgen -o
```

```
.././.././boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service  
.././.././boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/erpc_matrix_mu
```

In both cases, the following four files are generated into the `<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service` folder:

- erpc_matrix_multiply.h
- erpc_matrix_multiply_client.cpp
- erpc_matrix_multiply_server.h
- erpc_matrix_multiply_server.cpp

For multiprocessor examples, the eRPC file and pre-generated files can be found in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_common/erpc_matrix_multiply/service` folder.

For Linux OS users:

- Do not forget to set the permissions for the eRPC generator application.
- Run the application as `./erpcgen...` instead of as `erpcgen ....`

Parent topic: [eRPC example](#)

Create an eRPC application This section describes a generic way to create a client/server eRPC application:

1. **Design the eRPC application:** Decide which data types are sent between applications, and define functions that send/receive this data.
2. **Create the IDL file:** The IDL file contains information about data types and functions used in an eRPC application, and is written in the IDL language.
3. **Use the eRPC generator tool:** This tool takes an IDL file and generates the shim code for the client and the server-side applications.
4. **Create an eRPC application:**
 1. Create two projects, where one project is for the client side (primary core) and the other project is for the server side (secondary core).
 2. Add generated files for the client application to the client project, and add generated files for the server application to the server project.
 3. Add infrastructure files.
 4. Add user code for client and server applications.
 5. Set the client and server project options.
5. **Run the eRPC application:** Run both the server and the client applications. Make sure that the server has been run before the client request was sent.

A specific example follows in the next section.

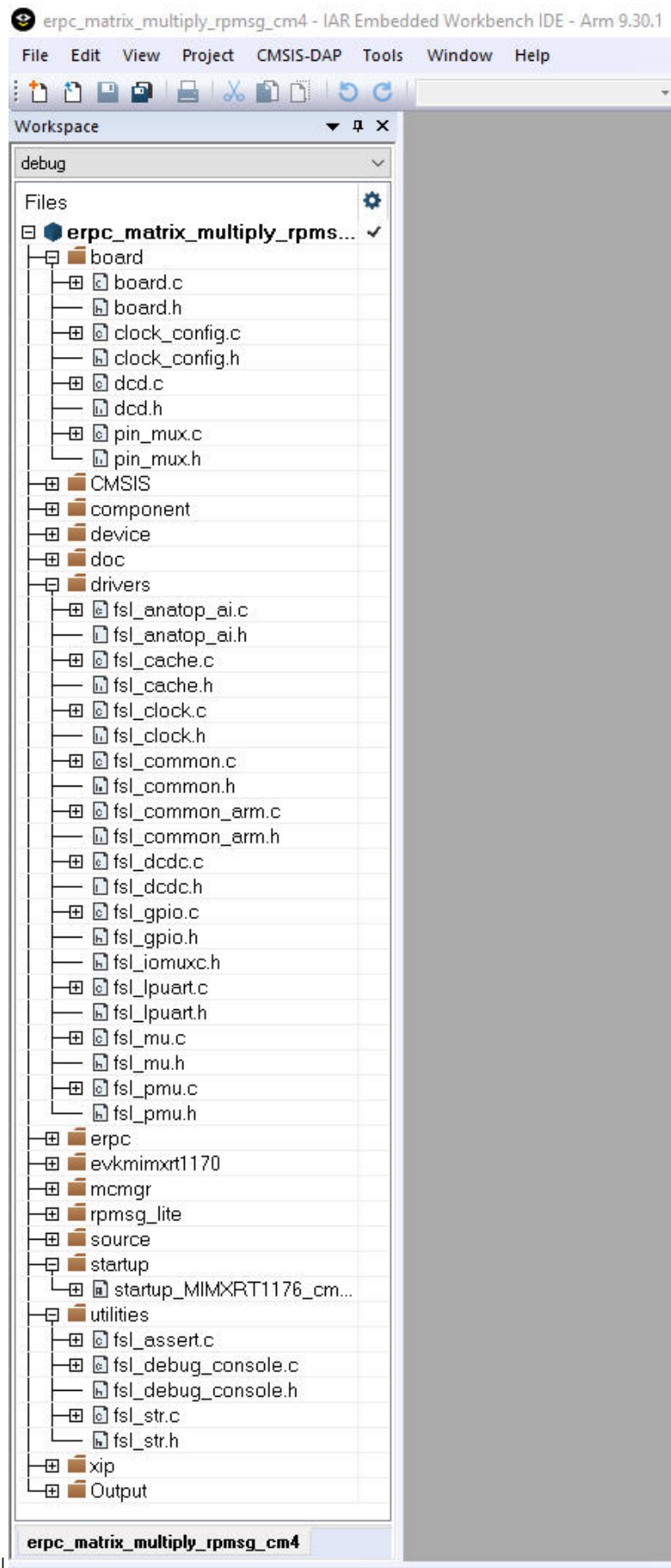
Multicore server application The “Matrix multiply” eRPC server project is located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmmsg/cm4/iar/`

The project files for the eRPC server have the `_cm4` suffix.

Server project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in:

- `<MCUXpressoSDK_install_dir>/devices/<device>`
- `<MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/`



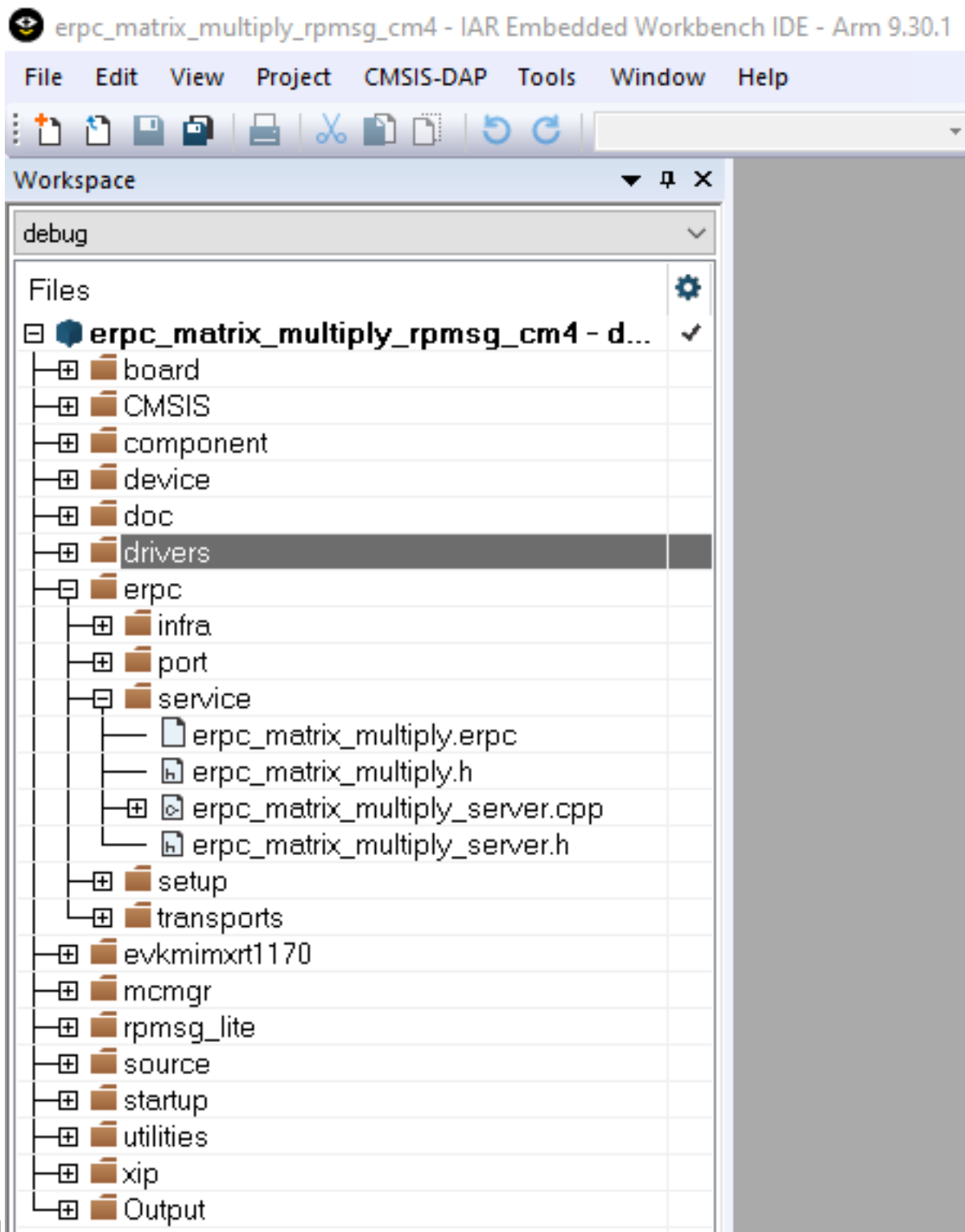
|

Parent topic: Multicore server application**Server related generated files** The server-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__server.h
- erpc__matrix__multiply__server.cpp

The server-related generated files contain the shim code for functions and data types declared in the IDL file. These files also contain functions for the identification of client requested functions, data deserialization, calling requested function's implementations, and data serialization and return, if requested by the client. These shim code files can be found in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/



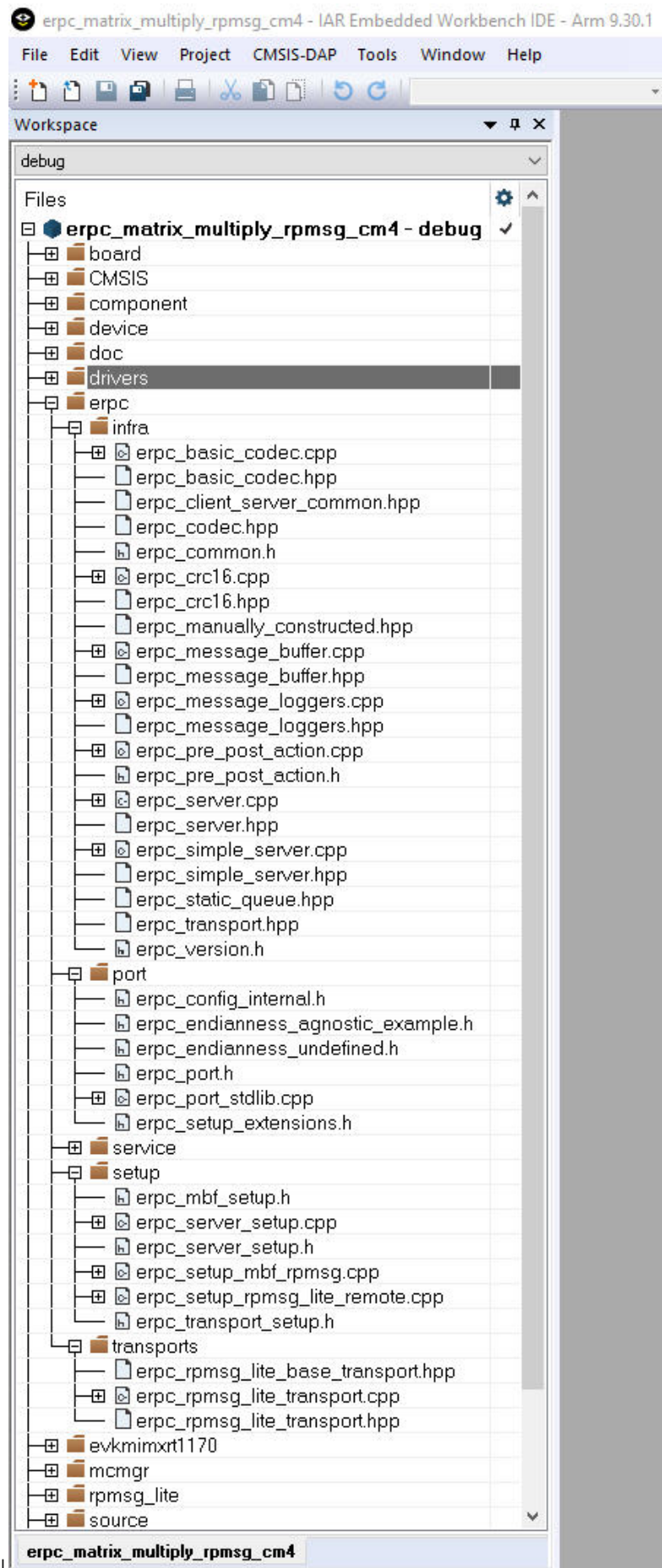
Parent topic: Multicore server application

Server infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.
 - Four files, `erpc_server.hpp`, `erpc_server.cpp`, `erpc_simple_server.hpp`, and `erpc_simple_server.cpp`, are used for running the eRPC server on the server-side applications. The simple server is currently the only implementation of the server, and its role is to catch client requests, identify and call requested functions, and send data back when requested.
 - Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
 - The `erpc_common.hpp` file is used for common eRPC definitions, typedefs, and enums.
 - The `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
 - Message buffer files are used for storing serialized data: `erpc_message_buffer.h` and `erpc_message_buffer.cpp`.
 - The `erpc_transport.h` file defines the abstract interface for transport layer.
- The **port** subfolder contains the eRPC porting layer to adapt to different environments.
 - `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
 - `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
 - `erpc_config_internal.h` internal erpc configuration file.
- The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.
 - The `erpc_server_setup.h` and `erpc_server_setup.cpp` files need to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
 - The `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_remote.cpp` files need to be added into the project in order to allow the C-wrapped function for transport layer setup.
 - The `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files need to be added into the project in order to allow message buffer factory usage.
- The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions in the setup folder.
 - RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files need to be added into the server project.



|

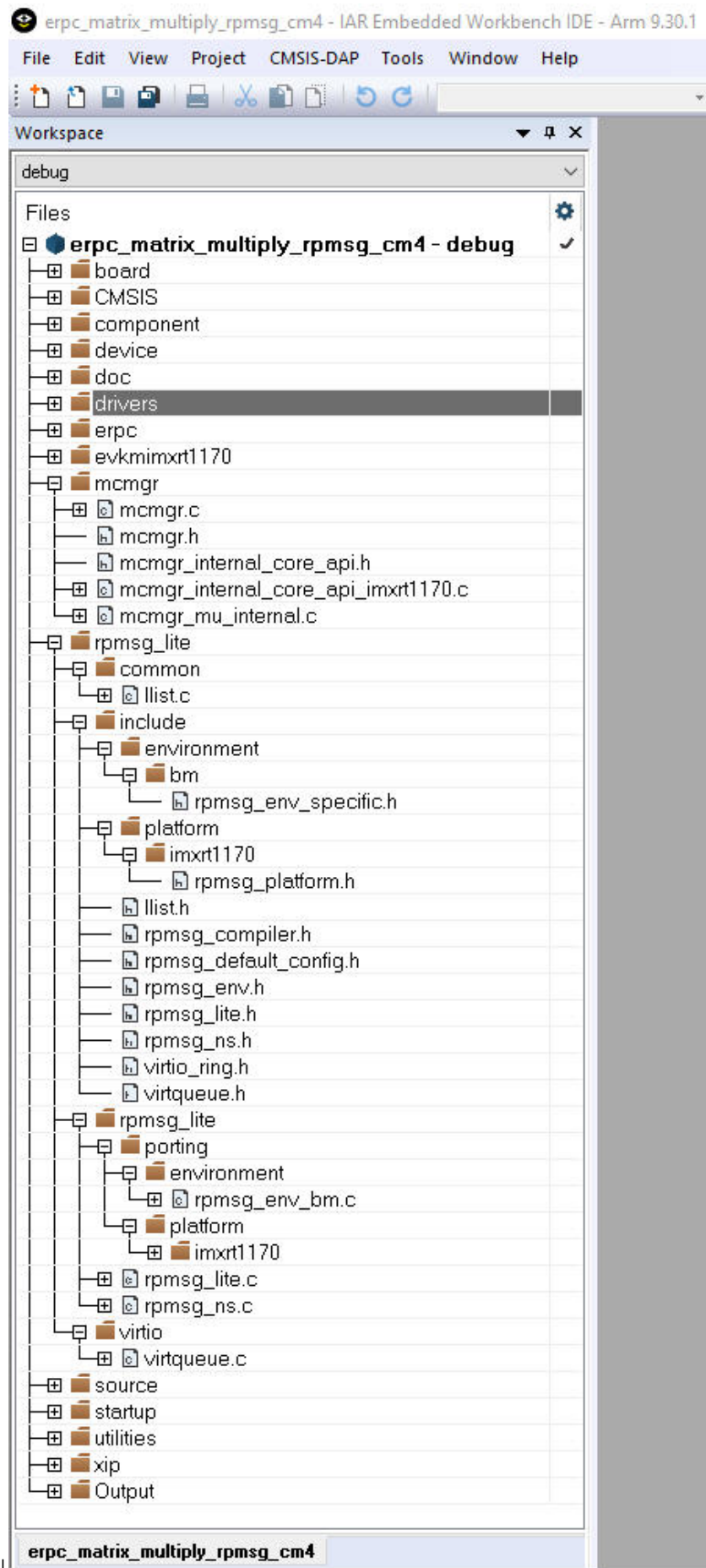
Parent topic: Multicore server application

Server multicore infrastructure files Because of the RPSMsg-Lite (transport layer), it is also necessary to include RPSMsg-Lite related files, which are in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/rpsmsg_lite/`

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/`



|

Parent topic: Multicore server application

Server user code The server's user code is stored in the `main_core1.c` file, located in the following folder:

`<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm4`

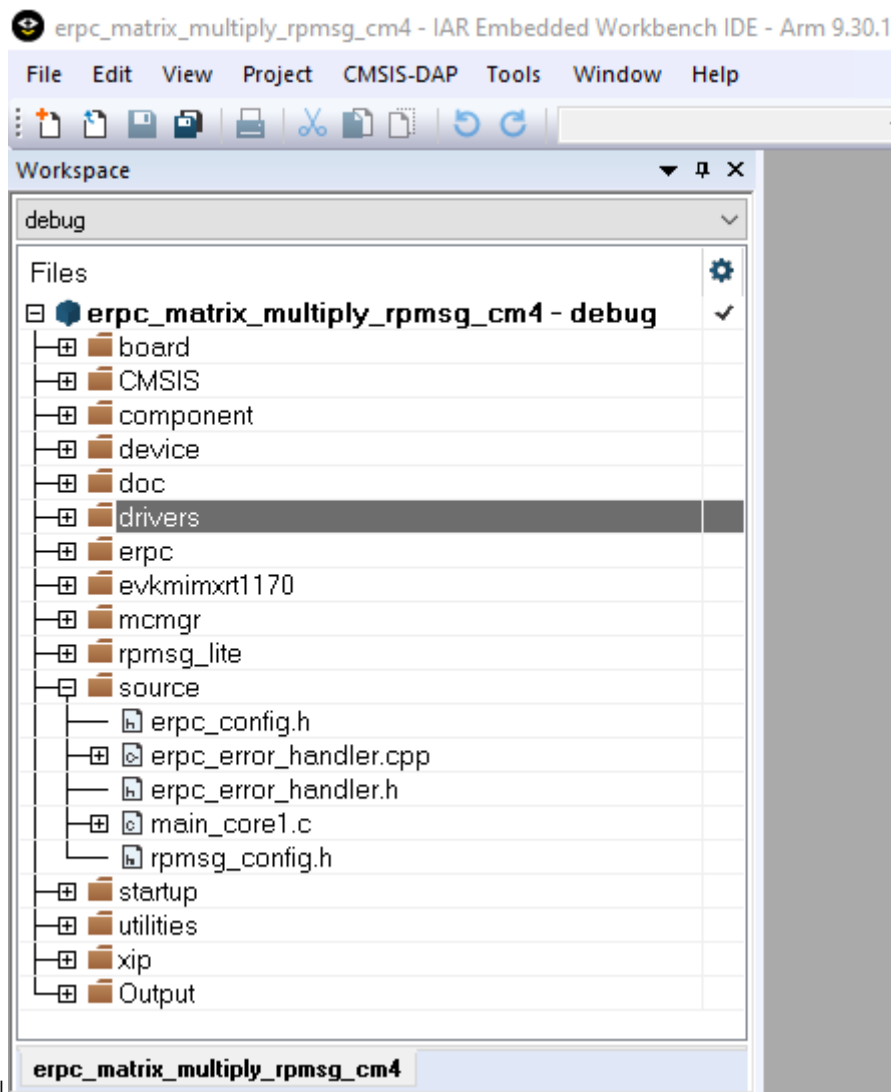
The `main_core1.c` file contains two functions:

- The **main()** function contains the code for the target board and eRPC server initialization. After the initialization, the matrix multiply service is added and the eRPC server waits for client's requests in the while loop.
- The **erpcMatrixMultiply()** function is the user implementation of the eRPC function defined in the IDL file.
- There is the possibility to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in the `erpc_error_handler.h` and `erpc_error_handler.cpp` files.

The eRPC-relevant code is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(const Matrix *matrix1, const Matrix *matrix2, Matrix *result_matrix)
{
    ...
}
int main()
{
    ...
    /* RPSMsg-Lite transport layer initialization */
    erpc_transport_t transport;
    transport = erpc_transport_rpmsg_lite_remote_init(src, dst, (void*)startupData,
    ERPC_TRANSPORT_RPMSG_LITE_LINK_ID, SignalReady, NULL);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_rpmsg_init(transport);
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server);
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Except for the application main file, there are configuration files for the RPMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the <MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/ erpc_matrix_multiply_rpmsg folder.



Parent topic:Multicore server application

Parent topic:[Create an eRPC application](#)

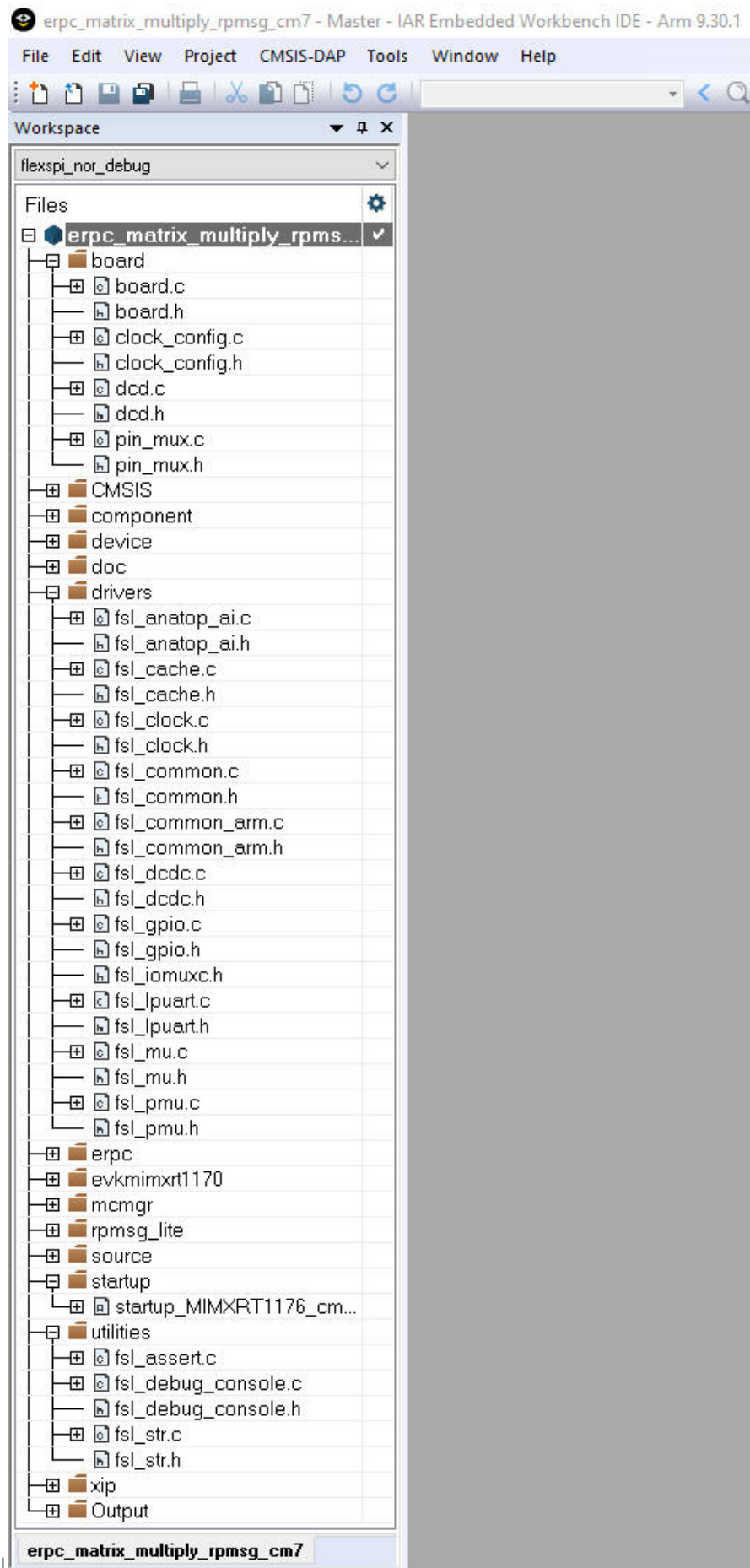
Multicore client application The “Matrix multiply” eRPC client project is located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg/cm7/iar

Project files for the eRPC client have the _cm7 suffix.

Client project basic source files The startup files, board-related settings, peripheral drivers, and utilities belong to the basic project source files and form the skeleton of all MCUXpresso SDK applications. These source files are located in the following folders:

- <MCUXpressoSDK_install_dir>/devices/<device>
- <MCUXpressoSDK_install_dir>/boards/<board_name>/multicore_examples/<example_name>/



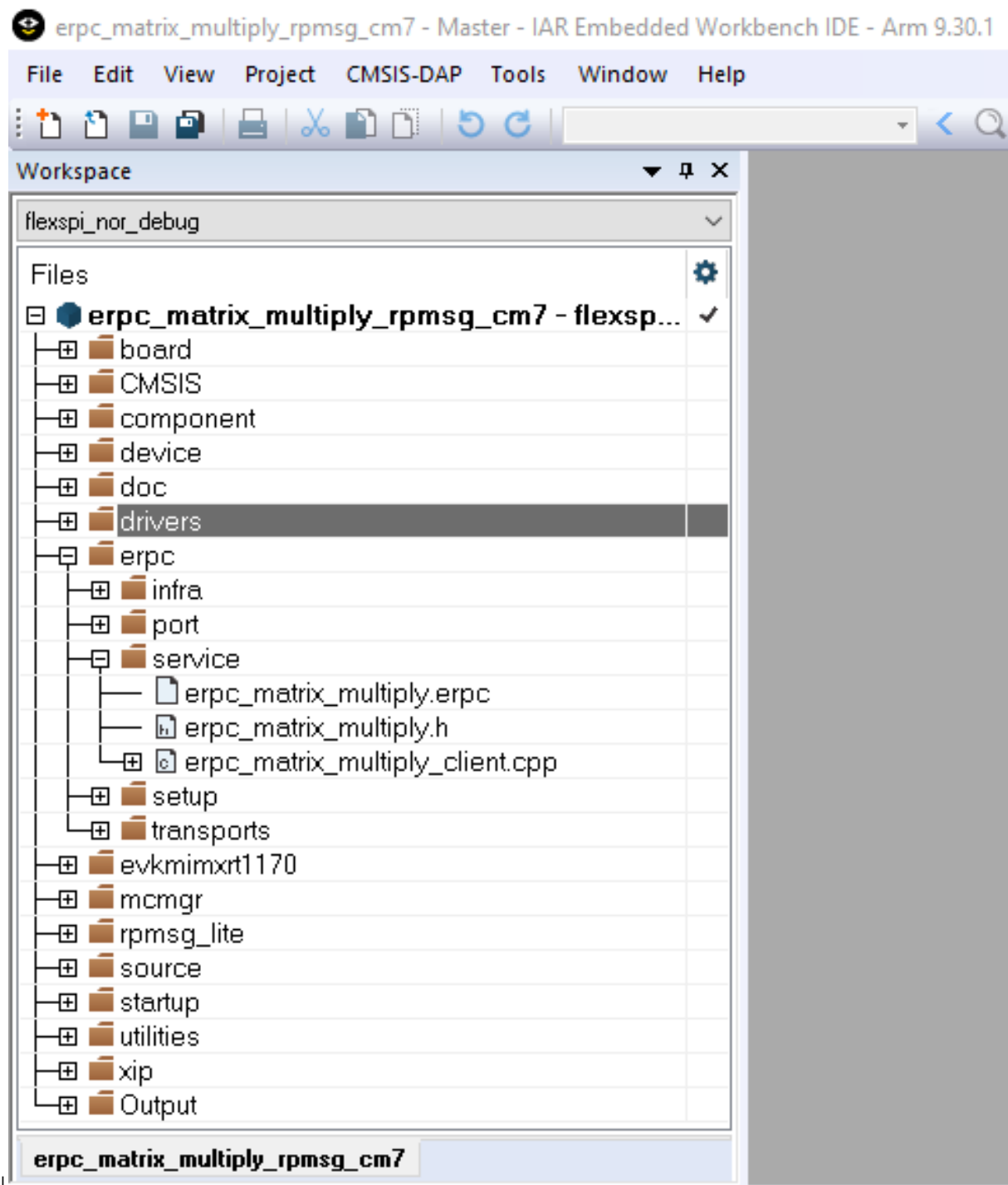
|

Parent topic:Multicore client application

Client-related generated files The client-related generated files are:

- erpc__matric__multiply.h
- erpc__matrix__multiply__client.cpp

These files contain the shim code for the functions and data types declared in the IDL file. These functions also call methods for codec initialization, data serialization, performing eRPC requests, and de-serializing outputs into expected data structures (if return values are expected). These shim code files can be found in the <MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_common/erpc_matrix_multiply/service/ folder.



Parent topic:Multicore client application

Client infrastructure files The eRPC infrastructure files are located in the following folder:

`<MCUXpressoSDK_install_dir>/middleware/multicore/erpc/erpc_c`

The **erpc_c** folder contains files for creating eRPC client and server applications in the C/C++ language. These files are distributed into subfolders.

- The **infra** subfolder contains C++ infrastructure code used to build server and client applications.

- Two files, `erpc_client_manager.h` and `erpc_client_manager.cpp`, are used for managing the client-side application. The main purpose of the client files is to create, perform, and release eRPC requests.
- Three files (`erpc_codec.hpp`, `erpc_basic_codec.hpp`, and `erpc_basic_codec.cpp`) are used for codecs. Currently, the basic codec is the initial and only implementation of the codecs.
- `erpc_common.h` file is used for common eRPC definitions, typedefs, and enums.
- `erpc_manually_constructed.hpp` file is used for allocating static storage for the used objects.
- Message buffer files are used for storing serialized data: `erpc_message_buffer.hpp` and `erpc_message_buffer.cpp`.
- `erpc_transport.hpp` file defines the abstract interface for transport layer.

The **port** subfolder contains the eRPC porting layer to adapt to different environments.

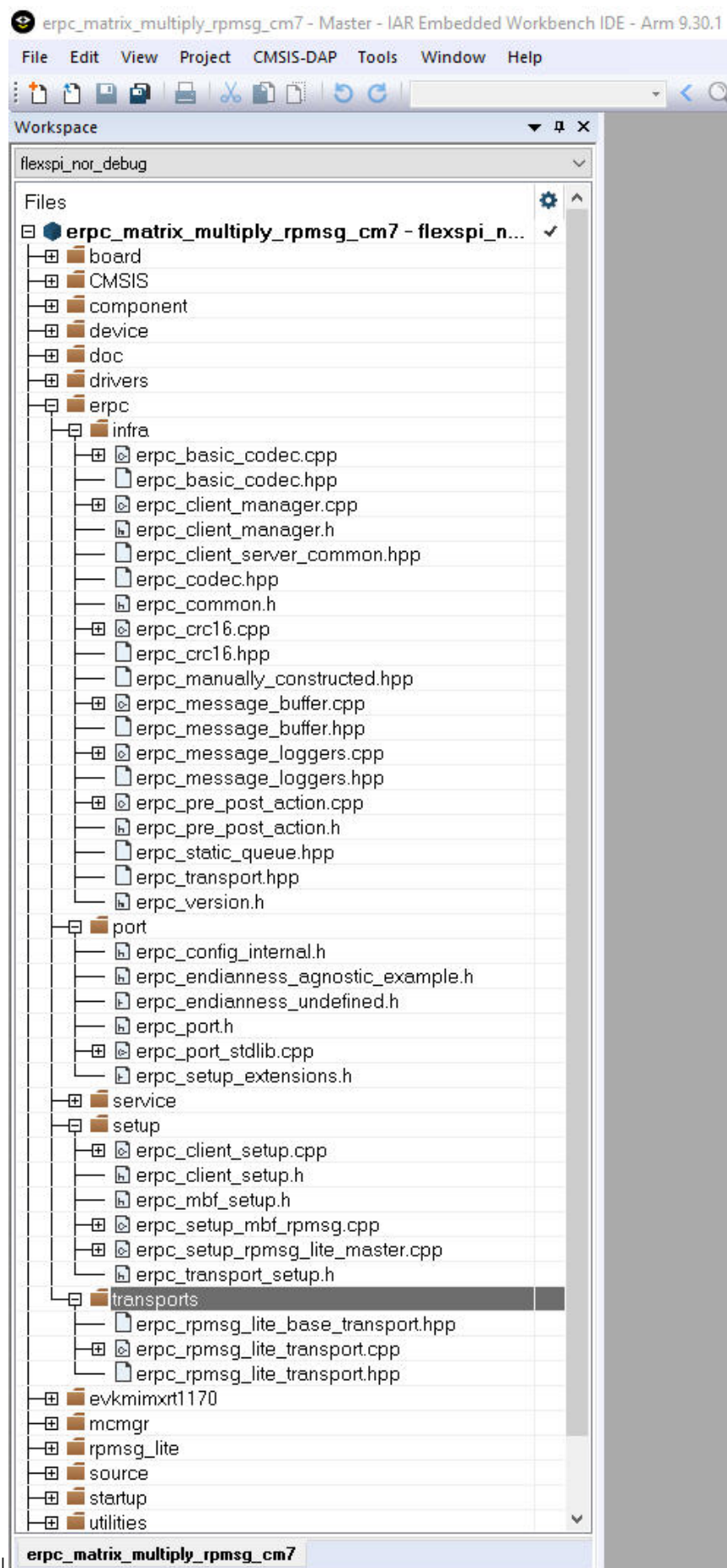
- `erpc_port.h` file contains definition of `erpc_malloc()` and `erpc_free()` functions.
- `erpc_port_stdlib.cpp` file ensures adaptation to `stdlib`.
- `erpc_config_internal.h` internal eRPC configuration file.

The **setup** subfolder contains a set of plain C APIs that wrap the C++ infrastructure, providing client and server init and deinit routines that greatly simplify eRPC usage in C-based projects. No knowledge of C++ is required to use these APIs.

- `erpc_client_setup.h` and `erpc_client_setup.cpp` files needs to be added into the “Matrix multiply” example project to demonstrate the use of C-wrapped functions in this example.
- `erpc_transport_setup.h` and `erpc_setup_rpmsg_lite_master.cpp` files needs to be added into the project in order to allow C-wrapped function for transport layer setup.
- `erpc_mbf_setup.h` and `erpc_setup_mbf_rpmsg.cpp` files needs to be added into the project in order to allow message buffer factory usage.

The **transports** subfolder contains transport classes for the different methods of communication supported by eRPC. Some transports are applicable only to host PCs, while others are applicable only to embedded or multicore systems. Most transports have corresponding client and server setup functions, in the setup folder.

- RPMsg-Lite is used as the transport layer for the communication between cores, `erpc_rpmsg_lite_base_transport.hpp`, `erpc_rpmsg_lite_transport.hpp`, and `erpc_rpmsg_lite_transport.cpp` files needs to be added into the client project.



|

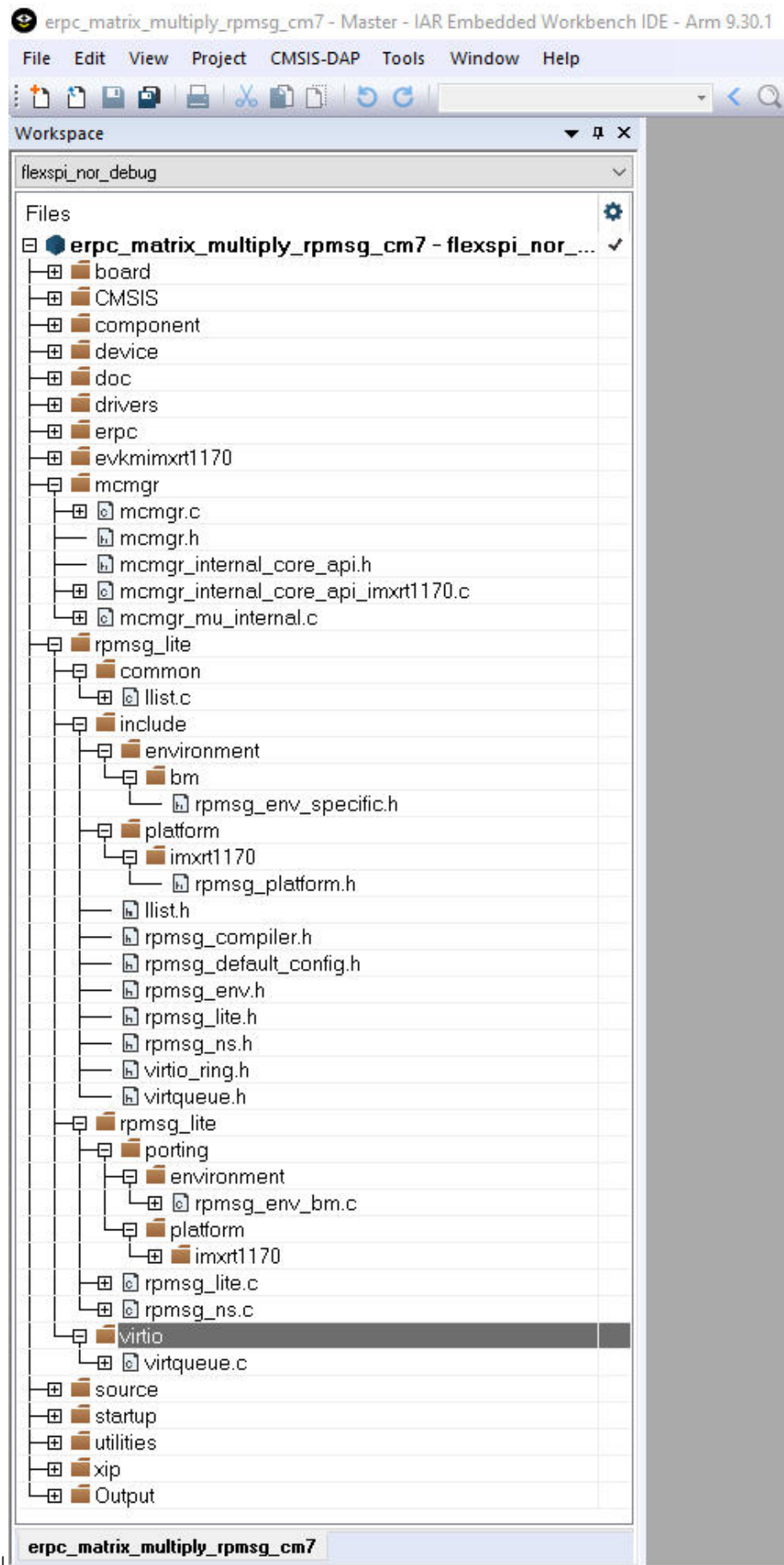
Parent topic: Multicore client application

Client multicore infrastructure files Because of the RPMsg-Lite (transport layer), it is also necessary to include RPMsg-Lite related files, which are in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/rpmsg_lite/

The multicore example applications also use the Multicore Manager software library to control the secondary core startup and shutdown. These source files are located in the following folder:

<MCUXpressoSDK_install_dir>/middleware/multicore/mcmgr/



Parent topic:Multicore client application

Client user code The client's user code is stored in the main_core0.c file, located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_example/erpc_matrix_multiply_rpmsg/cm7

The main_core0.c file contains the code for target board and eRPC initialization.

- After initialization, the secondary core is released from reset.
- When the secondary core is ready, the primary core initializes two matrix variables.
- The erpcMatrixMultiply eRPC function is called to issue the eRPC request and get the result.

It is possible to write the application-specific eRPC error handler. The eRPC error handler of the matrix multiply application is implemented in erpc_error_handler.h and erpc_error_handler.cpp files.

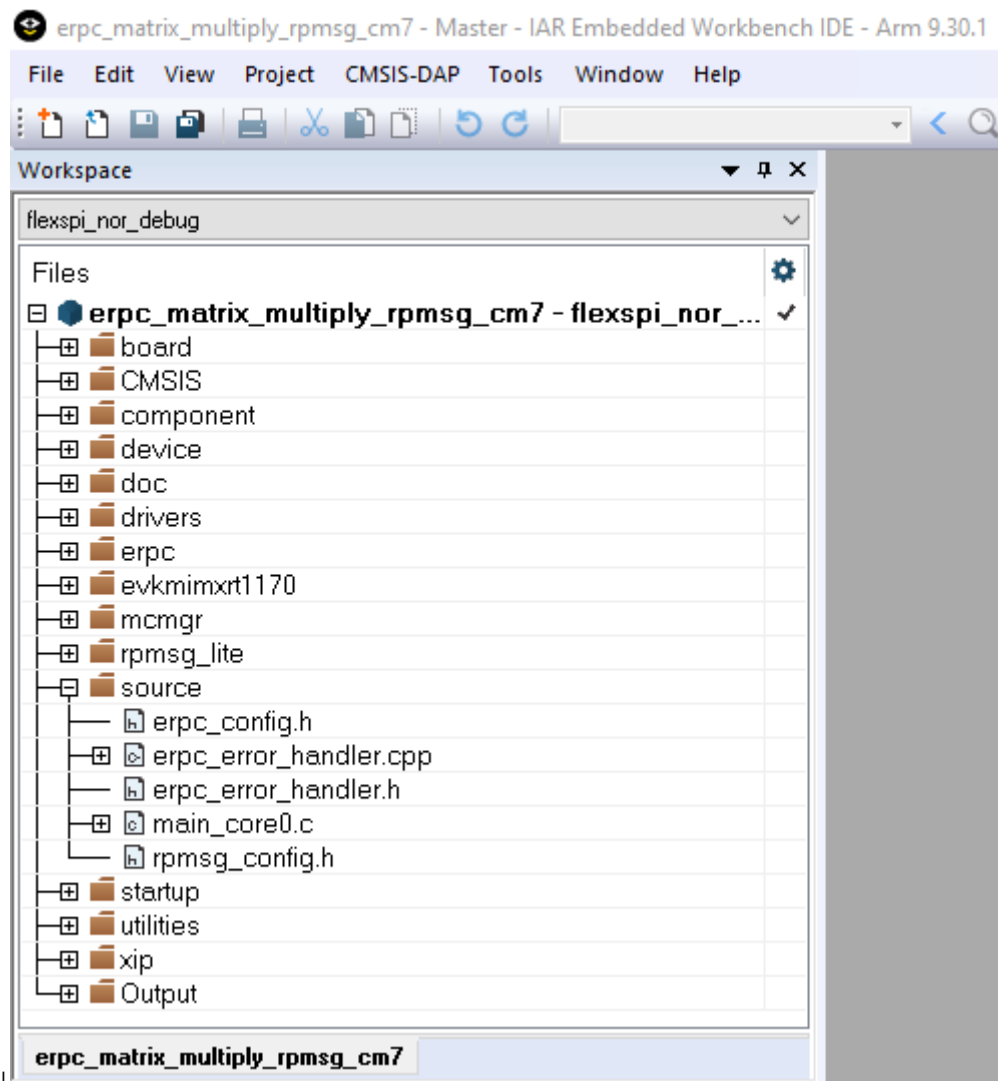
The matrix multiplication can be issued repeatedly, when pressing a software board button.

The eRPC-relevant code is captured in the following code snippet:

```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* RPSMsg-Lite transport layer initialization */
erpc_transport_t transport;
transport = erpc_transport_rpmsg_lite_master_init(src, dst,
ERPC_TRANSPORT_RPMSG_LITE_LINK_ID);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_rpmsg_init(transport);
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport, message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Except for the application main file, there are configuration files for the RPSMsg-Lite (rpmsg_config.h) and eRPC (erpc_config.h), located in the following folder:

<MCUXpressoSDK_install_dir>/boards/evkmimxrt1170/multicore_examples/erpc_matrix_multiply_rpmsg



Parent topic:Multicore client application

Parent topic:[Create an eRPC application](#)

Multiprocessor server application The “Matrix multiply” eRPC server project for multiprocessor applications is located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>` folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires server-related generated files (server shim code), server infrastructure files, and the server user code. There is no need for server multicore infrastructure files (MCMGR and RPMs-Lite). The RPMs-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_slave.cpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.hpp
	<eRPC base directory>/erpc_c/transports/erpc_(d)spi_slave_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.hpp

<eRPC base directory>/erpc_c/transport/erpc_uart_cmsis_transport.cpp

|

Server user code The server's user code is stored in the main_server.c file, located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_server_matrix_multiply_<transport_layer>/ folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

```
/* erpcMatrixMultiply function user implementation */
void erpcMatrixMultiply(Matrix matrix1, Matrix matrix2, Matrix result_matrix)
{
    ...
}
int main()
{
    ...
    /* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver.
    ↪operations */
    erpc_transport_t transport;
    transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
    ...
    /* MessageBufferFactory initialization */
    erpc_mbf_t message_buffer_factory;
    message_buffer_factory = erpc_mbf_dynamic_init();
    ...
    /* eRPC server side initialization */
    erpc_server_t server;
    server = erpc_server_init(transport, message_buffer_factory);
    ...
    /* Adding the service to the server */
    erpc_service_t service = create_MatrixMultiplyService_service();
    erpc_add_service_to_server(server, service);
    ...
    while (1)
    {
        /* Process eRPC requests */
        erpc_status_t status = erpc_server_poll(server)
        /* handle error status */
        if (status != kErpcStatus_Success)
        {
            /* print error description */
            erpc_error_handler(status, 0);
            ...
        }
        ...
    }
}
```

Parent topic:Multiprocessor server application

Multiprocessor client application The “Matrix multiply” eRPC client project for multiprocessor applications is located in the <MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/iar/ folder.

Most of the multiprocessor application setup is the same as for the multicore application. The multiprocessor server application requires client-related generated files (server shim code),

client infrastructure files, and the client user code. There is no need for client multicore infrastructure files (MCMGR and RPSMsg-Lite). The RPSMsg-Lite transport layer is replaced either by SPI or UART transports. The following table shows the required transport-related files per each transport type.

SPI	<eRPC base directory>/erpc_c/setup/erpc_setup_(d)spi_master.cpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.hpp
<eRPC base directory>/erpc_c/	transports/ erpc_(d)spi_master_transport.cpp
UART	<eRPC base directory>/erpc_c/setup/erpc_setup_uart_cmsis.cpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.hpp
<eRPC base directory>/erpc_c/	transports/erpc_uart_cmsis_transport.cpp

Client user code The client's user code is stored in the `main_client.c` file, located in the `<MCUXpressoSDK_install_dir>/boards/<board_name>/multiprocessor_examples/erpc_client_matrix_multiply_<transport_layer>/` folder.

The eRPC-relevant code with UART as a transport is captured in the following code snippet:

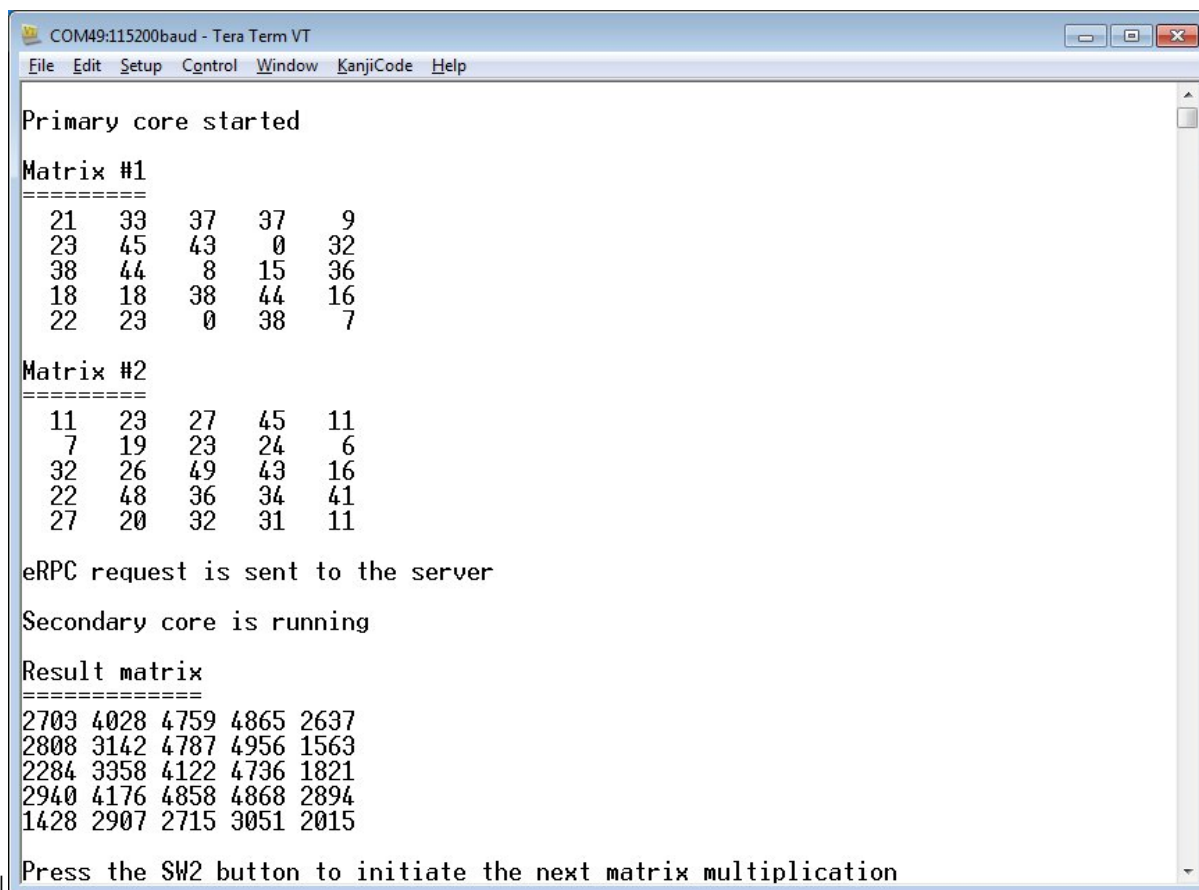
```
...
extern bool g_erpc_error_occurred;
...
/* Declare matrix arrays */
Matrix matrix1 = {0}, matrix2 = {0}, result_matrix = {0};
...
/* UART transport layer initialization, ERPC_DEMO_UART is the structure of CMSIS UART driver
↳operations */
erpc_transport_t transport;
transport = erpc_transport_cmsis_uart_init((void *)&ERPC_DEMO_UART);
...
/* MessageBufferFactory initialization */
erpc_mbf_t message_buffer_factory;
message_buffer_factory = erpc_mbf_dynamic_init();
...
/* eRPC client side initialization */
erpc_client_t client;
client = erpc_client_init(transport,message_buffer_factory);
...
/* Set default error handler */
erpc_client_set_error_handler(client, erpc_error_handler);
...
while (1)
{
    /* Invoke the erpcMatrixMultiply function */
    erpcMatrixMultiply(matrix1, matrix2, result_matrix);
    ...
    /* Check if some error occurred in eRPC */
    if (g_erpc_error_occurred)
    {
        /* Exit program loop */
        break;
    }
    ...
}
```

Parent topic:Multiprocessor client application

Parent topic:Multiprocessor server application

Parent topic:[Create an eRPC application](#)

Running the eRPC application Follow the instructions in *Getting Started with MCUXpresso SDK* (document MCUXSDKGSUG) (located in the <MCUXpressoSDK_install_dir>/docs folder), to load both the primary and the secondary core images into the on-chip memory, and then effectively debug the dual-core application. After the application is running, the serial console should look like:



```

COM49:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help

Primary core started

Matrix #1
=====
 21  33  37  37   9
 23  45  43   0  32
 38  44   8  15  36
 18  18  38  44  16
 22  23   0  38   7

Matrix #2
=====
 11  23  27  45  11
  7  19  23  24   6
 32  26  49  43  16
 22  48  36  34  41
 27  20  32  31  11

eRPC request is sent to the server

Secondary core is running

Result matrix
=====
2703 4028 4759 4865 2637
2808 3142 4787 4956 1563
2284 3358 4122 4736 1821
2940 4176 4858 4868 2894
1428 2907 2715 3051 2015

Press the SW2 button to initiate the next matrix multiplication

```

For multiprocessor applications that are running between PC and the target evaluation board or between two boards, follow the instructions in the accompanied example readme files that provide details about the proper board setup and the PC side setup (Python).

Parent topic:[Create an eRPC application](#)

Parent topic:[eRPC example](#)

Other uses for an eRPC implementation The eRPC implementation is generic, and its use is not limited to just embedded applications. When creating an eRPC application outside the embedded world, the same principles apply. For example, this manual can be used to create an eRPC application for a PC running the Linux operating system. Based on the used type of transport medium, existing transport layers can be used, or new transport layers can be implemented.

For more information and erpc updates see the github.com/EmbeddedRPC.

Note about the source code in the document Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

Unreleased

Added

Fixed

- Python code of the eRPC infrastructure was updated to match the proper python code style, add type annotations and improve readability.

1.14.0

Added

- Added Cmake/Kconfig support.
- Made java code jdk11 compliant, GitHub PR #432.
- Added imxrt1186 support into mu transport layer.
- erpcgen: Added assert for listType before usage, GitHub PR #406.

Fixed

- eRPC: Sources reformatted.
- erpc: Fixed typo in semaphore get (mutex -> semaphore), and write it can fail in case of timeout, GitHub PR #446.
- erpc: Free the arbitrated client token from client manager, GitHub PR #444.

- erpc: Fixed Makefile, install the erpc_simple_server header, GitHub PR #447.
- erpc_python: Fixed possible AttributeError and OSError on calling TCPTransport.close(), GitHub PR #438.
- Examples and tests consolidated.

1.13.0

Added

- erpc: Add BSD-3 license to endianness agnostic files, GitHub PR #417.
- eRPC: Add new Zephyr-related transports (zephyr_uart, zephyr_mbox).
- eRPC: Add new Zephyr-related examples.

Fixed

- eRPC,erpcgen: Fixing/improving markdown files, GitHub PR #395.
- eRPC: Fix Python client TCPTransports not being able to close, GitHub PR #390.
- eRPC,erpcgen: Align switch brackets, GitHub PR #396.
- erpc: Fix zephyr uart transport, GitHub PR #410.
- erpc: UART ZEPHYR Transport stop to work after a few transactions when using USB-CDC resolved, GitHub PR #420.

Removed

- eRPC,erpcgen: Remove cstbool library, GitHub PR #403.

1.12.0

Added

- eRPC: Add dynamic/static option for transport init, GitHub PR #361.
- eRPC,erpcgen: Winsock2 support, GitHub PR #365.
- eRPC,erpcgen: Feature/support multiple clients, GitHub PR #271.
- eRPC,erpcgen: Feature/buffer head - Framed transport header data stored in Message-Buffer, GitHub PR #378.
- eRPC,erpcgen: Add experimental Java support.

Fixed

- eRPC: Fix receive error value for spidev, GitHub PR #363.
- eRPC: UartTransport::init adaptation to changed driver.
- eRPC: Fix typo in assert, GitHub PR #371.
- eRPC,erpcgen: Move enums to enum classes, GitHub PR #379.
- eRPC: Fixed rpmsg tty transport to work with serial transport, GitHub PR #373.

1.11.0

Fixed

- eRPC: Makefiles update, GitHub PR #301.
- eRPC: Resolving warnings in Python, GitHub PR #325.
- eRPC: Python3.8 is not ready for usage of typing.Any type, GitHub PR #325.
- eRPC: Improved codec function to use reference instead of address, GitHub PR #324.
- eRPC: Fix NULL check for pending client creation, GitHub PR #341.
- eRPC: Replace sprintf with snprintf, GitHub PR #343.
- eRPC: Use MU_SendMsg blocking call in MU transport.
- eRPC: New LPSPI and LPI2C transport layers.
- eRPC: Freeing static objects, GitHub PR #353.
- eRPC: Fixed casting in deinit functions, GitHub PR #354.
- eRPC: Align LIBUSB_SIO.GetNumPorts API use with libusb_sio python module v. 2.1.11.
- erpcgen: Renamed temp variable to more generic one, GitHub PR #321.
- erpcgen: Add check that string read is not more than max length, GitHub PR #328.
- erpcgen: Move to g++ in pytest, GitHub PR #335.
- erpcgen: Use build=release for make, GitHub PR #334.
- erpcgen: Removed boost dependency, GitHub PR #346.
- erpcgen: Mingw support, GitHub PR #344.
- erpcgen: VS build update, GitHub PR #347.
- erpcgen: Modified name for common types macro scope, GitHub PR #337.
- erpcgen: Fixed memcpy for template, GitHub PR #352.
- eRPC,erpcgen: Change default build target to release + adding artefacts, GitHub PR #334.
- eRPC,erpcgen: Remove redundant includes, GitHub PR #338.
- eRPC,erpcgen: Many minor code improvements, GitHub PR #323.

1.10.0

Fixed

- eRPC: MU transport layer switched to blocking MU_SendMsg() API use.

1.10.0

Added

- eRPC: Add TCP_NODELAY option to python, GitHub PR #298.

Fixed

- eRPC: MUPTransport adaptation to new supported SoCs.
- eRPC: Simplifying CI with installing dependencies using shell script, GitHub PR #267.
- eRPC: Using event for waiting for sock connection in TCP python server, formatting python code, C specific includes, GitHub PR #269.
- eRPC: Endianness agnostic update, GitHub PR #276.
- eRPC: Assertion added for functions which are returning status on freeing memory, GitHub PR #277.
- eRPC: Fixed closing arbitrator server in unit tests, GitHub PR #293.
- eRPC: Makefile updated to reflect the correct header names, GitHub PR #295.
- eRPC: Compare value length to used length() in reading data from message buffer, GitHub PR #297.
- eRPC: Replace EXPECT_TRUE with EXPECT_EQ in unit tests, GitHub PR #318.
- eRPC: Adapt rpmsg_lite based transports to changed rpmsg_lite_wait_for_link_up() API parameters.
- eRPC, erpcgen: Better distinguish which file can and cannot be linked by C linker, GitHub PR #266.
- eRPC, erpcgen: Stop checking if pointer is NULL before sending it to the erpc_free function, GitHub PR #275.
- eRPC, erpcgen: Changed api to count with more interfaces, GitHub PR #304.
- erpcgen: Check before reading from heap the buffer boundaries, GitHub PR #287.
- erpcgen: Several fixes for tests and CI, GitHub PR #289.
- erpcgen: Refactoring erpcgen code, GitHub PR #302.
- erpcgen: Fixed assigning const value to enum, GitHub PR #309.
- erpcgen: Enable runTesttest_enumErrorCode_allDirection, serialize enums as int32 instead of uint32.

1.9.1

Fixed

- eRPC: Construct the USB CDC transport, rather than a client, GitHub PR #220.
- eRPC: Fix premature import of package, causing failure when attempting installation of Python library in a clean environment, GitHub PR #38, #226.
- eRPC: Improve python detection in make, GitHub PR #225.
- eRPC: Fix several warnings with deprecated call in pytest, GitHub PR #227.
- eRPC: Fix freeing union members when only default need be freed, GitHub PR #228.
- eRPC: Fix making test under Linux, GitHub PR #229.
- eRPC: Assert costumizing, GitHub PR #148.
- eRPC: Fix corrupt clientList bug in TransportArbitrator, GitHub PR #199.
- eRPC: Fix build issue when invoking g++ with -Wno-error=free-nonheap-object, GitHub PR #233.
- eRPC: Fix inout cases, GitHub PR #237.

- eRPC: Remove ERPC_PRE_POST_ACTION dependency on return type, GitHub PR #238.
- eRPC: Adding NULL to ptr when codec function failed, fixing memcpy when fail is present during deserialization, GitHub PR #253.
- eRPC: MessageBuffer usage improvement, GitHub PR #258.
- eRPC: Get rid of serial and enum34 dependency (enum34 is in python3 since 3.4 (from 2014)), GitHub PR #247.
- eRPC: Several MISRA violations addressed.
- eRPC: Fix timeout for Freertos semaphore, GitHub PR #251.
- eRPC: Use of rpmsg_lite_wait_for_link_up() in rpmsg_lite based transports, GitHub PR #223.
- eRPC: Fix codec nullptr dereferencing, GitHub PR #264.
- erpcgen: Fix two syntax errors in erpcgen Python output related to non-encapsulated unions, improved test for union, GitHub PR #206, #224.
- erpcgen: Fix serialization of list/binary types, GitHub PR #240.
- erpcgen: Fix empty list parsing, GitHub PR #72.
- erpcgen: Fix templates for malloc errors, GitHub PR #110.
- erpcgen: Get rid of encapsulated union declarations in global scale, improve enum usage in unions, GitHub PR #249, #250.
- erpcgen: Fix compile error:UniqueIdChecker.cpp:156:104:'sort' was not declared, GitHub PR #265.

1.9.0

Added

- eRPC: Allow used LIBUSBSIO device index being specified from the Python command line argument.

Fixed

- eRPC: Improving template usage, GitHub PR #153.
- eRPC: run_clang_format.py cleanup, GitHub PR #177.
- eRPC: Build TCP transport setup code into liberpc, GitHub PR #179.
- eRPC: Fix multiple definitions of g_client error, GitHub PR #180.
- eRPC: Fix memset past end of buffer in erpc_setup_mbf_static.cpp, GitHub PR #184.
- eRPC: Fix deprecated error with newer pytest version, GitHub PR #203.
- eRPC, erpcgen: Static allocation support and usage of rpmsg static FreeRTOSs related API, GitHub PR #168, #169.
- erpcgen: Remove redundant module imports in erpcgen, GitHub PR #196.

1.8.1

Added

- eRPC: New i2c_slave_transport transport introduced.

Fixed

- eRPC: Fix misra erpc c, GitHub PR #158.
- eRPC: Allow conditional compilation of message_loggers and pre_post_action.
- eRPC: (D)SPI slave transports updated to avoid busy loops in rtos environments.
- erpcgen: Re-implement EnumMember::hasValue(), GitHub PR #159.
- erpcgen: Fixing several misra issues in shim code, erpcgen and unit tests updated, GitHub PR #156.
- erpcgen: Fix bison file, GitHub PR #156.

1.8.0

Added

- eRPC: Support win32 thread, GitHub PR #108.
- eRPC: Add mbed support for malloc() and free(), GitHub PR #92.
- eRPC: Introduced pre and post callbacks for eRPC call, GitHub PR #131.
- eRPC: Introduced new USB CDC transport.
- eRPC: Introduced new Linux spidev-based transport.
- eRPC: Added formatting extension for VSC, GitHub PR #134.
- erpcgen: Introduce ustring type for unsigned char and force cast to char*, GitHub PR #125.

Fixed

- eRPC: Update makefile.
- eRPC: Fixed warnings and error with using MessageLoggers, GitHub PR #127.
- eRPC: Extend error msg for python server service handle function, GitHub PR #132.
- eRPC: Update CMSIS UART transport layer to avoid busy loops in rtos environments, introduce semaphores.
- eRPC: SPI transport update to allow usage without handshaking GPIO.
- eRPC: Native _WIN32 erpc serial transport and threading.
- eRPC: Arbitrator deadlock fix, TCP transport updated, TCP setup functions introduced, GitHub PR #121.
- eRPC: Update of matrix_multiply.py example: Add -serial and -baud argument, GitHub PR #137.
- eRPC: Update of .clang-format, GitHub PR #140.
- eRPC: Update of erpc_framed_transport.cpp: return error if received message has zero length, GitHub PR #141.
- eRPC, erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136, #139.
- eRPC, erpcgen: Core re-formatted using Clang version 10.
- erpcgen: Enable deallocation in server shim code when callback/function pointer used as out parameter in IDL.
- erpcgen: Removed '\$' character from generated symbol name in '_\$union' suffix, GitHub PR #103.

- erpcgen: Resolved mismatch between C++ and Python for callback index type, GitHub PR #111.
- erpcgen: Python generator improvements, GitHub PR #100, #118.
- erpcgen: Fixed error messages produced by -Wall -Wextra -Wshadow -pedantic-errors compiler flags, GitHub PR #136.

1.7.4

Added

- eRPC: Support MU transport unit testing.
- eRPC: Adding mbed os support.

Fixed

- eRPC: Unit test code updated to handle service add and remove operations.
- eRPC: Several MISRA issues in rpmsg-based transports addressed.
- eRPC: Fixed Linux/TCP acceptance tests in release target.
- eRPC: Minor documentation updates, code formatting.
- erpcgen: Whitespace removed from C common header template.

1.7.3

Fixed

- eRPC: Improved the test_callbacks logic to be more understandable and to allow requested callback execution on the server side.
- eRPC: TransportArbitrator::prepareClientReceive modified to avoid incorrect return value type.
- eRPC: The ClientManager and the ArbitratedClientManager updated to avoid performing client requests when the previous serialization phase fails.
- erpcgen: Generate the shim code for destroy of statically allocated services.

1.7.2

Added

- eRPC: Add missing doxygen comments for transports.

Fixed

- eRPC: Improved support of const types.
- eRPC: Fixed Mac build.
- eRPC: Fixed serializing python list.
- eRPC: Documentation update.

1.7.1

Fixed

- eRPC: Fixed semaphore in static message buffer factory.
- erpcgen: Fixed MU received error flag.
- erpcgen: Fixed tcp transport.

1.7.0

Added

- eRPC: List names are based on their types. Names are more deterministic.
- eRPC: Service objects are as a default created as global static objects.
- eRPC: Added missing doxygen comments.
- eRPC: Added support for 64bit numbers.
- eRPC: Added support of program language specific annotations.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Generating crc value is optional.
- eRPC: Fixed CMSIS Uart driver. Removed dependency on KSDK.
- eRPC: Forbid users use reserved words.
- eRPC: Removed outByref for function parameters.
- eRPC: Optimized code style of callback functions.

1.6.0

Added

- eRPC: Added @nullable support for scalar types.

Fixed

- eRPC: Improved code size of generated code.
- eRPC: Improved eRPC nested calls.
- eRPC: Improved eRPC list length variable serialization.

1.5.0

Added

- eRPC: Added support for unions type non-wrapped by structure.
- eRPC: Added callbacks support.
- eRPC: Added support @external annotation for functions.
- eRPC: Added support @name annotation.
- eRPC: Added Messaging Unit transport layer.
- eRPC: Added RPMSG Lite RTOS TTY transport layer.
- eRPC: Added version verification and IDL version verification between eRPC code and eRPC generated shim code.
- eRPC: Added support of shared memory pointer.
- eRPC: Added annotation to forbid generating const keyword for function parameters.
- eRPC: Added python matrix multiply example.
- eRPC: Added nested call support.
- eRPC: Added struct member “byref” option support.
- eRPC: Added support of forward declarations of structures
- eRPC: Added Python RPMsg Multiendpoint kernel module support
- eRPC: Added eRPC sniffer tool

1.4.0

Added

- eRPC: New RPMsg-Lite Zero Copy (RPMsgZC) transport layer.

Fixed

- eRPC: win_flex_bison.zip for windows updated.
- eRPC: Use one codec (instead of inCodec outCodec).

[1.3.0]

Added

- eRPC: New annotation types introduced (@length, @max_length, ...).
- eRPC: Support for running both erpc client and erpc server on one side.
- eRPC: New transport layers for (LP)UART, (D)SPI.
- eRPC: Error handling support.

[1.2.0]

Added

- eRPC source directory organization changed.
- Many eRPC improvements.

[1.1.0]

Added

- Multicore SDK 1.1.0 ported to KSDK 2.0.0.

[1.0.0]

Added

- Initial Release

3.5 Multimedia

3.5.1 Audio Voice

Audio Voice Components

MCUXpresso SDK : audio-voice-components

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

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

Visit [Audio Voice Components - Documentation](#) to review details on the contents in this sub-repo.

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

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

Overview This repository allows users to add additional functionality to the [Maestro Audio framework](#). This structure is designed for integration with Maestro and is not intended for standalone use. For information on the use of individual components, please refer to the [Maestro programmer's guide](#).

This repository acts as Zephyr module, to be able to use these libraries in Zephyr build system.

Content

- [asrc](#) - Libraries and public files of Asynchronous Sample Rate Converter, version 1.0.0
- [ssrc](#) - Libraries and public files of Synchronous Sample Rate Converter, version 1.0.0
- [opus](#) - Source files of Opus decoder and encoder, version 1.3.1
- [opusfile](#) - Source files for Opus streams in the Ogg container, version 0.12
- [ogg](#) - Source files of Ogg container, version 1.3.5
- [decoders](#) - Libraries and public files of following audio decoders:
 - [aac](#) - AAC decoder, version 1.0.0
 - [flac](#) - FLAC decoder, version 1.0.0
 - [mp3](#) - MP3 decoder, version 1.0.0
 - [wav](#) - WAV decoder, version 1.0.0
- [zephyr/](#) - Files allowing usage of the libraries in Zephyr build

Following table contains information about libraries and source files availability:

Asynchronous Sample Rate Converter The Asynchronous Sample Rate Converter (ASRC) software module compensates the drift between two mono audio signals. This is not a frequency converter and so the nominal signal frequency is the same before and after the ASRC. More details about ASRC are available in the User Guide, which is located in `asrc\doc\`.

Synchronous Sample Rate Converter The Synchronous Sample Rate Converter (SSRC) software module converts an audio signal (mono or stereo) with a certain sampling frequency to an audio signal with another sampling frequency. More details about SSRC are available in the [User Guide](#).

Opus For Opus decoder and encoder documentation please see following link: [opus](#).

Opus File The Opus File provides a API for decoding and basic manipulation of Opus streams in Ogg container and depends on [Opus](#) and [Ogg](#) libraries. For Opus File documentation please see following link: [opusfile](#).

Ogg Container For Ogg container documentation please see following link: [ogg](#).

Decoders Each decoder contains libraries for supported processor and toolchain (see table above), corresponding Public API file and documentation folder.

AAC For decoder features please see [aacdec](#), for API Usage please see [aacd_ug](#).

FLAC For decoder features please see [flacdec](#), for API Usage please see [flacd_ug](#).

MP3 For decoder features please see [mp3dec](#), for API Usage please see [mp3d_ug](#).

WAV For decoder features please see [wavdec](#), for API Usage please see [wavd_ug](#).

Zephyr build To add library into the Zephyr build, add `CONFIG_NXP_AUDIO_VOICE_COMPONENTS_*` for specific libraries into your `prj.conf`. For all configuration options, see *zephyr/Kconfig*.

List of supported libraries in Zephyr:

- Decoders:
 - AAC
 - FLAC
 - MP3
 - FLAC
 - OPUS
- Encoders
 - OPUS

AAC decoder

AAC decoder features

- The AAC decoder implementation supports the following:
- Supported profile : AAC-LC
- Sampling rate : 8 kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz, 48 kHz
- Channel : stereo and mono
- Bits per samples : 16 bit
- Container format : (MPEG-2 Style)AAC transport format - ADTS and ADIF.

Specification and reference

Performance

Memory information The memory usage of the decoder in bytes is:

- Code/flash = $26332 + 19264 = 45596$
- Data/RAM = 26832

Section	Size
.text	26332
.ro & .const	19264
.bss	26832

CPU usage

- CPU core clock in MHz: 20.97.

Track type	Duration of track in second	Frame size in bytes	Performance MIPS of codec (in MHz)
48 kHz, stereo	38 s	4096	12.2 MHz

API Usage of AAC Decoder

Overview

- This section describes the integration steps to call AAC decoder APIs by the application code. During each step, the used data structures and functions are explained. All CCI public APIs are defined in `aac_cci.h` header file. This file is located at `\decoders\aac`.

Configuration

Build Options AAC Decoder library is built with the following defined/enabled macros.

- There is no macro or define used to build the AAC decoder.

Buffer Allocation

- The AAC decoder does not perform dynamic memory allocation. The application calls the function `AACDecoderGetMemorySize()` to get the decoder memory requirements. This function must be called before all other decoder functions are invoked.
- The application first gets the required memory size for the decoder, then allocates memory for the decoder structures. Structures contain Main Decoder parameters and decoder information parameters.
- This function populates the required memory for the decoder and returns the required memory size in bytes.

Initialization

- `AACDecoderInit()` function must be called before decode API. This API allocates the memory to decoder main structure and also initializes the decoder main structure parameters.
- It also registers the call back functions to the decoder, which is used by the decoder to read or seek the input stream.

Decoding

- `AACDecoderDecode()` function is main decoding API of the decoder. This API decodes the encoded input stream and fills the PCM output samples into decoder output PCM buffer.
- This API gives the information about the number of samples produced by the decoder and also provides the pointer to the decoder output PCM samples buffer.

Seeking

- AACDecoderSeek() function calculates the actual frame boundary align offset from the un-align seek offset and returns the actual seek offset. It also resets the decoder internal states and variables.

Callback Usage All the callback functions are assigned to the respective pointers before the codec initialization is called. Callback APIs are described below.

Read Callback API AAC Decoder read call back API reads the bytes from the input stream and fills them into decoder internal bit stream buffer. It returns the number of bytes read from the input stream.

Seek Callback API This call back API is for the seek operation.

Get File Position Callback API This call back API gives the current file position.

FLAC decoder

FLAC decoder features

- The FLAC decoder implementation support the following:
- Sampling rate: 8 kHz, 11.05 kHz, 12 kHz, 16 kHz, 22.05 kHz, 32 kHz, 44.1 kHz, and 48 kHz.
- Channel : stereo and mono
- Bits per samples : 16 bits

Specification and reference

Official website

- FLAC lossless audio codec is at <https://xiph.org/flac>.

Inbound licensing

- For licensing information please refer to FLAC's official website: <https://xiph.org/flac/license.html>.

Performance

Memory information The memory usage of the decoder in bytes is:

- Code/flash = 15744 + 2080 = 17824
- Data/RAM = 27936

Section	Size
.text	15744
.ro & .const	2080
.bss	27936

CPU usage

- Output frame size: 16384 bytes.
- CPU core clock in MHz: 20.97.

Track type	Duration of track in second	Performance MIPS of codec (in MHz)
48 kHz, stereo	76 s	30.7 MHz
32 kHz, stereo	76 s	20.3 MHz
8 kHz, stereo	37 s	5.34 MHz

Following test cases are performed:

- Audio format listening test
- Audio quality test

For all above test cases, test tracks are played through the end without any distortion, glitching, hanging, or crashing.

API Usage of FLAC Decoder

Overview

- This section describes the integration steps to call FLAC decoder APIs by the application code. During each step the used data structures and functions are explained. All cci public APIs are defined in flac_cci.h header file. This file is located at `\decoders\flac\include`.

Configuration

Build Options

- `SUPPORT_16_BITS_ONLY` :- This macro is used to enable 16bits per sample flac decoder.
- `ASM` :- This macro is used to enable ARM assembly macros for 24bits per sample flac decoder.

Buffer Allocation

- The FLAC decoder does not perform dynamic memory allocation. The application calls the function `FLACDecoderGetMemorySize()` to get the decoder memory requirements. This function must be called before all other decoder functions are invoked.
- The application first gets the required memory size for the decoder and then allocates memory for the decoder structures. Structures contain Main Decoder parameters and decoder information parameters.
- This function populates the required memory for the decoder and returns the required memory size in bytes.

Initialization

- `FLACDecoderInit()` function must be called before decode API. This API allocates the memory to decoder main structure and also initializes the decoder main structure parameters.
- It also registers the call back functions to the decoder, which will be used by decoder to read or to seek the input stream.

Decoding

- `FLACDecoderDecode()` function is main decoding API of the decoder. This API decodes the encoded input stream and fills the PCM output samples into decoder output PCM buffer.
- This API gives the information about the number of samples produced by the decoder and also provides the pointer to the decoder output PCM samples buffer.

Seeking

- `FLACDecoderSeek()` function calculates the actual frame boundary align offset from the unalign seek offset and returns the actual seek offset. It also resets the decoder internal states and variables.

Callback Usage All the callback functions will be assigned to the respective pointers before the codec initialization is called. Callback APIs are described below.

Read Callback API FLAC Decoder read call back API reads the bytes from the input stream and fills them into decoder internal bit stream buffer. It returns the number of bytes read from the input stream.

Seek Callback API This call back API is for the seek operation.

Get File Position Callback API This call back API gives the current file position.

MP3 decoder

MP3 decoder features

- MP3 decoder supports mpeg-1, mpeg-2, mpeg-2.5.
- All MP3 features supported , including joint stereo, mid-side stereo, intensity stereo, and dual channel.
- Supported sampling rate: 8 kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz and 48 kHz.
- Supported channel: stereo and mono
- Supported bits per samples: 16 bit
- Supported bit rate: 8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 144, 160, 176, 192, 224, 256, 320, 384, 416, and 448.

Performance

Memory information The memory usage of the decoder (data obtained from IAR compiler) in bytes is:

- Code/flash = $26884 + 18372 = 45256$
- RAM = 16200

Section	Size
.text	26884
.ro & .const	18372
.bss	16200

CPU usage The performance of the decoder was measured using the real hardware platform (RT1060).

- CPU core clock in MHz: 600.

Track type	Duration of track in second	Frame size in bytes	Performance MIPS of codec (in MHz)
320 Kbps, 44.1 kHz, stereo	358 s	2304	~24 MHz
192 Kbps, 48 kHz, stereo	10 s	2304	~18 MHz

API Usage of MP3 Decoder

Overview

- This section describes the integration steps to call MP3 decoder APIs by the application code. During each step the used data structures and functions are explained. All cci public APIs are defined in mp3_cci.h header file. This file is located at \decoders\mp3.

Configuration

Build Options MP3 Decoder library is built with the following defined/enabled macros.

- There is no macro or define used to build the MP3 decoder.

Buffer Allocation

- The MP3 decoder does not perform dynamic memory allocation. The application calls the function MP3DecoderGetMemorySize() to get the decoder memory requirements. This function must be called before all other decoder functions are invoked.
- The application first gets the required memory size for the decoder and then allocates memory for the decoder structures. Structures contain Main Decoder parameters and decoder information parameters.
- This function populates the required memory for the decoder and returns the required memory size in bytes.

Initialization

- MP3DecoderInit() function must be called before decode API. This API allocates the memory to decoder main structure and also initializes the decoder main structure parameters.
- It also registers the call back functions to the decoder, which will be used by decoder to read or to seek the input stream.

Decoding

- MP3DecoderDecode() function is main decoding API of the decoder. This API decodes the encoded input stream and fills the PCM output samples into decoder output PCM buffer.
- This API gives the information about the number of samples produced by the decoder and also provides the pointer to the decoder output PCM samples buffer.

Seeking

- MP3DecoderSeek() function calculates the actual frame boundary align offset from the un-align seek offset and returns the actual seek offset. It also resets the decoder internal states and variables.

Callback Usage All the callback functions will be assigned to the respective pointers before the codec initialization is called. Callback APIs are described below.

Read Callback API MP3 Decoder read call back API reads the bytes from the input stream and fills them into decoder internal bit stream buffer. It returns the number of bytes read from the input stream.

Seek Callback API This call back API is for the seek operation.

Get File Position Callback API This call back API gives the current file position.

WAV decoder

WAV decoder features

- The WAV decoder implementation support the following:
- Sampling rate: 8 kHz, 11.025kHz, 16 kHz, 22.05 kHz, 32 kHz, 44.1 kHz, and 48 kHz.
- Channel: stereo and mono
- PCM format with 8/16/24 bits per sample.

Performance

Memory information The memory usage of the decoder in bytes is:

- Code/flash = 6260 + 342 = 6602
- Data/RAM = 16 + 20696 = 20712

Section	Size
.text	6260
.ro & .const	342
.bss	20696
.data	16

CPU usage The performance of the decoder was measured using the decoder standalone unit test.

- CPU core clock in MHz: 20.97 MHz.

Track type	Duration of track in second	Frame size in bytes	Performance MIPS of codec (in MHz)
48 kHz, stereo, PCM	12 s	4096	9.68 MHz

Following test cases were performed:

- Audio format listening test
- Audio quality test

For all above test cases, test tracks are played through the end without any distortion, glitching, hanging, or crashing.

API Usage of WAV Decoder

Overview

- This section describes the integration steps to call MP3 decoder APIs by the application code. During each step the used data structures and functions are explained. All cci public APIs are defined in wav_cci.h header file. This file is located at `\decoders\wav`.

Configuration

Build Options WAV Decoder library is built with the following defined/enabled macros.

- There is no macro or define used to build the WAV decoder.

Buffer Allocation

- The WAV decoder does not perform dynamic memory allocation. The application calls the function `WAVDecoderGetMemorySize()` to get the decoder memory requirements. This function must be called before all other decoder functions are invoked.
- The application first gets the required memory size for the decoder and then allocates memory for the decoder structures. Structures contain Main Decoder parameters and decoder information parameters.
- This function populates the required memory for the decoder and returns the required memory size in bytes.

Initialization

- `WAVDecoderInit()` function must be called before decode API. This API allocates the memory to decoder main structure and also initializes the decoder main structure parameters.
- It also registers the call back functions to the decoder, which will be used by decoder to read or to seek the input stream.

Decoding

- WAVDecoderDecode() function is main decoding API of the decoder. This API decodes the encoded input stream and fills the PCM output samples into decoder output PCM buffer.
- This API gives the information about the number of samples produced by the decoder and also provides the pointer to the decoder output PCM samples buffer.

Seeking

- WAVDecoderSeek() function calculates the actual frame boundary align offset from the un-align seek offset and returns the actual seek offset. It also resets the decoder internal states and variables.

Callback Usage All the callback functions will be assigned to the respective pointers before the codec initialization is called. Callback APIs are described below.

Read Callback API WAV Decoder read call back API reads the bytes from the input stream and fills them into decoder internal bit stream buffer. It returns the number of bytes read from the input stream.

Seek Callback API This call back API is for the seek operation.

Get File Position Callback API This call back API gives the current file position.

Synchronous Sample Rate Converter

Introduction The Synchronous Sample Rate Converter (SSRC) software module converts a mono or stereo audio signal with a certain sampling frequency to an audio signal with a different sampling frequency. The sample rate converter works synchronously, meaning that input and output sampling rates are exactly known for a mutual clock reference.

To accomplish a professional sampling conversion quality and minimal system footprint, the SRC SW module contains highly optimized components.

The SSRC module supports the following features.

- Multiple instances of the sample rate converter can run at the same time.
- Supported sampling frequencies: 32 kHz, 44.1 kHz, and 48 kHz plus the halves and the quarters of these three sample rates. The input and output sample rates are freely selectable out of the supported sampling rates
- Selectable Mono/Stereo Input/Output.
- Selectable quality level: high quality/ very high quality.

Acronyms *Table 1* lists the acronyms used in this document.

Acron	Description
Fs	Sampling Frequency
Fs-LOW	Lowest sample rate used for the conversion Note: Input sample rate for up sampling and the output sample rate for down sampling
FsIN	Input sample rate
FsOUT	Output sample rate
MIPS	Million Instructions Per Second
SSRC	Synchronous sample rate converter
THD+N	Total Harmonic Distortion plus Noise Note: The THD+N is defined as the total power of the unwanted signal divided by the power of the wanted signal. The wanted signal is defined as a full scale, 1 kHz sine wave.

Parent topic:[Introduction](#)

Performance figures The Total Harmonic Distortion Plus Noise (THD+N) of the converted signals is below - 76 (high-quality mode) and - 85 (very high-quality mode) for signal frequencies below $0.45 \cdot F_{sLOW}$ (=90 % of the Nyquist range of the lowest sample clock)

Table 1 and *Table 2* give the THD+N performance (FsIN on the vertical axis and FsOUT on the horizontal axis) for the two supported quality levels. The numbers in the tables give the worst-case THD+N measured for signal frequencies below $0.45 \cdot F_{sLOW}$. For each conversion ratio, 100 THD+N measurements were executed with signal frequencies linearly spread over the complete Nyquist range.

FsIN/ FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	-92.1	-79.7	-80.1	-80.1	-79.6	-80.2	-79.4	-79.1	-79.2
11025	-79	-92.9	-80	-79.9	-80.2	-79.8	-79.9	-79.5	-78.9
12000	-79	-79.2	-92.7	-80.1	-79.8	-80.3	-79.8	-79.8	-79.5
16000	-81.7	-78.8	-80.2	-93	-78.3	-77.7	-78.3	-78.3	-77.9
22050	-77.5	-81.8	-78.2	-79	-93	-79.9	-79.8	-80.3	-79.9
24000	-77.4	-77.9	-81.2	-79.1	-79.2	-92.5	-80.1	-79.8	-79.9
32000	-81	-77.5	-78.9	-81.2	-78.7	-80.1	-92.9	-79.7	-79.2
44100	-79.1	-81.2	-76.7	-77.8	-82	-78.2	-79.1	-93	-79.7
48000	-78.7	-78.8	-81.1	-77.6	-77.9	-81.8	-79.1	-79.3	-93

FsIN/ FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	-92.1	-86.6	-88.6	-91.5	-86.4	-89	-89.7	-89.3	-89.3
11025	-89.1	-92.9	-86.3	-86.3	-91.6	-86.3	-86.5	-89.7	-89.3
12000	-91.4	-88.4	-92.7	-89.6	-86.6	-91.5	-86.8	-86.6	-89.7
16000	-93.1	-88.4	-90.4	-93	-86.6	-88.8	-91.5	-86.5	-89.4
22050	-90.7	-93.5	-89.7	-89.3	-93	-86.5	-86.3	-91.5	-86.6
24000	-93.8	-90.5	-93.5	-91.7	-88.4	-92.5	-89.7	-86.6	-91.5
32000	-93.8	-91	-91.2	-93.3	-88.4	-90.5	-92.9	-86.7	-89
44100	-93.7	-93.6	-91.5	-90.6	-93.8	-89.8	-89.3	-93	-86.5
48000	-94.1	-92.6	-94	-94	-90.1	-93.7	-91.8	-88.4	-93

Parent topic:[Introduction](#)

Resource usage This section lists the memory and processing requirements for the SSRC module.

Memory requirements The following are the memory requirements for the SSRC module.

Memory item	Size in bytes
Instance memory (persistent)	548
Scratch memory (non-persistent)	15.536 ¹
Program memory for Arm9E and XScale	14k
Program memory for Arm7	15k

Parent topic:Resource usage

¹ Worst case number for I/O buffers of 40 ms. If smaller I/O buffers are used, this number is smaller. The required scratch memory is roughly equal to 2 times the buffer size on the highest sample rate.

Processing requirements The following tables give the MIPS performance of the SSRC module. The cycles are measured with zero wait state memory and for I/O buffers of 40 ms.

Note: The user processing 32-bit processing must refer to the very high-quality MIPS results.

On Arm7 and Arm9

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.13	4.77	5.17	1.84	6.75	7.33	3.55	9.1	9.89
11025	5.42	0.18	5.58	6.84	2.53	7.75	9.71	4.89	10.31
12000	5.85	6.39	0.2	7.01	8.97	2.76	9.89	12.94	5.32
16000	1.69	7.74	7.99	0.26	9.54	10.33	3.68	13.5	14.65
22050	7.2	2.33	10.09	10.83	0.36	11.17	13.67	5.07	15.49
24000	7.79	8.33	2.53	11.7	12.78	0.39	14.03	17.94	5.51
32000	3.12	10.32	10.58	3.38	15.48	15.98	0.52	19.08	20.66
44100	9.96	4.3	13.65	14.4	4.65	20.18	21.67	0.72	22.34
48000	10.8	11.34	4.68	15.58	16.67	5.06	23.4	25.56	0.78

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.07	7.71	8.24	2.28	10.5	11.28	4.41	13.44	14.48
11025	8.19	0.1	8.96	11.04	3.14	12	15.09	6.08	15.2
12000	8.76	9.52	0.1	11.3	14.48	3.41	15.36	20.07	6.61
16000	2.14	11.73	12.01	0.14	15.41	16.48	4.55	21	22.56
22050	10.78	2.94	15.39	16.38	0.19	17.92	22.08	6.27	24
24000	11.57	12.34	3.2	17.51	19.04	0.21	22.61	28.97	6.83
32000	4.19	15.48	15.77	4.27	23.46	24.01	0.28	30.83	32.96
44100	14.78	5.77	20.56	21.56	5.89	30.77	32.75	0.38	35.83
48000	15.92	16.7	6.28	23.15	24.69	6.41	35.02	38.08	0.42

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.13	13.61	14.52	4.43	19.03	20.43	8.8	25.06	26.99
11025	14.85	0.18	15.91	19.47	6.1	21.82	27.35	12.13	28.38
12000	15.84	17.36	0.2	19.97	25.4	6.64	27.85	36.26	13.21
16000	4.25	21.24	21.79	0.26	27.22	29.03	8.86	38.07	40.85
22050	20.02	5.85	27.72	29.7	0.36	31.81	38.94	12.2	43.63
24000	21.45	22.98	6.37	31.68	34.71	0.39	39.94	50.8	13.28
32000	8.39	28.74	29.29	8.5	42.48	43.58	0.52	54.43	58.07
44100	28.11	11.57	38.05	40.03	11.71	55.43	59.4	0.72	63.62
48000	30.19	31.71	12.59	42.9	45.96	12.74	63.36	69.42	0.78

Parent topic:Processing requirements

On Arm9e and XScale

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.03	1.14	1.25	0.54	1.95	2.14	1.04	3.85	4.23
11025	1.31	0.05	1.36	1.62	0.75	2.23	2.78	1.44	4.38
12000	1.43	1.57	0.05	1.68	2.13	0.82	2.84	3.72	1.57
16000	0.5	1.86	1.93	0.07	2.27	2.5	1.09	3.9	4.29
22050	2.19	0.69	2.42	2.61	0.1	2.72	3.24	1.5	4.46
24000	2.4	2.52	0.75	2.86	3.15	0.1	3.35	4.25	1.63
32000	0.92	3.12	3.18	1.01	3.72	3.86	0.14	4.55	4.99
44100	4.28	1.27	4.15	4.37	1.39	4.83	5.23	0.19	5.43
48000	4.7	4.9	1.39	4.8	5.03	1.51	5.72	6.3	0.21

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.06	1.87	2.02	1.07	3.09	3.36	2.07	6.09	6.63
11025	2.27	0.09	2.25	2.66	1.47	3.56	4.4	2.85	7.01
12000	2.45	2.76	0.09	2.75	3.43	1.6	4.5	5.83	3.1
16000	0.99	3.23	3.36	0.13	3.73	4.05	2.14	6.17	6.72
22050	3.69	1.36	4.14	4.55	0.17	4.51	5.31	2.95	7.13
24000	4.01	4.28	1.48	4.9	5.51	0.19	5.51	6.85	3.21
32000	1.83	5.26	5.39	1.98	6.46	6.71	0.25	7.47	8.09
44100	7.22	2.52	6.94	7.38	2.72	8.27	9.1	0.35	9.02
48000	7.85	8.33	2.74	8.02	8.57	2.97	9.81	11.03	0.38

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.03	1.21	1.33	0.61	2.08	2.29	1.17	4.1	4.51
11025	1.47	0.05	1.44	1.72	0.84	2.38	2.97	1.61	4.66
12000	1.62	1.76	0.05	1.78	2.26	0.91	3.03	3.98	1.75
16000	0.55	2.1	2.17	0.07	2.42	2.65	1.22	4.16	4.57
22050	2.49	0.76	2.73	2.95	0.1	2.88	3.45	1.68	4.75
24000	2.75	2.86	0.83	3.23	3.52	0.1	3.56	4.53	1.83
32000	1	3.56	3.63	1.11	4.2	4.34	0.14	4.84	5.3
44100	4.86	1.38	4.74	4.98	1.53	5.46	5.89	0.19	5.75
48000	5.38	5.55	1.5	5.5	5.71	1.66	6.47	7.05	0.21

FsIN / FsOUT	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	0.06	2.11	2.29	1.2	3.55	3.86	2.31	6.99	7.61
11025	2.62	0.09	2.52	3.01	1.66	4.07	5.07	3.19	8
12000	2.85	3.15	0.09	3.11	3.9	1.81	5.17	6.75	3.47
16000	1.09	3.73	3.85	0.13	4.22	4.57	2.41	7.1	7.72
22050	4.32	1.5	4.79	5.23	0.17	5.05	6.02	3.32	8.15
24000	4.74	4.99	1.64	5.69	6.3	0.19	6.22	7.8	3.61
32000	1.98	6.18	6.3	2.18	7.45	7.71	0.25	8.44	9.14
44100	8.43	2.72	8.18	8.64	3.01	9.59	10.47	0.35	10.1
48000	9.26	9.66	2.97	9.49	9.97	3.27	11.39	12.59	0.38

Parent topic:Processing requirements

On Cortex-A8 for worst case of 48000 Hz to 44100 Hz

Mode	MIPs
Mono at High Quality	3.13
Stereo at High Quality	3.61
Mono at Very High Quality	4.13
Stereo at Very High Quality	6.52

Parent topic:Processing requirements

Parent topic:Resource usage

Parent topic:[Introduction](#)

Application programmers interface (API) This section describes the application programming interface (API) libraries of the SSRC module.

Type definitions This section describes the type definitions of the SSRC module.

Types for allocation of instance and scratch memory The instance memory is the memory that contains the state of one instance of the SSRC module. Multiple instances of the SSRC module can exist, each with its own instance memory. S memory is the memory that is only used temporarily by the process function of the SSRC module. This memory can be used as scratch memory by any other function running in the same thread as the SSRC module. Different threads cannot share the scratch memories.

The application must allocate both the instance and the scratch memory. The SSRC module does not allocate memory.

There is a data type available for both the instance and the scratch memory, namely `SSRC_Instance_t` and `SSRC_Scratch_t`. The instance type is defined as structures of the correct size in the SSRC header file. Both the instance and the scratch memory must be 4 bytes aligned.

Parent topic:Type definitions

LVM_Fs_en Definition:

```
typedef enum
{
    LVM_FS_8000    = 0,
    LVM_FS_11025   = 1,
```

(continues on next page)

(continued from previous page)

```

LVM_FS_12000 = 2,
LVM_FS_16000 = 3,
LVM_FS_22050 = 4,
LVM_FS_24000 = 5,
LVM_FS_32000 = 6,
LVM_FS_44100 = 7,
LVM_FS_48000 = 8
} LVM_Fs_en;

```

Description:

Used to pass the input and the output sample rate to the SSRC.

Parent topic:Type definitions

LVM_Format_en Definition:

```

typedef enum
{
    LVM_STEREO          = 0,
    LVM_MONOINSTEREO    = 1,
    LVM_MONO             = 2
} LVM_Format_en;

```

Description:

The LVM_Format_en enumerated type is used to set the value of the SSRC data format.

The SSRC supports input data in two formats Mono and Stereo. For an input buffer of NumSamples = N (meaning N sample pairs for Stereo and MonoInStereo or N samples for Mono), the format of data in the buffer is as listed in *Table 1*:

Sample Number	Stereo	MonoInStereo	Mono
0	Left(0)	Mono(0)	Mono(0)
1	Right(0)	Mono(0)	Mono(1)
2	Left(1)	Mono(1)	Mono(2)
3	Right(1)	Mono(1)	Mono(3)
4	Left(2)	Mono(2)	Mono(4)
“	“	“	“
“	“	“	“
N-2	Left(N/2-1)	Mono(N/2-1)	Mono(N-2)
N-1	Right(N/2-1)	Mono(N/2-1)	Mono(N-1)
N	Left(N/2)	Mono(N/2)	Not Used
N+1	Right(N/2)	Mono(N/2)	Not Used
N+2	Left(N/2+1)	Mono(N/2+1)	Not Used
N+3	Right(N/2+1)	Mono(N/2+1)	Not Used
“	“	“	Not Used
“	“	“	Not Used
2*N-2	Left(N-1)	Mono(N-1)	Not Used

Parent topic:Type definitions

SSRC_Quality_en Definition:

```

typedef enum
{
    SSRC_QUALITY_HIGH    = 0,

```

(continues on next page)

(continued from previous page)

```

    SSRC_QUALITY_VERY_HIGH    = 1,
    SSRC_QUALITY_DUMMY        = LVM_MAXENUM
} SSRC_Quality_en;

```

Description:

Used to select the quality level of the SSRC. For details, see Performance figures. Selecting the highest-quality level, comes with a cost in the SSRC processing requirements. Therefore, it should only be done for critical applications.

Parent topic:Type definitions

Instance parameters Definition:

```

typedef struct
{
    SSRC_Quality_en    Quality;
    LVM_Fs_en          SSRC_Fs_In;
    LVM_Fs_en          SSRC_Fs_Out;
    LVM_Format_en      SSRC_NrOfChannels;
    short              NrSamplesIn;
    short              NrSamplesOut;
} SSRC_Params_t;

```

Description:

Used to pass the SSRC instance parameters to the SSRC module. It is a structure that contains the members for input sample rate, output sample rate, the number of channels, and the number of samples on the input and output audio stream.

Parent topic:Type definitions

Nr of samples mode Definition:

```

typedef enum
{
    SSRC_NR_SAMPLES_DEFAULT    = 0,
    SSRC_NR_SAMPLES_MIN        = 1,
    SSRC_NR_SAMPLES_DUMMY      = LVM_MAXENUM
} SSRC_NR_SAMPLES_MODE_en;

```

Description:

The SSRC_NR_SAMPLES_MODE_en enumerated type specifies the two different modes that can be used to retrieve the number of samples using the SSRC_GetNrSamples function.

Parent topic:Type definitions

Function return status Definition:

```

typedef enum
{
    SSRC_OK                  = 0,
    SSRC_INVALID_FS          = 1,
    SSRC_INVALID_NR_CHANNELS = 2,
    SSRC_NULL_POINTER        = 3,
    SSRC_WRONG_NR_SAMPLES    = 4,
    SSRC_ALIGNMMENT_ERROR    = 5,

```

(continues on next page)

(continued from previous page)

```

SSRC_INVALID_MODE      = 6,
SSRC_INVALID_VALUE     = 7,
SSRC_ALLINGMENT_ERROR  = 8,
LVXXX_RETURNSTATUS_DUMMY = LVM_MAXENUM
} SSRC_ReturnStatus_en;

```

Description:

The SSRC_ReturnStatus_en enumerated type specifies the different error codes returned by the API functions. For the exact meaning, see the individual function descriptions.

Parent topic:Type definitions

Parent topic:[Application programmers interface \(API\)](#)

Functions This section lists all the API functions of the SSRC module and explains their parameters.

SSRC_GetNrSamples Prototype:

```

SSRC_ReturnStatus_en SSRC_GetNrSamples
(SSRC_NR_SAMPLES_MODE_en Mode,
 SSRC_Params_t*       pSSRC_Params );

```

Description:

This function retrieves the number of samples or sample pairs for stereo used as an input and as an output of the SSRC module.

Narr	Type	Description
Mod	SSRC_NrSamples	There are two modes: - SSRC_NR_SAMPLES_DEFAULT: In this mode, the function returns the number of samples for 40 ms blocks - SSRC_NR_SAMPLES_MIN: the function returns the minimal number of samples supported for this conversion ratio. The SSRC_Init function accepts each integer multiple of this ratio. Formula: blocksize (ms) = 1/gcd(Fs_In,Fs_Out)
pSSRC_Params	SSRC_Params_t*	Pointer to the instance parameters. The application fills in the values of the input sample rate, the output sample rate, and the number of channels. Based on this input, the SSRC_GetNrSamples fills in the values for the number of samples for the input and the output audio stream.

Returns:

SSRC_OK	When the function call succeeds.
SSRC_INVALID_FS	When the requested input or output sampling rates are invalid.
SSRC_INVALID_NR_CHANN	When the channel format is not equal to LVM_MONO or LVM_STEREO.
SSRC_NULL_POINTER	When pSSRC_Params is a NULL pointer.
SSRC_INVALID_MODE	When mode is not a valid setting.

Note: The SSRC_GetNrSamples function returns the values from the following tables. Instead of calling the SSRC_GetNrSamples function, use the values from these tables directly.

Sample rate	Nr of samples
8000	320
11025	441
12000	480
16000	640
22050	882
24000	960
32000	1280
44100	1764
48000	1920

In/Out	8000	11025	12000	16000	22050	24000	32000	44100	48000
8000	11	320441	23	12	160441	13	14	80441	16
11025	441320	11	147160	441640	12	147320	4411280	14	147640
12000	32	160147	11	34	80147	12	38	40147	14
16000	21	640441	43	11	320441	23	12	160441	13
22050	441160	21	14780	441320	11	147160	441640	12	147320
24000	31	320147	21	32	160147	11	34	80147	12
32000	41	1280441	83	21	640441	43	11	320441	23
44100	44180	41	14740	441160	21	14780	441320	11	147160
48000	61	640147	41	31	320147	21	32	160147	11

Parent topic:Functions

SSRC_GetScratchSize Prototype:

```
SSRC_ReturnStatus_en SSRC_GetScratchSize
(SSRC_Params_t* pSSRC_Params,
 LVM_INT32* pScratchSize );
```

Description:

This function retrieves the scratch size for a given conversion ratio and for given buffer sizes at the input and at the output.

Name	Type	Description
pSSRC_Par	SSRC_Param	Pointer to the instance parameters. All members should have a valid value.
pScratch-Size	LVM_INT32*	Pointer to the scratch size. The SSRC_GetScratchSize function fills in the correct value (in bytes).

|

Returns:

SSRC_OK	When the function call succeeds.
SSRC_INVALID_FS	When the requested input or output sampling rates are invalid.
SSRC_INVALID_NR_CHANN	When the channel format is not equal to LVM_MONO or LVM_STEREO.
SSRC_NULL_POINTER	When pSSRC_Params or pScratchSize is a NULL pointer.
SSRC_WRONG_NR_SAMPLI	When the number of samples on the input or on the output are incorrect.

Parent topic:Functions

SSRC_Init Prototype:

```
SSRC_ReturnStatus_en SSRC_Init
(SSRC_Instance_t* pSSRC_Instance,
 SSRC_Scratch_t* pSSRC_Scratch,
 SSRC_Params_t* pSSRC_Params,
 LVM_INT16** ppInputInScratch,
 LVM_INT16** ppOutputInScratch);
```

Description:

The SSRC_Init function initializes an instance of the SSRC module.

Name	Type	Description
pSSRC_	SSRC_	Pointer to the instance of the SSRC. This application must allocate the memory before calling the SSRC_Init function.
pSSRC_	SSRC_	Pointer to the scratch memory. The pointer is saved inside the instance and is used by the SSRC_Process function. The application must allocate the scratch memory before calling the SSRC_Init function.
pSSRC_	SSRC_	Pointer to the instance parameters.
ppIn- putIn- Scratch	LVM_I	The SSRC module can be called with the input samples located in scratch. This pointer points to a location that holds the pointer to the location in the scratch memory that can be used to store the input samples. For example, to save memory.
ppOut- putIn- Scratch	LVM_I	The SSRC module can store the output samples in the scratch memory. This pointer points to a location that holds the pointer to the location in the scratch memory that can be used to store the output samples. For example, to save memory.

Returns:

SSRC_OK	When the function call succeeds.
SSRC_INVALID_FS	When the requested input or output sampling rates are invalid.
SSRC_INVALID_NR_CHANN	When the channel format is not equal to LVM_MONO or LVM_STEREO.
SSRC_NULL_POINTER	When pSSRC_Params or pScratchSize is a NULL pointer.
SSRC_WRONG_NR_SAMPLI	When the number of samples on the input or on the output are incorrect.
SSRC_ALIGNMENT_ERROR	When the instance memory or the scratch memory is not 4 bytes aligned.

Parent topic:Functions

SSRC_SetGains Prototype:

```
SSRC_ReturnStatus_en SSRC_SetGains
(SSRC_Instance_t* pSSRC_Instance,
 LVM_Mode_en      bHeadroomGainEnabled,
 LVM_Mode_en      bOutputGainEnabled,
 LVM_INT16        OutputGain);
```

Description:

This function sets headroom gain and the post gain of the SSRC. The SSRC_SetGains function is an optional function that should be used only in rare cases. Preferably, use the default settings.

Name	Type	Description
pSSRC	SSRC	Pointer to the instance of the SSRC.
bHeadroomGainEnabled	LVM	Parameter to enable or disable the headroom gain of the SSRC. The default value is LVM_MODE_ON. LVM_MODE_OFF can be used if it can be guaranteed that the input level is below -6 in all cases (the default headroom is -6 dB).
bOutputGainEnabled	LVM	Parameter to enable or disable the output gain. The default value is LVM_MODE_ON.
OutputGain	LVM	The value of the output gain. The output gain is a linear gain value. 0x7FFF is equal to +6 dB and 0x0000 corresponds to -inf dB. By default, a 3 dB gain is applied (OutputGain = 23197), resulting in an overall gain of -3 dB (-6 dB headroom +3 dB output gain). Unit Q format Data Range Default value Linear gain Q1.14 [0;32767] 23197

Returns:

SSRC_OK	When the function call succeeds
SSRC_NULL_POINTER	When pSSRC_Instance is a NULL pointer
SSRC_INVALID_MO	Wrong value used for the bHeadroomGainEnabled or the OutputGainEnabled parameters.
SSRC_INVALID_VAI	When OutputGain is out of the range [0;32767].

Parent topic:Functions

SSRC_Process Prototype:

```
SSRC_ReturnStatus_en SSRC_Process  
(SSRC_Instance_t* pSSRC_Instance,  
 LVM_INT16*      pSSRC_AudioIn,  
 LVM_INT16*      pSSRC_AudioOut);
```

Description:

Process function for the SSRC module. The function takes pointers as input and output audio buffers.

The sample format used for the input and output buffers is 16-bit little-endian. Stereo buffers are interleaved (L1, R1, L2, R2, and so on), mono buffers are deinterleaved (L1, L2, and so on).

Name	Type	Description
pSSRC_Instance	SSRC_Instance_t*	Pointer to the instance of the SSRC.
pSSRC_AudioIn	LVM_INT16*	Pointer to the input samples.
pSSRC_AudioOut	LVM_INT16*	Pointer to the output samples.

Returns:

SSRC_OK	When the function call succeeds.
SSRC_NULL_POINTER	When one of pSSRC_Instance, pSSRC_AudioIn, or pSSRC_AudioOut is NULL.

Parent topic:Functions

SSRC_Process_D32 Prototype:

```
SSRC_ReturnStatus_en SSRC_Process_D32  
(SSRC_Instance_t* pSSRC_Instance,  
 LVM_INT32*      pSSRC_AudioIn,  
 LVM_INT32*      pSSRC_AudioOut);
```

Description:

Process function for the SSRC module. The function takes pointers as input and output audio buffers.

The sample format used for the input and output buffers is 32-bit little-endian. Stereo buffers are interleaved (L1, R1, L2, R2, and so on), mono buffers are deinterleaved (L1, L2, and so on).

Name	Type	Description
pSSRC_Instance	SSRC_Instance_t*	Pointer to the instance of the SSRC.
pSSRC_AudioIn	LVM_INT32*	Pointer to the input samples.
pSSRC_AudioOut	LVM_INT32*	Pointer to the output samples.

Returns:

|SSRC_OK|When the function call succeeds.| |SSRC_NULL_POINTER|When one of pSSRC_Instance, pSSRC_AudioIn, or pSSRC_AudioOut is NULL.|

Parent topic:Functions

Parent topic:[Application programmers interface \(API\)](#)

Dynamic function usage This chapter explains how and when the SSRC functions are or can be used.

Define the number of samples to be used on input and output Call the function `SSRC_GetNrSamples`. Each integer multiple of the returned number of samples can be used.

Parent topic:Dynamic function usage

Allocate scratch memory To calculate the required size of the scratch memory, call the `SSRC_GetScratchSize` function. Allocate memory for the returned size.

Parent topic:Dynamic function usage

Initialize the SSRC instance Call the `SSRC_Init` function.

Parent topic:Dynamic function usage

Process samples The `SSRC_Process` function can now be called any number of times.

Parent topic:Dynamic function usage

Destroy the SSRC instance When the processing is completed, the allocated memory for the instance and the scratch can be freed.

Parent topic:Dynamic function usage

Parent topic:[Application programmers interface \(API\)](#)

Reentrancy None of the SSRC functions are re-entrant.

Parent topic:[Application programmers interface \(API\)](#)

Additional user information This section provides information on the Attenuation of the signal and Notes on integration.

Attenuation of the signal When a fully saturated or clipped input is applied to an SRC module, the aliases after the sample rate conversion, although sufficiently suppressed, can still result in a clipped output. To prevent clipped output, the output of the SSRC module is by default attenuated with 3 dB. Although not advised, this gain value can be changed using the `SSRC_SetGains` function.

Parent topic:[Additional user information](#)

Notes on integration Although the sample rate converter module works with audio signals on different sampling rates, it is a synchronous module. The module takes a block of input samples, consumes the input completely, and produces a full buffer with output samples. As a result, the SSRC only accepts a limited number of input and output block sizes. To flush last, incomplete, block of an audio stream, the block is padded with zeros until it is full before the SSRC processes it.

Parent topic:[Additional user information](#)

Example application The source code of the example application can be found in the `.\EX_APP\APP_FileIO\SRC` directory of the release package. The `.\EX_APP\APP_FileIO\MAKE` directory contains a make file that can be used to build the example application. When building the application, an executable is generated in the `.\EX_APP\APP_FileIO\EXE` directory.

The example application takes as command-line input parameters:

1. The path toward the input PCM file. It assumes raw 16 bit signed little-endian put. Stereo input samples should be interleaved (L1, L2 R1, R2,...), mono samples should be deinterleaved (L1, L2, and so on).
2. The path toward the output PCM file.
3. The input sample rate.
4. The output sample rate.
5. The channel format (mono or stereo).

Integration test A correct integration of the SSRC module can be verified in two ways.

- Bit accurate test
- THD+N measurement

Bit accurate test The TestFiles directory of the release package contains a test input (sampled at 44,100 Hz) and several expected output files (sample rates from 8000 Hz to 48,000 Hz). If the same test input file is applied to the SRC after integration in the target platform, the output is bit accurate with the expected output file that matches the output-sample rate

Parent topic:[Integration test](#)

THD+N measurement Produce a swept sine and feed it through the SSRC module. Do a THD+N measurement on the obtained output signal. The THD+N of the converted signals should be below -77 in the interval [0 - 0.45] FsLOW.

Parent topic:[Integration test](#)

Maestro Audio Framework

MCUXpresso SDK : Maestro

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

Documentation Overall details can be reviewed here: [MCUXpresso SDK Online Documentation](#)

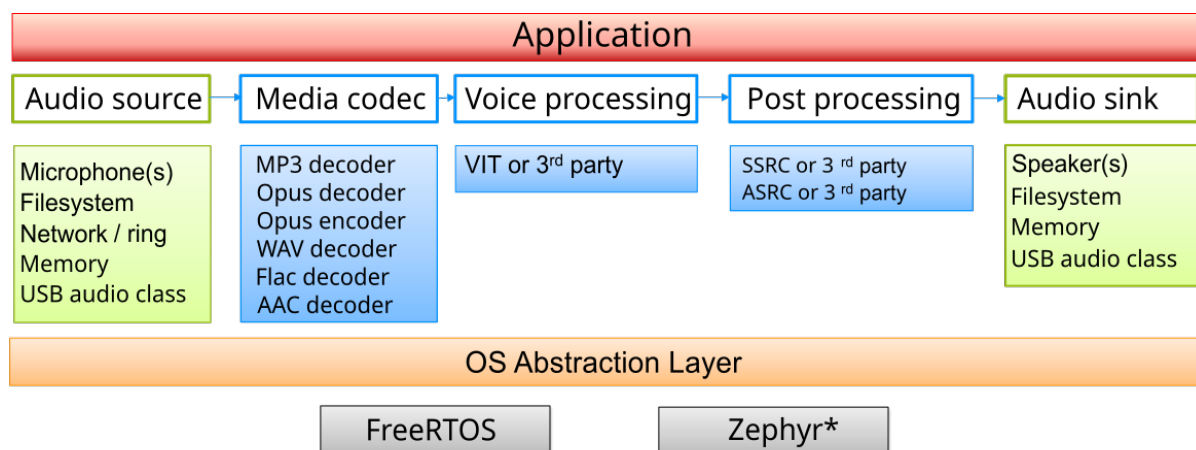
Visit [Maestro - Documentation](#) to review details on the contents in this sub-repo.

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

Contribution We welcome and encourage the community to submit patches directly to the Maestro project placed on github. Contributing can be managed via pull-requests.

Introduction Maestro audio framework intends to enable chaining of basic audio processing blocks, called *elements*. These blocks then form stream processing objects, called *pipeline*. This pipeline can be used for multiple audio processing use cases.

The processing blocks can include (but are not limited to) different audio sources (for example file or microphone), decoders or encoders, filters or effects, and audio sinks. Framework overview is depicted in the following picture:



*not all elements and libraries are supported in Zephyr port. For more information, see [Maestro on Zephyr](#)

The Maestro audio framework is an open-source component developed by NXP Semiconductors and released under the BSD-compatible license. It is running on RTOS (Zephyr or FreeRTOS), abstracted by OSA layer.

For detailed description of the audio Maestro framework, please refer to the [programmer's guide](#).

To see what is new, see [changelog](#).

Maestro on Zephyr Getting started guide and further information for Maestro on Zephyr may be found [here](#).

Maestro on FreeRTOS Maestro on FreeRTOS is supported in NXP's SDK. To get started, see [mcuxsdk doc](#).

Supported examples The current version of the Maestro audio framework supports several optional *features*, some of which are used in these examples:

- *maestro_playback*
- *maestro_record*
- *maestro_usb_mic*
- *maestro_usb_speaker*
- *maestro_sync*

The examples can be found in the **audio_examples** folder of the desired board. The demo applications are based on FreeRTOS and use multiple tasks to form the application functionality.

Example applications overview To set up the audio framework properly, it is necessary to create a streamer with `streamer_create` API. It is also essential to set up the desired hardware peripherals using the functions described in `streamer_pcm.h`. The Maestro example projects consist of several files regarding the audio framework. The initial file is `main.c` with code to create multiple tasks. For features including SD card (in the *maestro_playback* examples, reading a file from SD card is supported and in *maestro_record* writing to SD card is currently supported) the `APP_SDCARD_Task` is created. The command prompt and connected functionalities are handled by `APP_Shell_Task`.

One of the most important parts of the configuration is the `streamer_pcm.c` where the initialization of the hardware peripherals, input and output buffer management can be found. For further information please see also `streamer_pcm.h`

In the Maestro USB examples (*maestro_usb_mic* and *maestro_usb_speaker*), the USB configuration is located in the `usb_device_descriptor.c`, `audio_microphone.c` and `audio_speaker.c` files. For further information please see also `usb_device_descriptor.h`, `audio_microphone.h` and `audio_speaker.h`.

In order to be able to get the messages from the audio framework, it is necessary to create a thread for receiving the messages from the streamer, which is usually called a Message Task. The message thread is placed in the `app_streamer.c` file, reads the streamer message queue, and reacts to the following messages:

- `STREAM_MSG_ERROR` - stops the streamer and exits the message thread
- `STREAM_MSG_EOS` - stops the streamer and exits the message thread
- `STREAM_MSG_UPDATE_DURATION` - prints info about the stream duration
- `STREAM_MSG_UPDATE_POSITION` - prints info about current stream position
- `STREAM_MSG_CLOSE_TASK` - exits the message thread

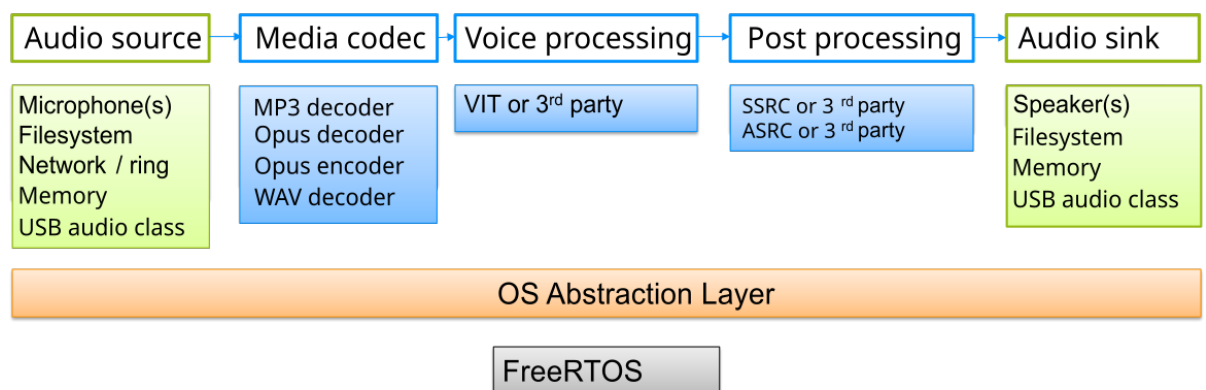
File structure

Folder	Description
src	Maestro audio framework sources
src/inc	Maestro include files
src/core	Maestro core sources
src/cci	Common decoder interface sources
src/cei	Common encoder interface sources
src/elements	Maestro elements sources
src/devices	External audio devices implementation (audio source & audio sink elements)
src/utlis	Helper utilities utilized by Maestro
docs	Generated documentation
doxygen	Documentation sources
components	Glue for audio libraries, so they can be used in elements
tests	Maestro tests
zephyr/	Zephyr related files
zephyr/samples/	Zephyr samples
zephyr/tests/	Zephyr tests
zephyr/audioTracks/	Audio tracks for testing
zephyr/wrappers/	Zephyr NXP SDK Wrappers
zephyr/doc/	Zephyr documentation configuration for Sphinx
zephyr/scripts/	Zephyr helper scripts, mostly for testing

Maestro Audio Framework Programmer's Guide

Introduction Maestro audio framework provides instruments for playback and capture of different audio streams. In order to do that the framework uses API for creating various audio and voice pipelines with the support of media and track information. This document describes the framework in its detail, and the usage of API for pipeline creation using different elements. The framework needs an operating system in order to create different tasks for audio processing and communication with the application.

Architecture overview A high-level block diagram of the streamer used in Maestro is shown below. An element is the most important class of objects in the streamer (see `streamer_element.c`). A chain of elements will be created and linked together when a *pipeline* is created. Data flows through this chain of elements in form of data buffers. An element has one specific function, which can be the reading of data from a file, decoding of this data, or outputting this data to a sink device. By chaining together several such elements, a pipeline is created that can do a specific task, for example, the playback.



Pipeline

The pipeline is created within the `streamer_create` API using the `streamer_create_pipeline` call. In the example applications provided in the MCUXpresso SDK the pipeline is created in the `app_streamer.c` file. In order to create a pipeline user needs to provide a `PipelineElements` structure consisting of array of element indexes `ElementIndex` and the number of elements in the pipeline. Then the pipeline is built automatically and user can specify the properties of the elements using the `streamer_set_property` API. All the element properties can be found in the `streamer_element_properties.h` file.

The streamer can handle up to two pipelines within a single task. The first pipeline with index 0 can be created using the `streamer_create` function as described above. Then the `streamer_create_pipeline` function should be used to create the second pipeline (pipeline with index 1). An example creation can be found in the `app_streamer.c` file in the [maestro_sync_example](#). Both pipelines are processed sequentially, so after the first pipeline is processed, the second pipeline is processed.

After the pipeline is successfully created, all elements and entire pipeline are in `STATE_NULL` state. A user can start the streamer by setting the pipeline state to `STATE_PLAYING` using the `streamer_set_state` function. The pipeline can also be paused or stopped using the same function. Use the `STATE_PAUSED` to pause and use `STATE_NULL` to stop. The function changes the state of each element that is in the pipeline in turn, and after all the elements have obtained the desired state, the state of entire pipeline is changed.

Elements The current version of the Maestro framework supports several types of elements (`StreamElementType`). In each pipeline should be used one source element (elements with the `_SRC` suffix) and one sink element (elements with the `_SINK` suffix). A decoder, encoder or `audio_proc` element can be connected between these two elements. The `audio_proc` element can be used more than once within the same pipeline.

Each element type (`StreamElementType`) has several functions that are determined by a unique element index (`ElementIndex`). These indexes are used to create a pipeline, and each element index can only be used once in the same pipeline. The `type_lookup_table` shows which `StreamElementType` supports which `ElementIndex`.

Each element index (`ElementIndex`) has its own properties and a list of these properties can be found in the `streamer_element_properties.h` file. These properties are divided into groups and each group is identified by a property mask (e.g. for speaker it is `PROP_SPEAKER_MASK`). Then the `property_lookup_table` in the `streamer_msg.c` file determines which property group relates to which element index (`ElementIndex`). When an element is created and added to the pipeline, its properties are set to their default values. Default values can be seen in the initialization function of a particular element. The initialization functions are specified in the `element_list` array in the `streamer_element.c` file (e.g. for the `audio_proc` element it is the `audio_proc_init_element` function). The user can get the value of the property using the `streamer_get_property` function or change its value using the `streamer_set_property` function.

The source code of the elements can be found in the `middleware\audio_voice\maestro\src\elements\` folder.

Add a new element type The user can add a new element type (`StreamElementType`) to the Maestro audio framework. For this, the following steps need to be done.

- Add a new element type to the `StreamElementType` enum type in the `streamer_api.h`.
- Create a new `*.c` and `*.h` files for the new element type in the `middleware\audio_voice\maestro\src\elements\` folder. All necessary structures and functions (functions for src pads, sink pads and element itself) needs to be defined in these files. Inspiration can be found in other elements.
- Link the initialization function to the element type in the `element_list` array in the `streamer_element.c` file. To do this, a new definition that enables the element needs to be

created (e.g. there is a `STREAMER_ENABLE_AUDIO_PROC` definition for the `audio_proc` element).

- Associate the newly created element type with an element index (`ElementIndex`) by adding a new pair to the `type_lookup_table` in the `streamer.c` file.
- If the user wants to use the newly created element in an application, the definition that enables the element must be defined at the project level.

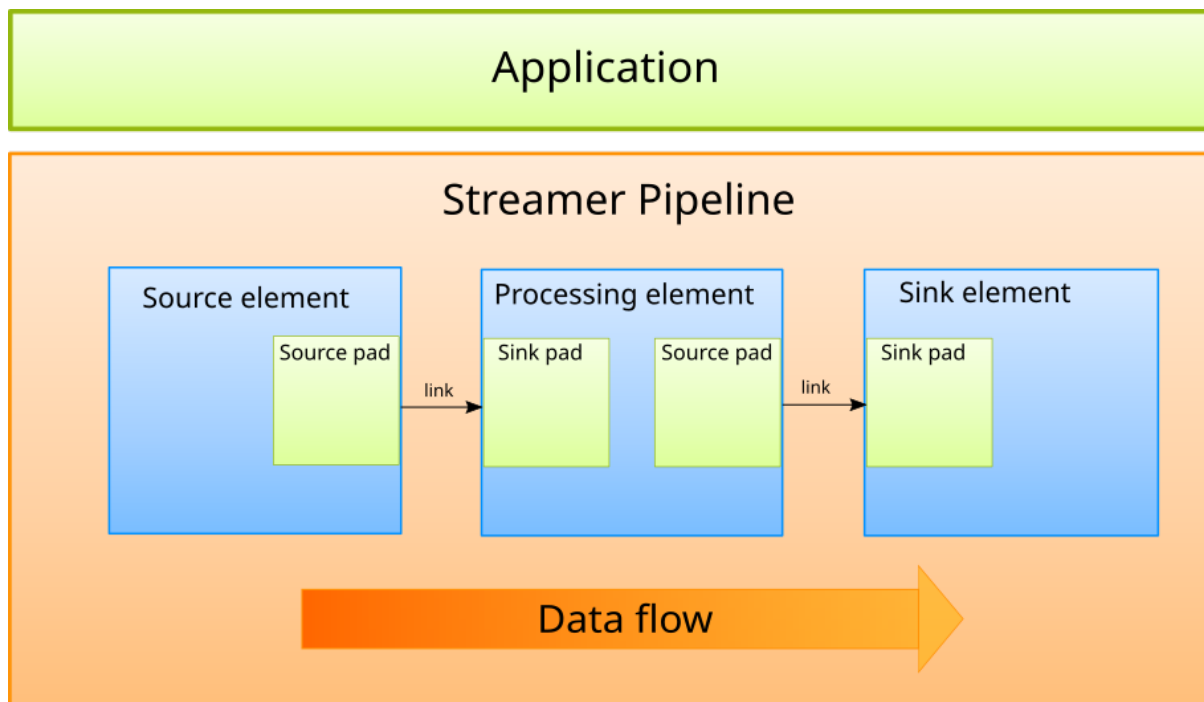
Mostly the user doesn't need to create a new element type, but just create an element index.

Add a new element index To create a new element index in the Maestro audio framework, follow these steps:

- Add a new element index to the `ElementIndex` enum type in the `streamer_api.h`.
- Create the required properties for the newly created element index in the `streamer_element_properties.h` file.
- Associate the newly created property group with newly created element index by adding a new pair to the `property_lookup_table` in the `streamer_msg.c` file.
- Associate the newly created element index with an element type (`StreamElementType`) by adding a new pair to the `type_lookup_table` in the `streamer.c` file.
- Add support for the created properties to functions of the associated element type. These functions are defined in files that correspond to a particular element type. The files are located in the `middleware\audio_voice\maestro\src\elements\` folder.

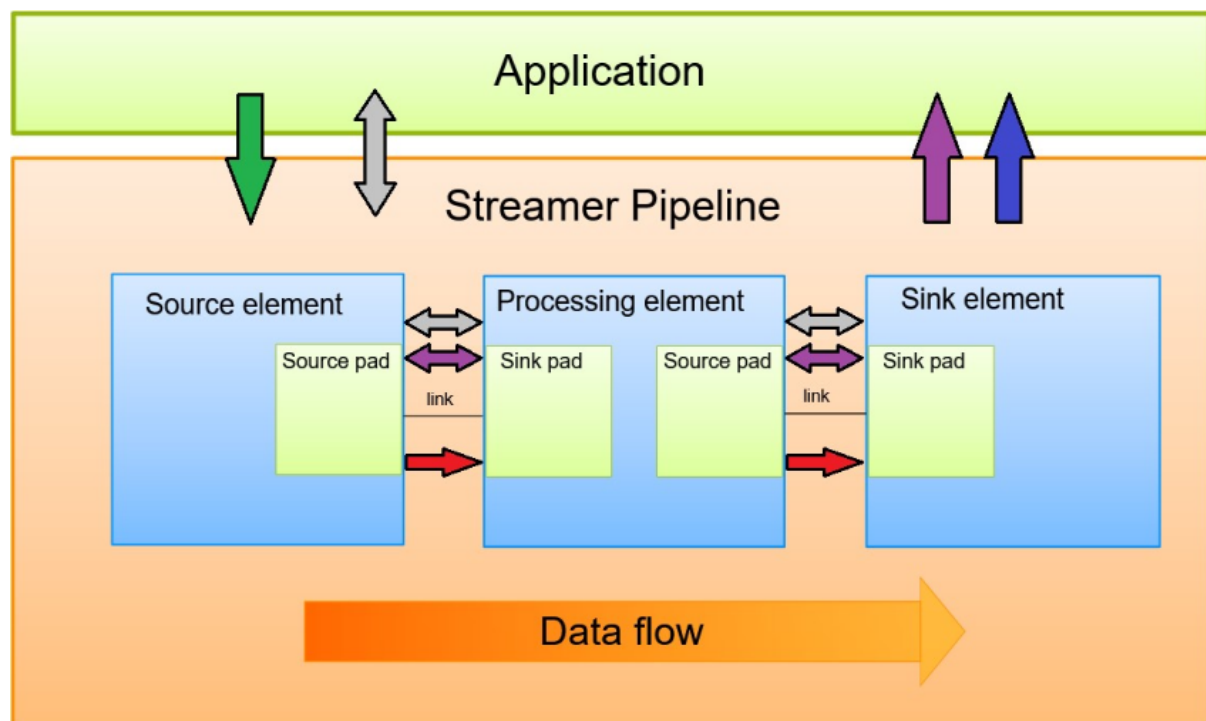
It is important to know that each element type (`StreamElementType`) can be associated with more than one element index (`ElementIndex`), but each element index (`ElementIndex`) can be associated with only one element type (`StreamElementType`).

Pads Pads are elements' inputs and outputs. A pad can be viewed as a “plug” or “port” on an element where links may be made with other elements, and through which data can flow to or from those elements. Data flows out of an element through a source pad, and elements accept incoming data through a sink pad. Source and sink elements have only source and sink pads, respectively. For detailed information about pads, please see the API reference from `pad.c`.



Internal communication The streamer (the core of the framework) provides several mechanisms for communication and data exchange between the application, a pipeline, and pipeline elements:

- Buffers are objects for passing streaming data between elements in the pipeline. Buffers always travel from sources to sinks (downstream).
- Messages are objects sent from the application to the streamer task to construct, configure, and control a streamer pipeline.
- Callbacks are used to transmit information such as errors, tags, state changes, etc. from the pipeline and elements to the application.
- Events are objects sent between elements. Events can travel upstream and downstream. Events may also be sent to the application.
- Queries allow applications to request information such as duration or current playback position from the pipeline. Elements can also use queries to request information from their peer elements (such as the file size or duration). They can be used both ways within a pipeline, but upstream queries are more common.



Decoders and encoders Maestro framework uses a common codec interface for decoding purposes and a common encoder interface for encoding. Those interfaces encapsulate the usage of specific codecs. Reference codecs are available in `audio-voice-components` repository which should be in `\middleware\audio_voice\components\` folder.

Common codec interface The Common Codec Interface is the intended interface for all used **decoders**. The framework will integrate a CCI decoder element into the streamer to interface with all decoders.

Using the CCI to interface with Metadata

- `cci_extract_meta_data` must be called before any other Codec Interface APIs. This API extracts the metadata information of the codec and fills this information in the

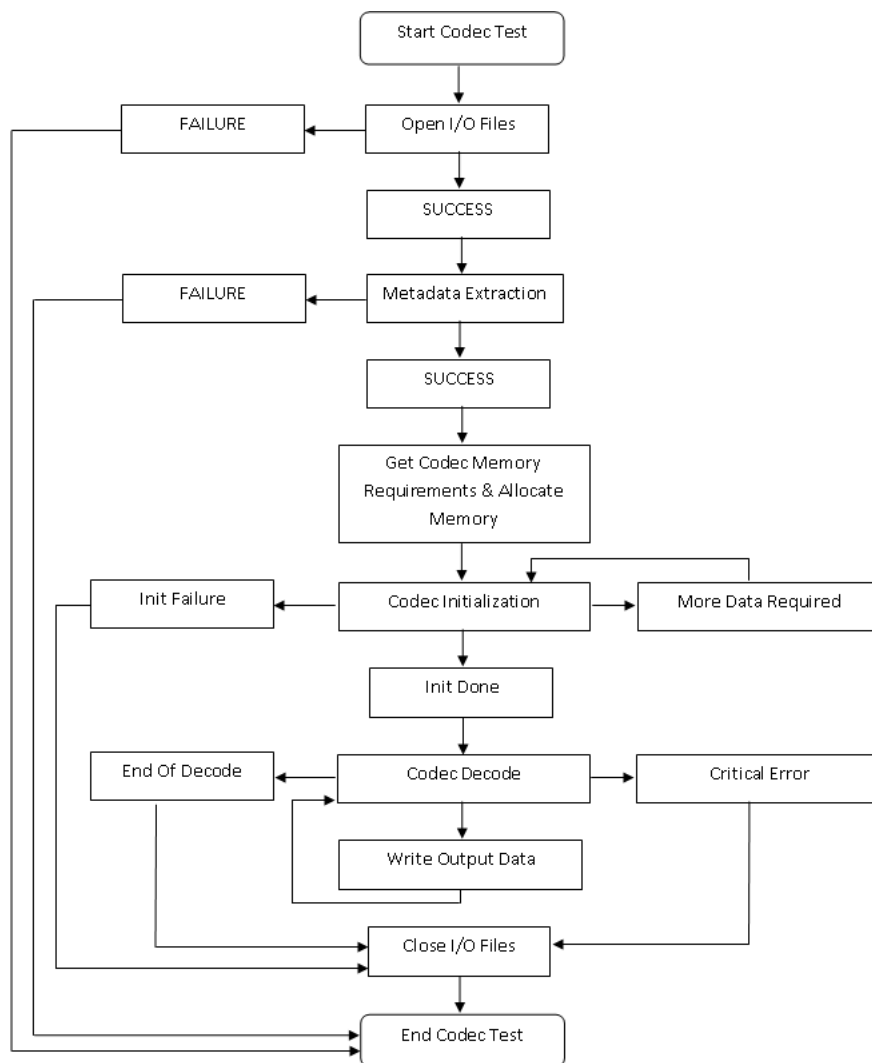
file_meta_data_t structure. The file_meta_data_t structure must be allocated by the application.

- This function first extracts the input file extension and based on that it calls the specific codec's metadata extraction function. If it finds an invalid extension or unsupported extension then it returns with META_DATA_FILE_NOT_SUPPORTED code for any unsupported file format.
- If this API finds the valid metadata then it returns with META_DATA_FOUND code. If this API does not find any metadata information then it returns with META_DATA_NOT_FOUND code. It also returns with META_DATA_FILE_NOT_SUPPORTED code for any unsupported file format.

Using the CCI to interface with Decoders

- `codec_get_mem_info` gets the memory requirement based on the specific decoder stream type. It returns the size in bytes of the specific codec. The user of the decoders must allocate memory of this size and this memory is used by the initialization API. The user or application must pass this allocated memory pointer to the init API.
- `codec_init` must be called before the codec's decode API. This API calls the codec-specific initialization function based on the codec stream type. This API allocates the memory to the codec main structure and also initializes the codec main structure parameters. It also registers the call back functions to the codec which will be used by the codec to read or seek the input stream.
- `codec_decode` is the main decoding API of the codec. This API calls the codec-specific decoding function based on the codec stream type. This API decodes the input raw stream and fills the PCM output samples into codec output PCM buffer. This API gives the information about the number of samples produced by the codec and also gives the pointer of the codec output PCM samples buffer.
- `codec_get_pcm_samples` must be called after the codec's decode API. This API calls the codec specific Get PCM Sample API based on the codec stream type. This API gets the PCM samples from the codec in constant block size and fills them into the output PCM buffer. It returns the number of samples get from the codec and also gives the pointer of the output PCM buffer.
- `codec_reset` calls the codec specific reset API base on stream type and resets the codec.
- `codec_seek` accepts the seek bytes offset converted from the time by application. This API calls the decoder's internal seek API to calculate the actual seek offset which frame boundary aligns. This API returns the actual seek offset.

The basic sequence to use a decoder with the CCI is shown below:



Adding new decoders to the CCI This section explains how to integrate a new decoder in the Common Codec Interface. The CCI assumes the decoder library to be used is in the `\middleware\audio_voice\audiocomponents\decoders\*decoder*\libs\` folder of the maestro framework. The CCI is just a wrapper around a specific implementation. The decoder is expected to be extended as needed to meet the APIs described above.

- Register Decoder Top level APIs in Common Codec Interface
 - Place the decoder lib in libs folder.
 - Add prototypes of the decoder top level APIs in `codec_interface.h` file (located at `maestro\src\cci\inc\` folder).
 - In `codec_interface.c` file (located at `maestro\src\cci\src\`), add top level Decoder APIs in decoder function table.
 - Pseudo code for this is as described below.

```

const codec_interface_function_table_t g_codec_function_table[STREAM_TYPE_COUNT] = {
#ifdef VORBIS_CODEC
{
    &VORBISDecoderGetMemorySize,
    &VORBISDecoderInit,
    &VORBISDecoderDecode,
    NULL,

```

(continues on next page)

(continued from previous page)

```

    NULL,
    &VORBISDecoderSeek,
    &VORBISDecoderGetIOFrameSize,
},
#else
{
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
}
#endif
};

```

- Enable or Disable Decoder
 - Define VORBIS_CODEEC macro in audio_cfg.h file.
 - Comment this macro if you want to disable VORBIS Decoder otherwise keep it defined in order to enable the decoder.
- Add Extract Metadata API for the decoder
 - Add extract metadata API source file for the decoder at streamer/cci/metadata/src/vorbis folder.
 - Add this code in extract metadata lib project space.
 - Build the extract metadata lib and copy that lib to libs folder.
 - Add the desired stream type into ccidec_extract_meta_data API (in codecextractmetadata.c file) to call VORBIS Decoder extract metadata API.
- Add stream type of the new decoder in the stream type enum audio_stream_type_t in codec_interface_public_api.h
 - Stream type of the decoder in stream type enum and decoder APIs in decoder function table must be in the same sequence.

Common encoder interface Please see the following section about the [cei](#).

Maestro performance

Memory information The memory usage of the framework components using reference codecs (data obtained from GNU ARM compiler) in bytes is:

text	data	bss	component
48790	2752	4	aac decoder
4348	16400	212	asrc
15512	0	4	flac decoder
76462	16	5013	maestro
34211	0	4	mp3 decoder
211974	0	0	opus
65446	0	4	ssrc
5850	16	12	wav decoder

Maestro framework uses dynamic allocation of audio buffers. The total amount of memory allocated for the pipeline depends on the following parameters:

- Number of elements in the pipeline
- Element types
- Audio stream properties
 - Sampling rate
 - Bit width
 - Channel number
 - Frame size

CPU usage The performance of the pipeline was measured using the real hardware platform (RT1060).

- CPU core clock in MHz: 600.

Pipeline type	Performance MIPS of pipeline (in MHz)
audio source -> audio sink	~10.26 MHz
audio source -> file sink	~9.84 MHz
file source (8-channel PCM) -> audio sink	~16.5 MHz

For performance details about the supported codecs please see audio-voice-components repository documentation.

CEI encoder The Maestro streamer contains an element adapting an extensible set of audio encoders in the form of functions conforming to the CEI (Common Encoder Interface). This element enables the user to choose and configure a suitable encoder at runtime.

Header files CEI itself and the CEI encoders are using following header files, in which you may be interested:

- `cei.h` - contains types used by the element itself and an encoder implementing the CEI
- `cei_enctypes.h` - contains a list of possible encoders and types used for interfacing with a CEI encoder
- `cei_table.h` - contains a table of functions implementing integrated CEI encoders

Instantiating the element This element's index is `ELEMENT_ENCODER_INDEX` and its type is `TYPE_ELEMENT_ENCODER`, as defined in `streamer_api.h`. It has one source pad (data input) and one sink pad (data output). It is initialized like any other element, meaning that it is instantiated and inserted into the pipeline using the `create_element`, `add_element_pipeline` and `link_elements` functions. Inversely, for destroying the element, the `unlink_elements`, `remove_element_pipeline` and `destroy_element` are used. This element alone does not depend on any additional software layers other than these required by the Maestro streamer itself, so no pre-initialization before this element instantiation is necessary.

Element properties Use Maestro streamer property API (`streamer_set_property` and `streamer_get_property`) for setting or getting these. The constants are defined in `streamer_element_properties.h`.

- `PROP_ENCODER_CHUNK_SIZE`

- **Synopsis:** Determines the length of a chunk pulled from the sibling of the source pad and essentially influences the size of allocated buffers. If the actual amount of data pulled is smaller, the rest is zero-filled.
- **Type:** unsigned 32-bit integer
- **Default value:** 1920
- **Constraints:**
 - * Must be bigger than zero, otherwise `STREAM_ERR_INVALID_ARGS` is returned.
 - * Cannot be changed if the actual encoder has been created. If done so, `STREAM_ERR_ELEMENT_BAD_STATUS` is returned.
- `PROP_ENCODER_TYPE`
 - **Synopsis:** Determines the exact encoder (CEI implementation) to be used.
 - **Type:** `CeiEncoderType` (`cei_enctypes.h`)
 - **Default value:** `CEIENC_LAST`
 - **Constraints:**
 - * Must not be equal to `CEIENC_LAST`, otherwise `STREAM_ERR_INVALID_ARGS` will be returned.
 - * Selected encoder must be implemented, otherwise `STREAM_ERR_INVALID_ARGS` will be returned.
 - * Cannot be changed if the actual encoder has been created. If done so, `STREAM_ERR_ELEMENT_BAD_STATUS` will be returned.
 - **Behaviour influenced:** The encoder element process function will return `FLOW_ERROR` if this property isn't set.
- `PROP_ENCODER_CONFIG`
 - **Synopsis:** Determines encoder-specific configuration (application, bitrate, ...).
 - **Type:** Pointer to the encoder-specific configuration structure.
 - **Default value:** Determined by the encoder.
 - **Constraints:**
 - * The encoder has to be configurable. If it is not, `STREAM_ERR_ERR_GENERAL` will be returned on any access.
 - * The structure has to conform to the encoder requirements. If the encoder returns an error code, `STREAM_ERR_GENERAL` will be returned.
- `PROP_ENCODER_BITSTREAMINFO`
 - **Synopsis:** Specifies information about the incoming bitstream (sample rate, sample depth, ...).
 - **Type:** Pointer to `CeiBitstreamInfo` (`cei_enctypes.h`).
 - **Default value:**

```
(CeiBitstreamInfo) {
    .sample_rate = 0,
    .num_channels = 0,
    .endian = AF_LITTLE_ENDIAN,
    .sign = TRUE,
    .sample_size = 0,
    .interleaved = TRUE
}
```

– **Constraints:**

- * Cannot be changed if the actual encoder has been created. If done so, `STREAM_ERR_ELEMENT_BAD_STATUS` will be returned.
- * As of now, only bitstreams containing 16-bit interleaved (if 2 or more channels will be encoded) samples are supported. If anything else was set to the `sample_size` and `interleaved_members`, `STREAM_ERR_INVALID_ARGS` will be returned.

– **Behaviour influenced:**

- * Given the characteristics of some elements available, different packets of data (header and payload, referred to as “chunk” above) may be pulled by this element. Each packet can contain a different header, which may or may not contain useful information about the bitstream. If a packet with the `AudioPacketHeader` (`todofile.h`) is pulled at first and any other iteration of the streamer pipeline, the bitstream parameters configured by this property are implicitly available and are not expected to be specified by the user. Other packet header types (such as `RawPacketHeader`) don’t contain any bitstream parameters and require the user to specify the parameters manually using this property. Failure to do so will result in the element’s process function returning `FLOW_ERROR`. Same situation will occur if a packet with the `AudioPacketHeader` is received and its contents differ from the already acquired bitstream parameters.
- * As of now, CEI is defined to work with 16-bit signed little-endian (s16le) samples, which are interleaved if the bitstream contains more than one channels. This element handles endianness and unsigned to signed conversion.

CEI definition - implementing your own encoder The CEI defines following function pointer types:

- `CeiFnGetMemorySize`: Returns number of bytes required for encoder state for a given number of channels.
- `CeiFnEncoderInit`: Initialize an encoder for a given sample rate and channel count.
- `CeiFnEncoderGetConfig`: Copy current or default configuration to a given structure pointer.
- `CeiFnEncoderSetConfig`: Configure the encoder from a given structure pointer.
- `CeiFnEncode`: Encode a given buffer to a given output buffer.

Detailed descriptions of function behaviour, parameters and expected return values are available as docblocks in the `cei.h` file.

Each encoder is implemented as a set of pointers pointing to functions conforming to these types, grouped in the `CeiEncoderFunctions` structure. Specifying the `CeiEncoderGetConfig` `fnGetConfig` and `CeiFnEncoderSetConfig` `fnSetConfig` members is optional, as an encoder does not have to be configurable. If so desired, specify `NULL`. Implementation of the remaining functions is mandatory, however. If at least one of these functions isn’t implemented and `NULL` is specified instead, the encoder will be considered as not implemented.

To register an implemented encoder with the element, add a new entry to the `CeiEncoderType` enum and add the `CeiEncoderFunctions` struct value to the table `CeiEncoderFunctions` `ceiEncTable[]` located in the `cei_table.h` header file. Note and match the order of items in that table, as a `CeiEncoderType` value is used as an index. Same goes for the `size_t` `ceiEncConfigSizeTable[]`. If configuration is not applicable, specify 0 at the appropriate index. If configuration is applicable, describe the configuration structure in the `cei_encetypes.h` header file and add its size to that table.

Maestro playback example

Table of content

- [Overview](#)
- [Hardware requirements](#)
- [Hardware modifications](#)
- [Preparation](#)
- [Running the demo](#)
- [Example configuration](#)
- [Functionality](#)
- [States](#)
- [Commands in detail](#)
- [Processing Time](#)

Overview The Maestro playback example demonstrates audio processing on the ARM cortex core utilizing the Maestro Audio Framework library.

The application is controlled by commands from a shell interface using serial console and the audio files are read from the SD card.

Depending on target platform or development board there are different modes and features of the demo supported.

- **Standard** - The mode demonstrates playback of encoded files from an SD card with up to 2 channels, up to 48 kHz sample rate and up to 16 bit width. This mode is enabled by default.
- **Multi-channel** - The mode demonstrates playback of raw PCM files from an SD card with 2 or 8 channels, 96kHz sample rate and 32 bit width. The decoders and synchronous sample rate converter are not supported in this mode. The Multi-channel mode is only supported on selected platforms, see the table below. The [Example configuration](#) section contains information on how to enable it.

As shown in the table below, the application is supported on several development boards and each development board may have certain limitations, some development boards may also require hardware modifications or allow to use of an audio expansion board. Therefore, please check the supported features and [Hardware modifications](#) or [Example configuration](#) sections before running the demo.

Limitations:

- Note:
 - *LPCXpresso55s69* - MCUXpresso IDE project default debug console is semihost
- Decoder:
 - **AAC:**
 - * The reference decoder is supported only in the MCUXpresso IDE and ARMGCC.
 - **FLAC:**
 - * *LPCXpresso55s69* - When playing FLAC audio files with too small frame size (block size), the audio output may be distorted because the board is not fast enough.
 - **OPUS:**
 - * *LPCXpresso55s69* - The decoder is disabled due to insufficient memory may be distorted because the board is not fast enough.
- Sample rate converter:
 - **SSRC:**

- * *LPCXpresso55s69* - When a memory allocation ERROR occurs, it is necessary to disable the SSRC element due to insufficient memory.

Known issues:

- Decoder:
 - **MP3:**
 - * The reference decoder has issues with some of the files. One of the channels can be sometimes distorted or missing parts of the signal.
 - **OPUS:**
 - * The decoder doesn't support all the combinations of frame sizes and sample rates. The application might crash when playing an unsupported file.

More information about supported features can be found on the [Supported features](#) page.

Hardware requirements

- Desired development board
- Micro USB cable
- Headphones with 3.5 mm stereo jack
- SD card with supported audio files
- Personal computer
- Optional:
 - Audio expansion board [AUD-EXP-42448 \(REV B\)](#)

Hardware modifications Some development boards need some hardware modifications to run the application. If the development board is not listed here, its default setting is required.

- *EVKB-MIMXRT1170:*
 1. Please remove below resistors if on board wifi chip is not DNP:
 - R228, R229, R232, R234
 2. Please make sure R136 is weld for GPIO card detect.

Preparation

1. Connect a micro USB cable between the PC host and the debug USB port on the development board.
2. Open a serial terminal with the following settings:
 - 115200 baud rate
 - 8 data bits
 - No parity
 - One stop bit
 - No flow control
3. Download the program to the target board.
4. Insert the headphones into the Line-Out connector (headphone jack) on the development board.

5. Either press the reset button on your development board or launch the debugger in your IDE to begin running the demo.

Running the demo When the example runs successfully, you should see similar output on the serial terminal as below:

```
*****
Maestro audio playback demo start
*****

[APP_Main_Task] started

Copyright 2022 NXP
[APP_SDCARD_Task] start
[APP_Shell_Task] start

>> [APP_SDCARD_Task] SD card drive mounted
```

Type `help` to see the command list. Similar description will be displayed on serial console (*If multi-channel playback mode is enabled, the description is slightly different*):

```
>> help

"help": List all the registered commands

"exit": Exit program

"version": Display component versions

"file": Perform audio file decode and playback

USAGE: file [stop|pause|volume|seek|play|list|info]
stop           Stops actual playback.
pause          Pause actual track or resume if already paused.
volume <volume> Set volume. The volume can be set from 0 to 100.
seek <seek_time> Seek currently paused track. Seek time is absolute time in milliseconds.
play <filename> Select audio track to play.
list           List audio files available on mounted SD card.
info           Prints playback info.
```

Details of commands can be found [here](#).

Example configuration The example can be configured by user. Before configuration, please check the [table](#) to see if the feature is supported on the development board.

- **Enable Multi-channel mode:**

- Add the MULTICHANNEL_EXAMPLE symbol to preprocessor defines on project level.
- Connect AUD-EXP-42448 (see the point below).

- **Connect AUD-EXP-42448:**

- *EVKC-MIMXRT1060:*
 1. Disconnect the power supply for safety reasons.
 2. Insert AUD-EXP-42448 into J19 to be able to use the CS42448 codec for multichannel output.
 3. Uninstall J99.
 4. Set the DEMO_CODEC_WM8962 macro to 0 in the `app_definitions.h` file

5. Set the DEMO_CODEC_CS42448 macro to 1 in the app_definitions.h file.

Functionality The file `play <filename>` command calls the `STREAMER_file_Create` or `STREAMER_PCM_Create` function from the `app_streamer.c` file depending on the selected mode.

- When the *Standard* mode is enabled, the command calls the `STREAMER_file_Create` function that creates a pipeline with the following elements:
 - `ELEMENT_FILE_SRC_INDEX`
 - `ELEMENT_DECODER_INDEX`
 - `ELEMENT_SRC_INDEX` (If `SSRC_PROC` is defined)
 - `ELEMENT_SPEAKER_INDEX`
- When the *Multi-channel* mode is enabled, the command calls `STREAMER_PCM_Create` function, which creates a pipeline with the following elements:
 - `ELEMENT_FILE_SRC_INDEX` (PCM format only)
 - `ELEMENT_SPEAKER_INDEX`
 - *Note:*
 - * If the input file is an 8 channel PCM file, output to all 8 channels is available. The properties of the PCM file are set in the `app_streamer.c` file using file source properties sent to the streamer:
 - `PROP_FILESRC_SET_SAMPLE_RATE` - default value is 96000 [Hz]
 - `PROP_FILESRC_SET_NUM_CHANNELS` - default value is 8
 - `PROP_FILESRC_SET_BIT_WIDTH` - default value is 32

Playback itself can be started with the `STREAMER_Start` function.

Each of the elements has several properties that can be accessed using the `streamer_get_property` or `streamer_set_property` function. These properties allow a user to change the values of the appropriate elements. The list of properties can be found in `streamer_element_properties.h`. See the example of setting property value in the following piece of code from the `app_streamer.c` file:

```
ELEMENT_PROPERTY_T prop;

EXT_PROCESS_DESC_T ssrc_proc = {SSRC_Proc_Init, SSRC_Proc_Execute, SSRC_Proc_Deinit,
↪ &get_app_data()->proc_args};

prop.prop = PROP_SRC_PROC_FUNCPTR;
prop.val = (uintptr_t)&ssrc_proc;

if (streamer_set_property(streamer, 0, prop, true) != 0)
{
    return -1;
}

prop.prop = PROP_AUDIOSINK_SET_VOLUME;
prop.val = volume;
streamer_set_property(streamer, 0, prop, true);
```

Some of the predefined values can be found in the `streamer_api.h`.

States The application can be in 3 different states:

- Idle

- Running
- Paused

In each state, each command can have a different behavior. For more information, see [Commands in detail](#) section.

Commands in detail The applicatin is controlled by commands from the shell interface and the available commands for the selected mode can be displayed using the `help` command. Commands are processed in the `cmd.c` file.

- [help, version](#)
- [file stop](#)
- [file pause](#)
- [file volume <volume>](#)
- [file seek <seek_time>](#)
- [file play <filename>](#)
- [file list](#)
- [file info](#)

Legend for diagrams:

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((State)):::state
B{Condition}:::condition
C[Error message]:::error
D[Process function]:::function
```

help, version

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((Idle)):::state --> D[Write help or version]:::function
B((Running)):::state --> D
C((Paused)):::state --> D
D-->E((No state change)):::state
```

file stop

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```



```
B((Idle)):::state --> B
C((Running)):::state --> E((Idle)):::state
D((Paused)):::state --> E
```

file pause

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
B((Idle)):::state --> B
C((Running)):::state --> E((Paused)):::state
D((Paused)):::state --> F((Running)):::state
```

file volume <volume>

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
B((Idle)):::state --> M[Error: Play a track first]:::error
C((Running)):::state --> G{Volume
parameter
empty?}:::condition
D((Paused)):::state --> G
G -- Yes --> H[Error: Enter volume parameter]:::error
G -- No --> I{Volume
in range?}:::condition
I -- No --> J[Error: invalid value]:::error
I -- Yes --> K[Set volume]:::function
J --> L((No state
change)):::state
K --> L
H--> L
```

file seek <seek_time> The seek argument is only supported in the Standard mode.

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
B((Idle)):::state --> E[Error: First select
an audio track to play]:::error
E-->B
C((Running)):::state --> F[Error: First
pause the track]:::error
F --> C
D((Paused)):::state --> G{Seek
parameter
empty?}:::condition
G --No --> H{AAC file?}:::condition
```

```

G --Yes --> I[Error: Enter
a seek time value]:::error
I-->N((Paused)):::state;
H --Yes -->J[Error: The AAC decoder
does not support
the seek command]:::error
J-->N
H --No -->K{Seek
parameter
positive?}:::condition
K --No -->L[Error: The seek
time must be
a positive value]:::error
L-->N
K --Yes -->M[Seek the file]:::function
M-->N

```

file play <filename>

flowchart TD

```

classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

C((Running)):::state --> Z[Error: First stop
current track]:::error
D((Paused)):::state --> Z
B((Idle)):::state --> E{SD Card
inserted?}:::condition
E -- No -->F[Error: Insert SD
card]:::error
E -- Yes -->G{File
name
empty?}:::condition
G -- Yes -->H[Error: Enter
file name]:::error
G -- No -->I{File exists?}:::condition
I -- No -->O[Error: File
doesn't exist]:::error
I -- Yes -->J{Supported
format?}:::condition
J -- Yes -->K[Play the track]:::function
J -- No -->L[Error: Unsupported
file]:::error
K -->M((Running)):::state
L --> W((No state
change)):::state
O --> W
H --> W
F --> W
Z --> W

```

file list

flowchart TD

```

classDef function fill:#69CA00

```

```
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

B((Idle)):::state --> G{SD Card
inserted?}:::condition
C((Running)):::state --> G
D((Paused)):::state --> G
G -- Yes --> H[List supported files]:::function
G -- No --> I[Error: Insert SD card]:::error
I --> J((No state
change)):::state
H --> J
```

file info

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

B((Idle)):::state --> E[Write file info]:::function
C((Running)):::state --> E
D((Paused)):::state --> E
E --> F((No state
change)):::state
```

Processing Time Typical streamer pipeline execution times and their individual elements for the EVKC-MIMXRT1060 development board are presented in the following tables. The time spent on output buffers is not included in the traversal measurements. However, file reading time is accounted for. In the case of the WAV codec, the audio file was accessed in every pipeline run. Therefore, during each run, the file was read from the SD card. However, for the MP3 codec, where data must be processed in complete MP3 frames, the file was not read in every run. Instead, it was read periodically only when the codec buffer did not contain a complete frame of data.

For further details, please refer to the [Processing Time](#) document.

WAV	streamer	file_src	codec	SSRC_proc	speaker
48kHz	1.1 ms	850 μ s	150 μ s	70 μ s	40 μ s
44kHz	1.75 ms	850 μ s	180 μ s	670 μ s	40 μ s

MP3	streamer	file_src	codec	SSRC_proc	speaker
48 kHz with file read	2.9 ms	2.3 μ s	450 μ s	60 μ s	50 μ s
48 kHz without file read	0.5 ms	x	400 μ s	40 μ s	40 μ s
44 kHz with file read	3.2 ms	2.3 μ s	440 μ s	400 μ s	50 μ s
44 kHz without file read	0.9 ms	x	440 μ s	390 μ s	40 μ s

Maestro record example

Table of content

- [Overview](#)
- [Hardware requirements](#)
- [Hardware modifications](#)
- [Preparation](#)
- [Running the demo](#)
- [Example configuration](#)
- [Functionality](#)
- [States](#)
- [Commands in detail](#)
- [Processing Time](#)

Overview The Maestro record example demonstrates audio processing on the ARM cortex core utilizing the Maestro Audio Framework library.

The application is controlled by commands from a shell interface using serial console.

Depending on target platform or development board there are different modes and features of the demo supported.

- **Loopback** - The application demonstrates a loopback from the microphone to the speaker without any audio processing. Mono, stereo or multichannel mode can be used, depending on the hardware, see [table](#) below.
- **File recording** - The application takes audio samples from the microphone inputs and stores them to an SD card as an PCM file. The PCM file has following parameters:
 - Mono and stereo : 2 channels, 16kHz, 16bit width
 - Multi-channel (AUD-EXP-42448): 6 channels, 16kHz, 32bit width
- **Voice control** - The application takes audio samples from the microphone input and uses the VIT library to recognize wake words and voice commands. If a wake word or a voice command is recognized, the application write it to the serial terminal.
- **Encoding** - The application takes PCM samples from memory and sends them to the Opus encoder. The encoded data is stored in memory and compared to a reference. The result of the comparison is finally written into the serial terminal.

As shown in the table below, the application is supported on several development boards, and each development board may have certain limitations, some development boards may also require hardware modifications or allow to use of an audio expansion board. Therefore, please check the supported features and [Hardware modifications](#) or [Example configuration](#) sections before running the demo.

Limitations:

- Note:
 - *LPCXPRESSO55S69* - MCUXpresso IDE project default debug console is semihost
- Addition libraries
 - **VIT:**
 - * The VIT is supported only in the MCUXpresso IDE and ARMGCC.
 - * *LPCXPRESSO55S69* - The VIT is disabled by default due to insufficient memory. To enable it, see the [Example configuration](#) section.

- * *EVK-MCXN5XX* - Some VIT models can't fit into memory. In order to free some space it is necessary to disable SD card handling and opus encoder. To disable it, see the [Example configuration](#) section.
- **VoiceSeeker:**
 - * The VoiceSeeker is supported only in the MCUXpresso IDE and ARMGCC.
- Encoder
 - **OPUS:**
 - * *LPCXpresso55s69* - The encoder is not supported due to insufficient memory.
- The File recording mode is not supported on *RW612BGA* development board due to missing SD card slot.

Known issues:

- *EVKB-MIMXRT1170* - After several tens of runs (the number of runs is not deterministic), the development board restarts because a power-up sequence is detected on the RESET pin (due to a voltage drop).

More information about supported features can be found on the [Supported features](#) page.

Hardware requirements

- Desired development board
- Micro USB cable
- Headphones with 3.5 mm stereo jack
- Personal computer
- Optional:
 - SD card for file output
 - Audio expansion board [AUD-EXP-42448 \(REV B\)](#)
- *LPCXpresso55s69*:
 - Source of sound with 3.5 mm stereo jack connector

Hardware modifications Some development boards need some hardware modifications to run the application. If the development board is not listed here, its default setting is required.

- *EVKB-MIMXRT1170*:
 1. Please remove below resistors if on board wifi chip is not DNP:
 - R228, R229, R232, R234
 2. Please make sure R136 is weld for GPIO card detect.
- *EVK-MCXN5XX*:
 - Short: JP7 2-3, JP8 2-3, JP10 2-3, JP11 2-3
- *RW612BGA*:
 - Connect: JP50; Disconnect JP9, JP11

Preparation

1. Connect a micro USB cable between the PC host and the debug USB port on the development board
2. Open a serial terminal with the following settings:
 - 115200 baud rate
 - 8 data bits
 - No parity
 - One stop bit
 - No flow control
3. Download the program to the target board.
4. Insert the headphones into the Line-Out connector (headphone jack) on the development board.
5. *LPCXpresso55s69*:
 - Insert source of sound to audio Line-In connector (headphone jack) on the development board.
6. Either press the reset button on your development board or launch the debugger in your IDE to begin running the demo.

Running the demo When the example runs successfully, you should see similar output on the serial terminal as below:

```
*****
Maestro audio record demo start
*****

Copyright 2022 NXP
[APP_SDCARD_Task] start
[APP_Shell_Task] start

>> [APP_SDCARD_Task] SD card drive mounted
```

Type help to see the command list. Similar description will be displayed on serial console:

```
>> help

"help": List all the registered commands

"exit": Exit program

"version": Display component versions

"record_mic": Record MIC audio and perform one (or more) of following actions:
- playback on codec
- perform VoiceSeeker processing
- perform voice recognition (VIT)
- store samples to a file.

USAGE: record_mic [audio|file|<file_name>|vit] 20 [<language>]
The number defines length of recording in seconds.

Please see the project defined symbols for the languages supported.
Then specify one of: en/cn/de/es/fr/it/ja/ko/pt/tr as the language parameter.
For voice recognition say supported WakeWord and in 3s frame supported command.
```

(continues on next page)

(continued from previous page)

Please note that this VIT demo is near-field and uses 1 on-board microphone.

NOTES: This command returns to shell after the recording is finished.

To store samples to a file, the "file" option can be used to create a file with a predefined name, or any file name (without whitespaces) can be specified instead of the "file" option.

"opus_encode": Initializes the streamer with the Opus memory-to-memory pipeline and encodes a hardcoded buffer.

Details of commands can be found [here](#).

Example configuration The example can be configured by user. There are several options how to configure the example settings, depending on the environment. For configuration using west and Kconfig, please follow the instructions [here](#). Before configuration, please check the [table](#) to see if the feature is supported on the development board.

- **Connect AUD-EXP-42448:**

- *EVKC-MIMXRT1060:*

1. Disconnect the power supply for safety reasons.
2. Insert AUD-EXP-42448 into J19 to be able to use the CS42448 codec for multichannel output.
3. Uninstall J99.
4. Set the DEMO_CODEC_WM8962 macro to 0 in the app_definitions.h file
5. Set the DEMO_CODEC_CS42448 macro to 1 in the app_definitions.h file.
6. Enable VoiceSeeker, see point bellow.

- *Note:*

- * The audio stream is as follows:

- Stereo INPUT 1 (J12) -> LINE 1&2 OUTPUT (J6)
 - Stereo INPUT 2 (J15) -> LINE 3&4 OUTPUT (J7)
 - MIC1 & MIC2 (P1, P2) -> LINE 5&6 OUTPUT (J8)
 - Insert the headphones into the different line outputs to hear the inputs.
 - To use the Stereo INPUT 1, 2, connect an audio source LINE IN jack.

- **Enable VoiceSeeker:**

- On some development boards the VoiceSeeker is enabled by default, see the [table](#) above.

- If more than one channel is used and VIT is enabled, the VoiceSeeker that combines multiple channels into one must be used, as VIT can only work with mono signal.

- Using MCUXPresso IDE:

- * It is necessary to add VOICE_SEEKER_PROC symbol to preprocessor defines on project level:

- (Project -> Properties -> C/C++ Build -> Settings -> MCU C Compiler -> Preprocessor)

- Using Kconfig:

- * Enable the VoiceSeeker in the guiconfig using MCUX_PRJSEG_middleware.audio_voice.components.voice_seeker

- **Enable VIT:**

- *LPCXpresso55s69 and MCX-N5XX:*

- * In MCUXPresso IDE (SDK package):

- 1. Remove SD_ENABLED and STREAMER_ENABLE_FILE_SINK symbols from preprocessor defines on project level.
 2. Add VIT_PROC symbol to preprocessor defines on project level:
 - (Project -> Properties -> C/C++ Build -> Settings -> MCU C Compiler -> Preprocessor)

- * In armgcc in SDK package:

- 1. Remove SD_ENABLED and STREAMER_ENABLE_FILE_SINK symbols from preprocessor defines in flags.cmake file.
 2. Remove OPUS_ENCODE=1 and STREAMER_ENABLE_ENCODER preprocessor defines in flags.cmake file.
 3. Add VIT_PROC symbol to preprocessor defines in flags.cmake file.
 4. Remove sdmmc_config.c,h files from CMakeLists.txt file.

- * In Kconfig:

- 1. Disable File sink MCUX_COMPONENT_middleware.audio_voice.maestro.element.file_sink.enable
 2. Make sure SD card support is disabled MCUX_COMPONENT_middleware.sdmmc.sd and MCUX_COMPONENT_middleware.sdmmc.host.usdhc
 3. Make sure sdmmc_config files (.c, .h) is excluded from project build
 - remove mcux_add_source function that adds the sources in reconfig.cmake in maestro_record/cm33_core0 folder
 4. Disable fatfs MCUX_COMPONENT_middleware.fatfs and MCUX_COMPONENT_middleware.fatfs.sd
 5. Disable file utils MCUX_COMPONENT_middleware.audio_voice.maestro.file_utils.enable
 6. Make sure Opus encoder is disabled MCUX_COMPONENT_middleware.audio_voice.maestro.element.encoder.opus.enable
 7. Make sure VIT_PROC symbol is defined
 - remove mcux_remove_macro function that removes the VIT_PROC preprocessor definition in reconfig.cmake in maestro_record folder
 8. Make sure VIT processing is enabled MCUX_PRJSEG_middleware.audio_voice.components.vit

- **VIT model generation:**

- For custom VIT model generation (defining own wake words and voice commands) please use <https://vit.nxp.com/>

- **Disable SD card handling:**

- In MCUXPresso IDE:

- * Remove SD_ENABLED and STREAMER_ENABLE_FILE_SINK symbols from preprocessor defines on project level:

- (Project -> Properties -> C/C++ Build -> Settings -> MCU C Compiler -> Preprocessor)

- In armgcc in SDK package:

- * Remove `SD_ENABLED` and `STREAMER_ENABLE_FILE_SINK` symbols from preprocessor defines in `flags.cmake` file.
- In Kconfig:
 1. Disable File sink `MCUX_COMPONENT_middleware.audio_voice.maestro.element.file_sink.enable`
 2. Make sure SD card support is disabled `MCUX_COMPONENT_middleware.sdmmc.sd`

Functionality The `record_mic` or `opus_encode` command calls the `STREAMER_mic_Create` or `STREAMER_opusmem2mem_Create` function from the `app_streamer.c` file depending on the selected mode.

- When the *Loopback* mode is selected, the command calls the `STREAMER_mic_Create` function that creates a pipeline with the following elements:
 - `ELEMENT_MICROPHONE_INDEX`
 - `ELEMENT_SPEAKER_INDEX`
- When the *File recording* mode is selected, the command calls the `STREAMER_mic_Create` function that creates a pipeline with the following elements: - `ELEMENT_MICROPHONE_INDEX` - `ELEMENT_FILE_SINK_INDEX`
- When the *Voice control* mode is selected, the command calls the `STREAMER_mic_Create` function that creates a pipeline with the following elements: - `ELEMENT_MICROPHONE_INDEX` - `ELEMENT_VOICSEEKER_INDEX` (If `VOICE_SEEKER_PROC` is defined) - `ELEMENT_VIT_INDEX`
- When the *Encoding* mode is selected, the command calls the `STREAMER_opusmem2mem_Create` function that creates a pipeline with the following elements: - `ELEMENT_MEM_SRC_INDEX` - `ELEMENT_ENCODER_INDEX` - `ELEMENT_MEM_SINK_INDEX`

Recording itself can be started with the `STREAMER_Start` function.

Each of the elements has several properties that can be accessed using the `streamer_get_property` or `streamer_set_property` function. These properties allow a user to change the values of the appropriate elements. The list of properties can be found in `streamer_element_properties.h`. See the example of setting property value in the following piece of code from the `app_streamer.c` file:

```
ELEMENT_PROPERTY_T prop;

prop.prop = PROP_MICROPHONE_SET_NUM_CHANNELS;
prop.val = DEMO_MIC_CHANNEL_NUM;
streamer_set_property(handle->streamer, 0, prop, true);

prop.prop = PROP_MICROPHONE_SET_BITS_PER_SAMPLE;
prop.val = DEMO_AUDIO_BIT_WIDTH;
streamer_set_property(handle->streamer, 0, prop, true);

prop.prop = PROP_MICROPHONE_SET_FRAME_MS;
prop.val = DEMO_MIC_FRAME_SIZE;
streamer_set_property(handle->streamer, 0, prop, true);

prop.prop = PROP_MICROPHONE_SET_SAMPLE_RATE;
prop.val = DEMO_AUDIO_SAMPLE_RATE;
streamer_set_property(handle->streamer, 0, prop, true);
```

Some of the predefined values can be found in the `streamer_api.h`.

States The application can be in 2 different states:

- Idle
- Running

Commands in detail

- *help, version*
- *record_mic audio <time>*
- *record_mic file <time>*
- *record_mic <file_name> <time>*
- *record_mic vit <time> <language>*
- *opus_encode*

Legend for diagrams:

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((State)):::state
B{Condition}:::condition
C[Error message]:::error
D[Process function]:::function
```

help, version

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((Idle)):::state --> C[Write help or version]:::function
B((Running)):::state --> C
C --> E((No state
change)):::state
```

record_mic audio <time>

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
B((Idle)):::state --> D{time
> 0 ?}:::condition
D -- Yes --> F[recording]:::function
D -- No --> E[Error: Record length
must be greater than 0]:::error
E --> B
F --> C((Running)):::state
```

```
C --> G{time
expired?}:::condition
G -- No --> C
G -- Yes --> B
```

record_mic file <time>/record_mic <file_name> <time>

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

B((Idle)):::state --> C{time
> 0 ?}:::condition
C -- Yes --> D{SD card
inserted?}:::condition
C -- No --> E[Error: Record length
must be greater than 0]:::error
E --> B
D -- Yes --> G{Custom
file name?}:::condition
G -- Yes --> H[Create custom
file name]:::function
G -- No --> I[Create default
file name]:::function
H --> J[Recording]:::function
I --> J
J --> K((Running)):::state
K --> L{time
expired?}:::condition
L -- No --> K
L -- Yes --> B
D -- No --> F[Error: Insert SD
card first]:::error
F --> B
```

record_mic vit <time> <language>

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

B((Idle)):::state --> C{time
> 0 ?}:::condition
C -- Yes --> E{Selected
language?}:::condition
C -- No --> D[Error: Record length
must be greater than 0]:::error
D --> B
E -- Yes --> G{Supported
language?}:::condition
E -- No --> F[Error: Language
```

```

not selected]:::error
F --> B
G -- Yes --> I[Recording with
voice recognition]:::function
G -- No --> H[Error: Language not supported]:::error
H --> B
I --> J((Running)):::state
J --> K{time
expired?}:::condition
K -- No --> J
K -- Yes --> B

```

opus_encode

flowchart TD

```

classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

```

```

B((Idle)):::state --> C[Encode file]:::function
C --> D[Check result]:::function
D --> B

```

Processing Time Typical execution times of the streamer pipeline for the EVKC-MIMXRT1060 development board are detailed in the following table. The duration spent on output buffers and reading from the microphone is excluded from traversal measurements. Three measured pipelines were considered. The first involves a loopback from microphone to speaker, supporting both mono and stereo configurations. The second pipeline is a mono voice control setup, comprising microphone and VIT blocks. The final pipeline is a stereo voice control setup, integrating microphone, voice seeker, and VIT blocks.

For further details of execution times on individual elements, please refer to the [Processing Time](#) document.

	streamer
microphone -> speaker 1 channel	40 µs
microphone -> speaker 2 channels	115 µs
microphone -> VIT	7.4 ms
microphone -> voice seeker -> VIT	9.9 ms

Maestro sync example

Table of content

- [Overview](#)
- [Hardware requirements](#)
- [Hardware modifications](#)
- [Preparation](#)
- [Running the demo](#)

- [Example configuration](#)
- [Functionality](#)
- [States](#)
- [Commands in detail](#)

Overview The Maestro sync example demonstrates the use of synchronous pipelines (Tx and Rx in this case) processing in a single streamer task on the ARM cortex core utilizing the Maestro Audio Framework library.

The application is controlled by commands from a shell interface using serial console.

The feature is useful for testing the latency of the pipeline or implementing algorithms requiring reference signals such as echo cancellation. The VoiceSeeker library available in this example is not featuring AEC (Acoustic Echo Cancellation), but NXP is offering it in the premium version of the library. More information about the premium version can be found at [VoiceSeeker](#). page. The demo uses two pipelines running synchronously in a single streamer task:

1. Playback (Tx) pipeline:

- Playback of audio data in PCM format stored in flash memory to the audio Line-Out connector (speaker).

2. Recording (Rx) pipeline:

- Record audio data using a microphone.
- VoiceSeeker processing.
- Wake words + voice commands recognition.
- Save the VoiceSeeker output to the voiceseecker_output.pcm file on the SD card.

As shown in the table below, the application is supported on several development boards, and each development board may have certain limitations, some development boards may also require hardware modifications or allow to use of an audio expansion board. Therefore, please check the supported features and [Hardware modifications](#) or [Example configuration](#) sections before running the demo.

Limitations:

- Addition labraries
 - **VIT:**
 - * The VIT is supported only in the MCUXpresso IDE.
 - **VoiceSeeker:**
 - * The VoiceSeeker is supported only in the MCUXpresso IDE.

Known issues:

- No known issues.

More information about supported features can be found on the [Supported features](#) page.

Hardware requirements

- Desired development board
- Micro USB cable
- Speaker with 3.5 mm stereo jack
- Personal computer
- Optional:

- SD card for file output

Hardware modifications Some development boards need some hardware modifications to run the application. If the development board is not listed here, its default setting is required.

- *EVKC-MIMXRT1060*:
 1. Please make sure resistors below are removed to be able to use SD-Card.
 - R368, R347, R349, R365, R363
 2. Please Make sure J99 is installed.

Preparation

1. Connect a micro USB cable between the PC host and the debug USB port on the development board
2. Open a serial terminal with the following settings:
 - 115200 baud rate
 - 8 data bits
 - No parity
 - One stop bit
 - No flow control
3. Download the program to the target board.
4. Insert the speaker into the Line-Out connector (headphone jack) on the development board.
5. *Optional*: Insert an SD card into the SD card slot to record to the VoiceSeeker output.
6. Either press the reset button on your development board or launch the debugger in your IDE to begin running the demo.

Running the demo When the example runs successfully, you should see similar output on the serial terminal as below:

```
*****
Maestro audio sync demo start
*****
```

```
Copyright 2022 NXP
[APP_SDCARD_Task] start
[APP_Shell_Task] start
```

```
>> [APP_SDCARD_Task] SD card drive mounted
```

Type help to see the command list. Similar description will be displayed on serial console:

```
>> help
```

```
"help": List all the registered commands
```

```
"exit": Exit program
```

```
"version": Display component versions
```

```
"start [nosdcard]": Starts a streamer task.
```

```
- Initializes the streamer with the Memory->Speaker pipeline and with
```

(continues on next page)

(continued from previous page)

the Microphone->VoiceSeeker->VIT->SDcard pipeline.
 - Runs repeatedly until stop command.
 nosdcard - Doesn't use SD card to store data.

"stop": Stops a running streamer:

"debug [on|off]": Starts / stops debugging.
 - Starts / stops saving VoiceSeeker input data (reference and microphone data) to SDRAM.
 - After the stop command, this data is transferred to the SD card.

Details of commands can be found [here](#).

Example configuration The example can be configured by user. Before configuration, please check the [table](#) to see if the feature is supported on the development board.

- **Enable the premium version of VoiceSeeker:**

- The premium version of the VoiceSeeker library with AEC is API compatible with this example.
- To get the premium version, please visit [VoiceSeeker](#) page.
- The following steps are required to run this example with the VoiceSeeker&AEC library.
 - * Link the voiceseeker.a library instead of voiceseeker_no_aec.a.
 - * Set the RDSP_ENABLE_AEC definition to 1U in the voiceseeker.h file

- **VIT model generation:**

- For custom VIT model generation (defining own wake words and voice commands) please use <https://vit.nxp.com/>

Functionality The start <nosdcard> command calls the STREAMER_Create function from the app_streamer.c file that creates pipelines with the following elements:

- Playback pipeline:
 - ELEMENT_MEM_SRC_INDEX
 - ELEMENT_SPEAKER_INDEX
- Record pipeline:
 - ELEMENT_MICROPHONE_INDEX
 - ELEMENT_VOICSEEKER_INDEX
 - ELEMENT_VIT_PROC_INDEX
 - ELEMENT_FILE_SINK_INDEX (If the nosdcard argument is not used)

Processing itself can be started with the STREAMER_Start function.

Each of the elements has several properties that can be accessed using the streamer_get_property or streamer_set_property function. These properties allow a user to change the values of the appropriate elements. The list of properties can be found in streamer_element_properties.h. See the example of setting property value in the following piece of code from the app_streamer.c file:

```
ELEMENT_PROPERTY_T prop;

MEMSRC_SET_BUFFER_T buf;
```

(continues on next page)

(continued from previous page)

```

buf.location = (int8_t *)TESTAUDIO_DATA;
buf.size     = TESTAUDIO_LEN;

prop.prop = PROP_MEMSRC_SET_BUFF;
prop.val  = (uintptr_t)&buf;
if (STREAM_OK != streamer_set_property(handle->streamer, 0, prop, true))
{
    return kStatus_Fail;
}

prop.prop = PROP_MEMSRC_SET_MEM_TYPE;
prop.val  = AUDIO_DATA;
if (STREAM_OK != streamer_set_property(handle->streamer, 0, prop, true))
{
    return kStatus_Fail;
}

prop.prop = PROP_MEMSRC_SET_SAMPLE_RATE;
prop.val  = DEMO_SAMPLE_RATE;
if (STREAM_OK != streamer_set_property(handle->streamer, 0, prop, true))
{
    return kStatus_Fail;
}

```

Some of the predefined values can be found in the `streamer_api.h`.

States The application can be in 2 different states:

- Idle
- Running

Commands in detail

- [help, version](#)
- [start \[nosdcard\]](#)
- [stop](#)
- [debug \[on|off\]](#)

Legend for diagrams:

flowchart TD

```

classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

```

```

A((State)):::state
B{Condition}:::condition
C[Error message]:::error
D[Process function]:::function

```

help, version

flowchart TD

```

classDef function fill:#69CA00
classDef condition fill:#0EAFE0

```



```
classDef state fill:#F9B500
classDef error fill:#F54D4D

A((Idle)):::state --> C[Write help or version]:::function
B((Running)):::state --> C
C --> E((No state
change)):::state
```

start [nosdcard]

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

A((Idle)):::state --> B{nosdcard
parameter?}:::condition
B -- Yes --> CH[Playing to Line-out and
recording]:::function
CH --> L((Running)):::state
B -- No --> C{Is SD card
inserted?}:::condition
C -- Yes --> E[Playing to Line-out and
recording to SD card]:::function
E --> F((Running)):::state
F --> G{Debugging
is enabled?}:::condition
G -- No --> F
G -- Yes --> H[Save reference and
microphone data to SDRAM]:::function
H --> F
C -- No --> D[Error: Insert SD
card first]:::error
D --> A
J((Running)):::state --> K[Error: The streamer task is
already running]:::error
K --> J
```

stop

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D

A((Idle)):::state --> A
B((Running)):::state --> C{Is debugging
enabled?}:::condition
C -- Yes --> E[Copy reference and
microphone data to
the SD card]:::function
E --> G((Idle)):::state
C -- No --> G
```

debug [on | off]

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((Idle)):::state --> B[Error: First, start
the streamer task]:::error
C((Running)):::state --> D{Any
parameter?}:::condition
D -- Yes --> F{Started with
nosdc card
parameter?}:::condition
F -- No --> H[Set debugging]:::function
H --> C
F -- Yes --> G[Error: Debugging cannot be used]:::error
G --> C
D -- No --> E[Error: Use the parameter
either on or off]:::error
E --> C
```

Maestro USB microphone example**Table of content**

- [Overview](#)
- [Hardware requirements](#)
- [Hardware modifications](#)
- [Preparation](#)
- [Running the demo](#)
- [Example configuration](#)
- [Functionality](#)
- [States](#)
- [Commands in detail](#)

Overview The Maestro USB microphone example demonstrates audio processing on the ARM cortex core utilizing the Maestro Audio Framework library.

The application is controlled by commands from a shell interface using serial console.

The development board will be enumerated as a USB audio class 2.0 device on the USB host. The application takes audio samples from the microphone inputs and sends them to the USB host via the USB bus. User will see the volume levels obtained from the USB host but this is only an example application. To leverage the volume values, the demo has to be modified.

As shown in the table below, the application is supported on several development boards, and each development board may have certain limitations, some development boards may also require hardware modifications or allow to use of an audio expansion board. Therefore, please check the supported features and [Hardware modifications](#) or [Example configuration](#) sections before running the demo.

Limitations:

- *Note:*

1. When connected to MacBook, change the PCM format from (0x02,0x00,) to (0x01,0x00,) in the `g_config_descriptor[CONFIG_DESC_SIZE]` in the `usb_descriptor.c` file. Otherwise, it can't be enumerated and noise is present when recording with the QuickTime player because the sampling frequency and bit resolution do not match.
2. When device functionality is changed, please uninstall the previous PC driver to make sure the device with changed functionality can run normally.
3. If you're having audio problems on Windows 10 for recorder, please disable signal enhancement as the following if it is enabled and have a try again.

Known issues:

- No known issues.

More information about supported features can be found on the [Supported features](#) page.

Hardware requirements

- Desired development board
- 2x Micro USB cable
- Personal Computer
- *LPCXpresso55s69:*
 - Source of sound with 3.5 mm stereo jack connector

Hardware modifications Some development boards need some hardware modifications to run the application. If the development board is not listed here, its default setting is required.

Preparation

1. Connect the first micro USB cable between the PC host and the debug USB port on the development board
2. Open a serial terminal with the following settings:
 - 115200 baud rate
 - 8 data bits
 - No parity
 - One stop bit
 - No flow control
3. Download the program to the target board.
4. *LPCXpresso55s69:*
 - Insert source of sound to Audio Line-In connector (headphone jack) on the development board.
5. Connect the second micro USB cable between the PC host and the USB port on the development board.
6. Either press the reset button on your development board or launch the debugger in your IDE to begin running the demo.

Running the demo When the example runs successfully, you should see similar output on the serial terminal as below:

```
*****
Maestro audio USB microphone solutions demo start
*****

Copyright 2022 NXP
[APP_Shell_Task] start

>> usb_mic -1

Starting maestro usb microphone application
The application will run until the board restarts
[STREAMER] Message Task started
Starting recording
[STREAMER] start usb microphone
Set Cur Volume : 1f00
```

Type help to see the command list. Similar description will be displayed on serial console:

```
>> help

"help": List all the registered commands

"exit": Exit program

"version": Display component versions

"usb_mic": Record MIC audio and playback to the USB port as an audio 2.0
microphone device.

USAGE: usb_mic <seconds>
<seconds> Time in seconds how long the application should run.
When you enter a negative number the application will
run until the board restarts.
EXAMPLE: The application will run for 20 seconds: usb_mic 20
```

Details of commands can be found [here](#).

Example configuration The example only supports one mode and do not support any additional libraries, so the example can't be configured by user.

Functionality The `usb_mic` command calls the `STREAMER_mic_Create` function from the `app_streamer.c` file that creates pipeline with the following elements: - ELEMENT_MICROPHONE_INDEX - ELEMENT_USB_SINK_INDEX

Recording itself can be started with the `STREAMER_Start` function.

Each of the elements has several properties that can be accessed using the `streamer_get_property` or `streamer_set_property` function. These properties allow a user to change the values of the appropriate elements. The list of properties can be found in `streamer_element_properties.h`. See the example of setting property value in the following piece of code from the `app_streamer.c` file:

```
ELEMENT_PROPERTY_T prop;

prop.prop = PROP_MICROPHONE_SET_SAMPLE_RATE;
prop.val = AUDIO_SAMPLING_RATE;

streamer_set_property(handle->streamer, 0, prop, true);
```

(continues on next page)

(continued from previous page)

```
prop.prop = PROP_MICROPHONE_SET_NUM_CHANNELS;
prop.val = 1;

streamer_set_property(handle->streamer, 0, prop, true);

prop.prop = PROP_MICROPHONE_SET_FRAME_MS;
prop.val = 1;

streamer_set_property(handle->streamer, 0, prop, true);
```

Some of the predefined values can be found in the `streamer_api.h`.

States The application can be in 2 different states:

- Idle
- Running

Commands in detail

- [help, version](#)
- [usb_mic <seconds>](#)

Legend for diagrams:

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((State)):::state
B{Condition}:::condition
C[Error message]:::error
D[Process function]:::function
```

help, version

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((Idle)):::state --> C[Write help or version]:::function
B((Running)):::state --> C
C --> E((No state change)):::state
```

usb_mic <seconds>

flowchart TD

```
classDef function fill:#c6d22c
classDef condition fill:#7cb2de
classDef state fill:#fcb415
classDef error fill:#ff9900
```

```
B((Idle)):::state --> C{seconds
== 0?}:::condition
C -- No --> E{seconds
< 0?}:::condition
C -- Yes --> D[Error: Incorrect
command parameter]:::error
D --> B
E -- Yes --> G[recording]:::function
G --> H((Running)):::state
H --> H
E -- No --> F[recording]:::function
F --> I((Running)):::state
I --> J{seconds
expired?}:::condition
J -- No --> I
J -- Yes --> B
```

Maestro USB speaker example

Table of content

- [Overview](#)
- [Hardware requirements](#)
- [Hardware modifications](#)
- [Preparation](#)
- [Running the demo](#)
- [Example configuration](#)
- [Functionality](#)
- [States](#)
- [Commands in detail](#)

Overview The Maestro USB speaker example demonstrates audio processing on the ARM cortex core utilizing the Maestro Audio Framework library.

The application is controlled by commands from a shell interface using serial console.

The development board will be enumerated as a USB audio class 2.0 device on the USB host. The application takes audio samples from the USB host and sends them to the audio Line-Out port. User will see the volume levels obtained from the USB host but this is only an example application. To leverage the volume values, the demo has to be modified.

Depending on target platform or development board there are different modes and features of the demo supported.

- **Standard** - The mode demonstrates playback with up to 2 channels, up to 48 kHz sample rate and up to 16 bit width. This mode is enabled by default.
- **Multi-Channel** - In this mode the device is enumerated as a UAC 5.1. This mode is disabled by default. See the [Example configuration](#) section to see how to enable the mode.
 - When playing an 5.1 audio file, the example sends only the front-left and front-right channels to the audio Line-Out port (the other channels are ignored), since this example only supports on-board codecs with stereo audio output.

As shown in the table below, the application is supported on several development boards, and each development board may have certain limitations, some development boards may also require hardware modifications or allow to use of an audio expansion board. Therefore, please check the supported features and [Hardware modifications](#) or [Example configuration](#) sections before running the demo.

Limitations:

- *Note:*
 - If the USB device audio speaker example uses an ISO IN feedback endpoint, please attach the device to a host like PC which supports feedback function. Otherwise, there might be attachment issue or other problems.

Known issues:

- No known issues.

More information about supported features can be found on the [Supported features](#) page.

Hardware requirements

- Desired development board
- 2x Micro USB cable
- Personal Computer
- Headphones with 3.5 mm stereo jack

Hardware modifications Some development boards need some hardware modifications to run the application. If the development board is not listed here, its default setting is required.

Preparation

1. Connect the first micro USB cable between the PC host and the debug USB port on the development board
2. Open a serial terminal with the following settings:
 - 115200 baud rate
 - 8 data bits
 - No parity
 - One stop bit
 - No flow control
3. Download the program to the target board.
4. Connect the second micro USB cable between the PC host and the USB port on the development board.
5. Insert the headphones into Line-Out connector (headphone jack) on the development board.
6. Either press the reset button on your development board or launch the debugger in your IDE to begin running the demo.

Running the demo When the example runs successfully, you should see similar output on the serial terminal as below:

```
*****
Maestro audio USB speaker solutions demo start
*****

Copyright 2022 NXP
[APP_Shell_Task] start

>> usb_speaker -1

Starting maestro usb speaker application
The application will run until the board restarts
[STREAMER] Message Task started
Starting playing
[STREAMER] start usb speaker
Set Cur Volume : fbd5
```

Type help to see the command list. Similar description will be displayed on serial console:

```
>> help

"help": List all the registered commands

"exit": Exit program

"version": Display component versions

"usb_speaker": Play data from the USB port as an audio 2.0
speaker device.

USAGE:    usb_speaker <seconds>
<seconds> Time in seconds how long the application should run.
           When you enter a negative number the application will
           run until the board restarts.
EXAMPLE:  The application will run for 20 seconds: usb_speaker 20
```

Details of commands can be found [here](#).

Example configuration The example can be configured by user. Before configuration, please check the [table](#) to see if the feature is supported on the development board.

- **Enable Multi-channel mode:**

- The feature can be enabled by set the USB_AUDIO_CHANNEL5_1 macro to 1U in the usb_device_descriptor.h file.
- *Note:* When device functionality is changed, such as UAC 5.1, please uninstall the previous PC driver to make sure the device with changed functionality can run normally.

Functionality The `Usb_speaker` command calls the `STREAMER_speaker_Create` function from the `app_streamer.c` file that creates pipeline with the following elements: - ELEMENT_USB_SRC_INDEX - ELEMENT_SPEAKER_INDEX

Playback itself can be started with the `STREAMER_Start` function.

Each of the elements has several properties that can be accessed using the `streamer_get_property` or `streamer_set_property` function. These properties allow a user to change the values of the appropriate elements. The list of properties can be found in `streamer_element_properties.h`. See the example of setting property value in the following piece of code from the `app_streamer.c` file:


```
ELEMENT_PROPERTY_T prop;

prop.prop = PROP_USB_SRC_SET_SAMPLE_RATE;
prop.val = AUDIO_SAMPLING_RATE;

streamer_set_property(handle->streamer, 0, prop, true);

prop.prop = PROP_USB_SRC_SET_NUM_CHANNELS;
prop.val = 2;

streamer_set_property(handle->streamer, 0, prop, true);

prop.prop = PROP_USB_SRC_SET_FRAME_MS;
prop.val = 1;

streamer_set_property(handle->streamer, 0, prop, true);
```

Some of the predefined values can be found in the `streamer_api.h`.

States The application can be in 2 different states:

- Idle
- Running

Commands in detail

- [help, version](#)
- [usb_speaker <seconds>](#)

Legend for diagrams:

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((State)):::state
B{Condition}:::condition
C[Error message]:::error
D[Process function]:::function
```

help, version

flowchart TD

```
classDef function fill:#69CA00
classDef condition fill:#0EAFE0
classDef state fill:#F9B500
classDef error fill:#F54D4D
```

```
A((Idle)):::state --> C[Write help or version]:::function
B((Running)):::state --> C
C --> E((No state change)):::state
```

usb_speaker <seconds>

flowchart TD

```

classDef function fill:#c6d22c
classDef condition fill:#7cb2de
classDef state fill:#fcb415
classDef error fill:#FF999C

B((Idle)):::state --> C{Duration
== 0?}:::condition
C -- No --> E{Duration
< 0?}:::condition
C -- Yes --> D[Error: Incorrect
command parameter]:::error
D --> B
E -- Yes --> G[playing]:::function
G --> H((Running)):::state
H --> H
E -- No --> F[playing]:::function
F --> I((Running)):::state
I --> J{Duration
expired?}:::condition
J -- No --> I
J -- Yes --> B

```

Supported features The current version of the audio framework supports several optional features. These can be limited to some MCU cores or development boards variants. More information about support can be found on the specific example page:

- [maestro_playback](#)
- [maestro_record](#)
- [maestro_usb_mic](#)
- [maestro_usb_speaker](#)
- [maestro_sync](#)

Some features are delivered as prebuilt library and the binaries can be found in the `\middleware\audio_voice\components\*component*\libs` folder. The source code of some features can be found in the `\middleware\audio_voice\maestro\src` folder.

Decoders Supported decoders and its options are:

Decoder	Sample rates [kHz]	Number of channels	Bit depth
AAC	8, 11.025, 12, 16, 22.05, 24, 32, 44.1, 48	1, 2 (mono/stereo)	16
FLAC	8, 11.025, 12, 16, 22.05, 32, 44.1, 48	1, 2 (mono/stereo)	16
MP3	8, 11.025, 12, 16, 22.05, 24, 32, 44.1, 48	1, 2 (mono/stereo)	16
OPUS	8, 16, 24, 48	1, 2 (mono/stereo)	16
WAV	8, 11.025, 16, 22.05, 32, 44.1, 48	1, 2 (mono/stereo)	8, 16, 24

For more details about the reference decoders please see audio-voice-components repository documentation `\middleware\audio_voice\components\`.

Encoders

- **OPUS encoder** - The current version of the audio framework only supports a OPUS encoder. For more details about the encoder please see the following [link](#).

Sample rate converters

- **SSRC** - Synchronous sample rate converter. More details about SSRC are available in the User Guide, which is located in `middleware\audio_voice\components\ssrc\doc\`.
- **ASRC** - Asynchronous sample rate converter is not used in our examples, but it is part of the maestro middleware and can be enabled. To enable ASRC, the `maestro_framework_asrc` and `CMSIS_DSP_Library_Source` components must be added to the project. Furthermore, it is necessary to switch from Redlib to Newlib (semihost) library and add a platform definition to the project (e.g. for RT1170: `PLATFORM_RT1170_CORTEXM7`). Supported platforms can be found in the `PL_platformTypes.h` file. More details about ASRC are available in the User Guide, which is located in `middleware\audio_voice\components\asrc\doc\`.

Additional libraries

- **VIT** - Voice Intelligent Technology (VIT) Wake Word and Voice Command Engines provide free, ready to use voice UI enablement for developers. It enables customer-defined wake words and commands using free online tools. More details about VIT are available in the VIT package, which is located in `middleware\audio_voice\components\vit\{platform}\Doc\` (depending on the platform) or via following [link](#).
- **VoiceSeeker** - VoiceSeeker is a multi-microphone voice control audio front-end signal processing solution. More details about VoiceSeeker are available in the VoiceSeeker package, which is located in `middleware\audio_voice\components\voice_seeker\{platform}\Doc\` (depending on the platform) or via following [link](#).

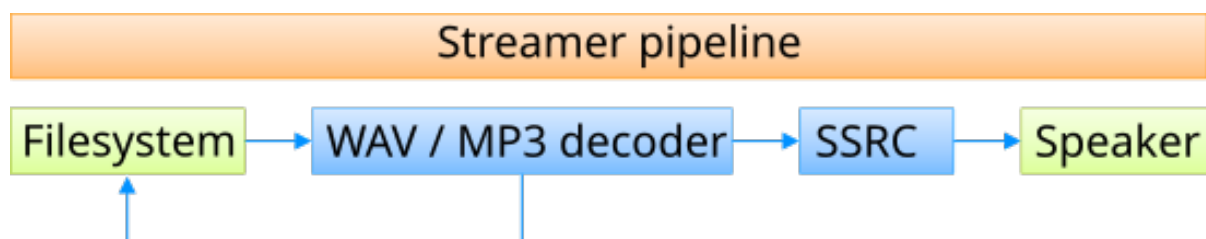
Processing Time

Table of content

- [Maestro playback example](#)
- [Maestro record example](#)

The individual time measurements were conducted using a logic analyzer by monitoring changes in the GPIO port levels on the EVKC-MIMXRT1060 development board. These measurements were executed for each individual pipeline run, capturing the timing at each corresponding element, and, when relevant, the interconnections between these elements.

Maestro playback example For the Maestro playback example the following reference audio file was used: `test_48khz_16bit_2ch.wav`. In this example, the pipeline depicted in the diagram was considered. Media codecs WAV and MP3 were taken into account. To compare the times spent on the SSRC block, sampling rates for both codecs were selected: 44.1 kHz and 48 kHz.



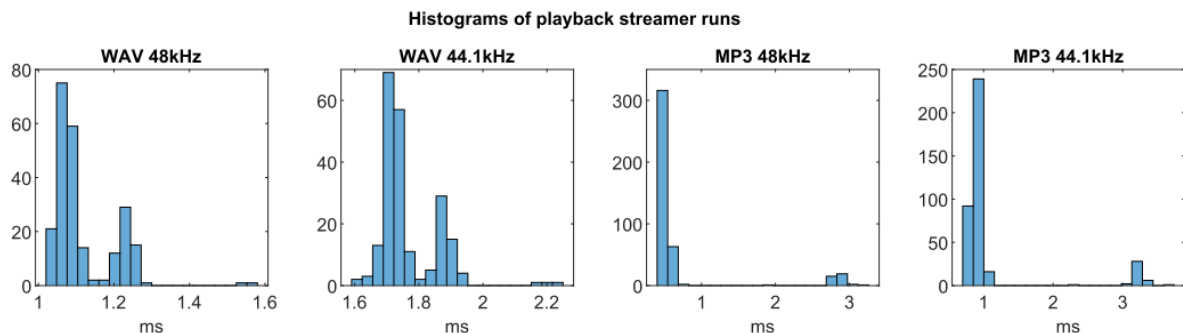
The measurement of streamer pipeline run started at the beginning of `streamer_process_pipelines()`: `streamer.c` and ended in the function `streamer_pcm_write()`: `streamer_pcm.c` just before the output buffer.

In the scenario involving the WAV codec, the audio file was accessed in every iteration of the streamer pipeline. Meaning, during each run, the file was read directly from the SD card. However, in the case of the MP3 codec, where data processing necessitates complete MP3 frames, the file wasn't read during every run. Rather, it was accessed periodically, triggered when the codec buffer lacked a complete MP3 frame of data. The total time spent on codec processing varies significantly depending on the type and implementation of the codec. For certain types of codecs, like FLAC, there may be multiple file accesses during a single pipeline run. The provided values are specific to the reference implementation. For details about the codecs please see `audio-voice-components` documentation `middleware\audio_voice\components\`.

The duration of the streamer pipeline illustrates that with a sampling frequency of 48 kHz, there is no resampling occurring at the SSRC element. Consequently, the overall pipeline time is lower than in the case of 44.1 kHz audio, where resampling takes place.

To enhance comprehension of the system's behavior, histograms of the pipeline run times and its elements are included. The greater time variance with the MP3 codec is precisely due to the absence of file reads in every run. In clusters with shorter times, there are no file accesses, while in clusters with longer times, file reads occur. This indicates that the majority of runs do not involve file access.

	WAV 48 kHz	WAV 44 kHz	MP3 48 kHz file read	MP3 48 kHz w/o file read	MP3 44 kHz file read	MP3 44 kHz w/o file read
mean	1.11 ms	1.76 ms	2.87 ms	0.51 ms	3.22 ms	0.89 ms
min	1.03 ms	1.60 ms	2.74 ms	0.41 ms	2.33 ms	0.74 ms
max	1.29 ms	2.23 ms	3.24 ms	1.83 ms	3.73 ms	1.12 ms



Time on each element In the tables and histograms below, the timings for individual elements and their connections are provided. Given that the file reading function was invoked during the codec's operation, the tables for individual elements display the total time on the codec element, the time on the codec element before the file read, and the time on the codec element after the file read. The individual blocks in the tables are as follows:

- **streamer** - total time of one pipeline run without time on output buffers
- **codec start** - time on decoder before file read
- **codec end** - time on decoder after file read
- **codec total** - `codec_start+codec_end`
- **file_src** - file reading time
- **SSRC_proc** - time on SSRC element
- **audio_sink** - time on audio sink without output buffers

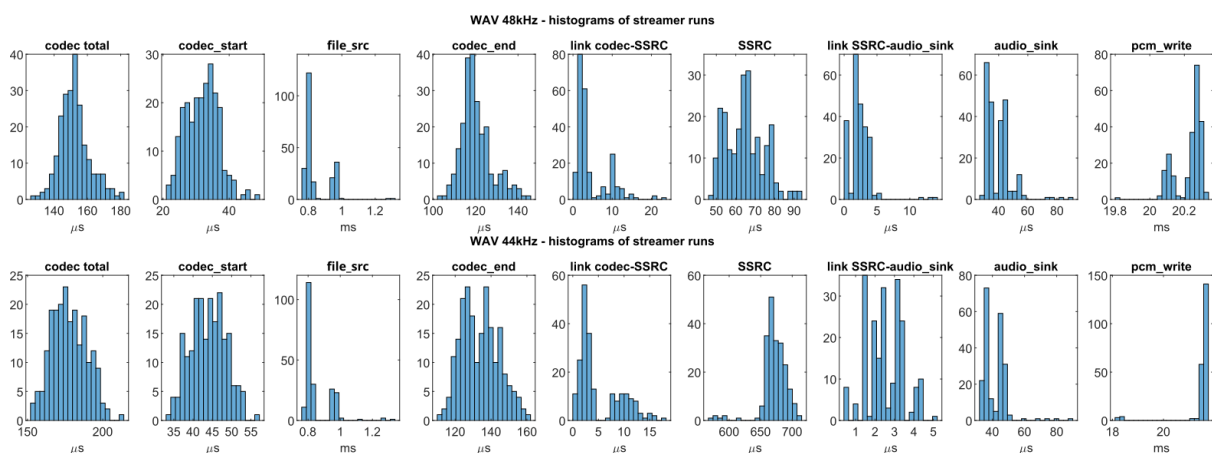
- **pcm_write** - time on output buffers
- **link** - time on element links

The start times of the time intervals for individual blocks and their respective links were measured by altering the GPIO pin level in the following functions:

- **streamer** - streamer_process_pipelines():streamer.c
- **codec** - decoder_sink_pad_process_handler():decoder_pads.c
- **file_src** - filesrc_read():file_src_rtos.c
- **SSRC_proc** - SSRC_Proc_Execute():ssrc_proc.c
- **audio_sink** - audiosink_sink_pad_chain_handler():audio_sink.c
- **pcm_write** - streamer_pcm_write():streamer_pcm.c
- **link** - pad_push():pad.c

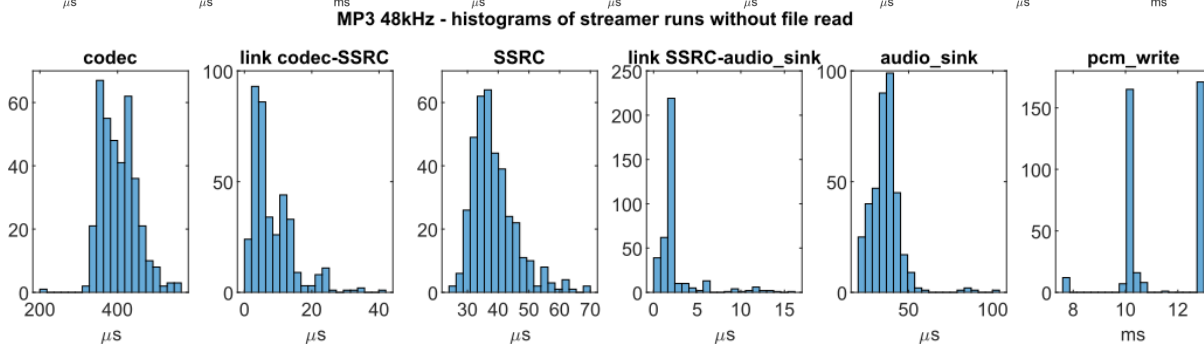
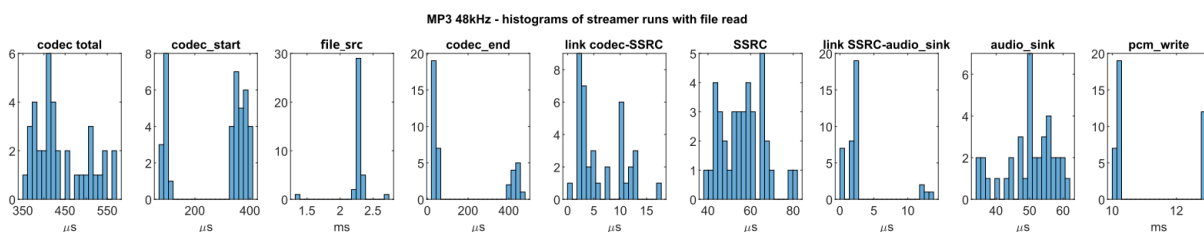
WAV 48kHz	stream	codec total	codec start	file_sr	codec end	link codec- SSRC	SSRC_p	link audio_sink	SSRC- audio_sink	au- dio_sin	pcm_write
mean	1.119 ms	152 µs	31 µs	0.843 ms	120 µs	5 µs	64 µs	2 µs		40 µs	20.228 ms
min	1.026 ms	125 µs	21 µs	0.773 ms	104 µs	<1 µs	47 µs	<1 µs		30 µs	19.805 ms
max	1.290 ms	193 µs	49 µs	1.311 ms	144 µs	23 µs	93 µs	14 µs		91 µs	20.324 ms

WAV 44kHz	stream	codec total	codec start	file_sr	codec end	link codec- SSRC	SSRC_p	link audio_sink	SSRC- audio_sink	au- dio_sin	pcm_write
mean	1.765 ms	178 µs	44 µs	0.853 ms	134 µs	5 µs	671 µs	3 µs		42 µs	21.472 ms
min	1.604 ms	145 µs	33 µs	0.770 ms	112 µs	<1 µs	574 µs	<1 µs		33 µs	18.163 ms
max	2.233 ms	218 µs	57 µs	1.335 ms	161 µs	18 µs	715 µs	5 µs		89 µs	21.746 ms



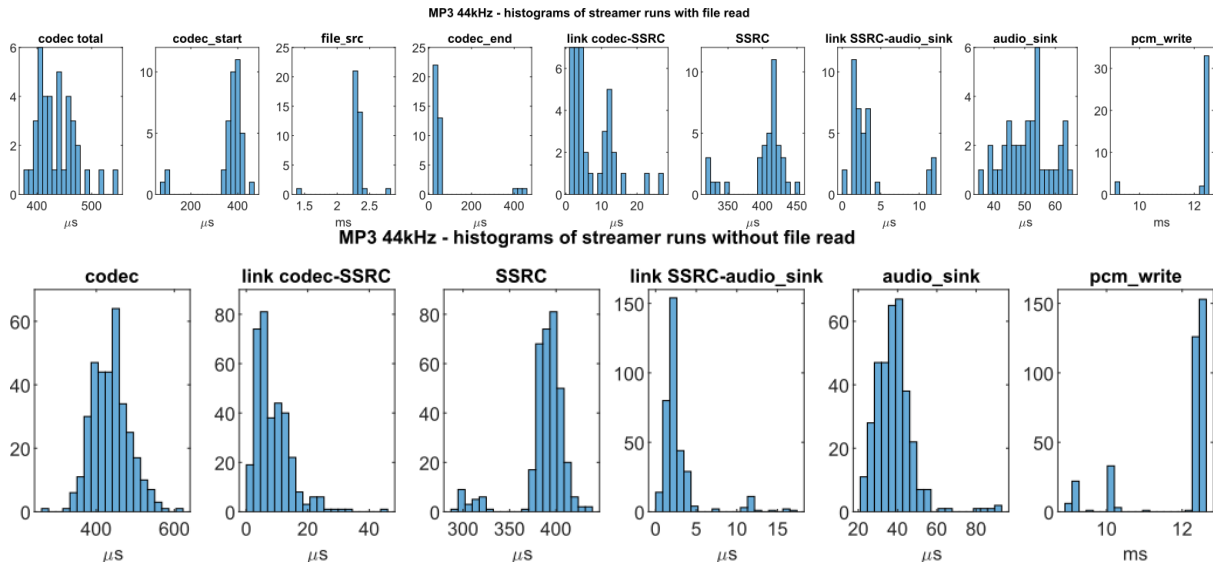
MP3 48 kHz w/ file read	stream	codec total	codec start	file_sr	codec end	link codec- SSRC	SSRC_	link SSRC- audio_sink	au- dio_sir	pcm_write
mean	2.871 ms	441 µs	279 µs	2.271 ms	162 µs	6 µs	56 µs	3 µs	50 µs	11.019 ms
min	2.739 ms	353 µs	74 µs	1.353 ms	26 µs	<1 µs	40 µs	<1 µs	34 µs	10.091 ms
max	3.244 ms	570 µs	409 µs	2.728 ms	467 µs	18 µs	80 µs	14 µs	62 µs	12.910 ms

MP3 48 kHz w/o file read	stream	codec total	codec start	file_s	codec end	link codec- SSRC	SSRC_	link SSRC- audio_sink	au- dio_sir	pcm_write
mean	0.508 ms	403 µs	x	x	x	8 µs	39 µs	3 µs	36 µs	11.326 ms
min	0.407 ms	208 µs	x	x	x	<1 µs	25 µs	<1 µs	21 µs	7.715 ms
max	1.834 ms	563 µs	x	x	x	41 µs	69 µs	16 µs	104 µs	12.941 ms

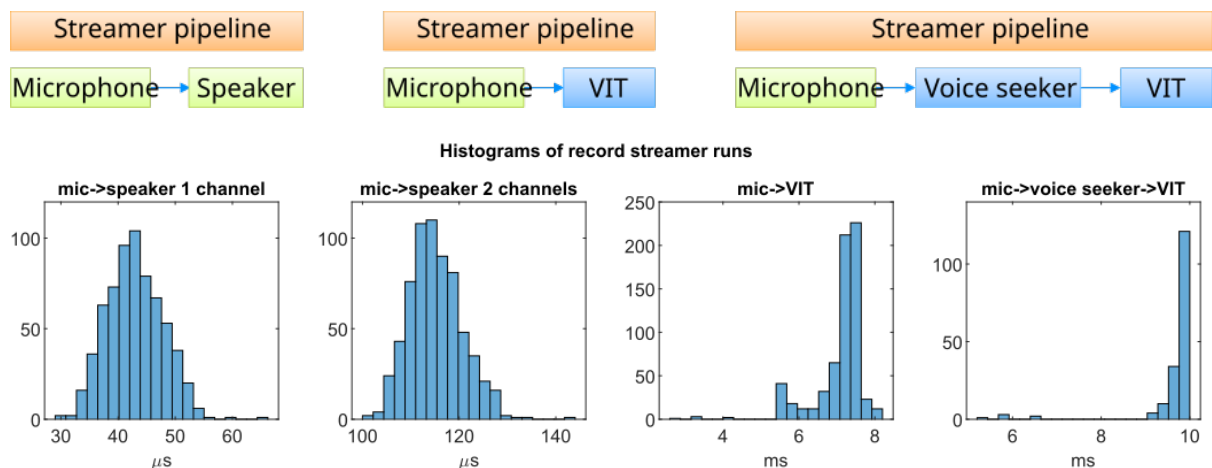


MP3 44 kHz w/ file read	stream	codec total	codec start	file_sr	codec end	link codec- SSRC	SSRC_	link SSRC- audio_sink	au- dio_sir	pcm_write
mean	3.217 ms	436 µs	367 µs	2.300 ms	66 µs	7 µs	403 µs	3 µs	51 µs	12.188 ms
min	2.329 ms	383 µs	73 µs	1.411 ms	26 µs	2 µs	318 µs	<1 µs	35 µs	9.119 ms
max	3.726 ms	547 µs	464 µs	2.801 ms	441 µs	27 µs	454 µs	12 µs	65 µs	12.529 ms

MP3 kHz w/o file read	44	streamer	codec total	codec start	file_s	codec end	link codec- SSRC	SSRC_	link SSRC- audio_sink	au- dio_sir	pcm_write
mean		0.891 ms	437 µs	x	x	x	9 µs	388 µs	3 µs	38 µs	11.934 ms
min		0.738 ms	268 µs	x	x	x	<1 µs	290 µs	<1 µs	22 µs	8.964 ms
max		1.115 ms	620 µs	x	x	x	45 µs	438 µs	17 µs	92 µs	12.624 ms



Maestro record example Typical execution times of the streamer pipeline and its individual elements for the EVKC-MIMXRT1060 development board are detailed in the following tables. The duration spent on output buffers and reading from the microphone is excluded from traversal measurements. Three measured pipelines are depicted in the figure below. The first involves a loopback from microphone to speaker, supporting both mono and stereo configurations. The second pipeline is a mono voice control setup, comprising microphone and VIT blocks. The final pipeline is a stereo voice control setup, integrating microphone, voice seeker, and VIT blocks. The measurement of streamer pipeline run started at the beginning of `streamer_process_pipelines():streamer.c` and ended in the function `streamer_pcm_write():streamer_pcm.c` just before the output buffer.



The individual blocks in the tables are as follows:

- **streamer** - total time of one pipeline run without time on output buffers and without time reading from the microphone
- **audio_src_start** - time on audio src before reading from the microphone
- **audio_src_end** - time on audio src after reading from the microphone
- **pcm_read** - reading from the microphone
- **voiceseeker** - time on voice seeker element
- **vit** - time on VIT element
- **audio_sink** - time on audio sink without output buffers
- **pcm_write** - time on output buffers
- **link** - time on element links

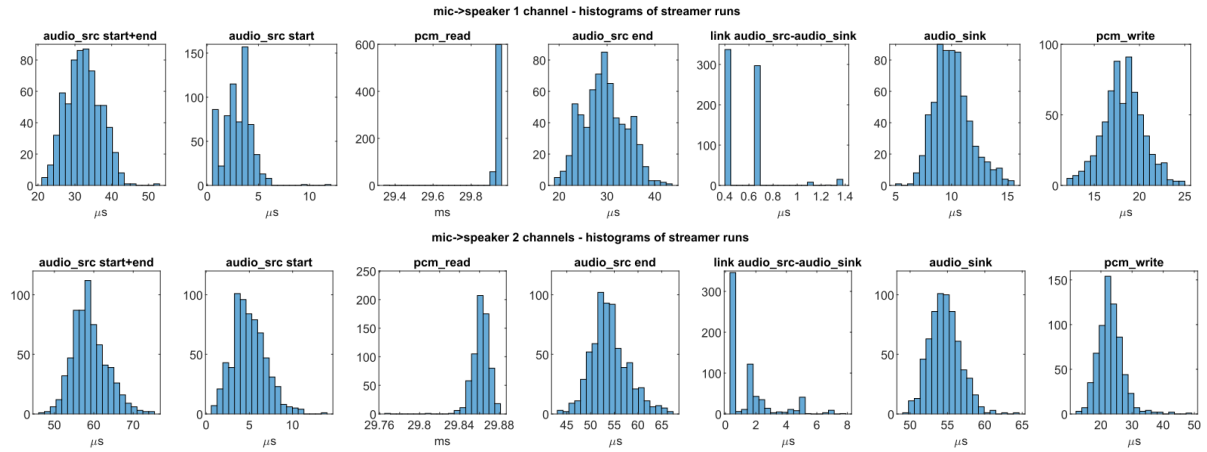
The start times of the time intervals for individual blocks and their respective links were measured by altering the GPIO pin level in the following functions:

- **streamer** - streamer_process_pipelines():streamer.c
- **audio_src** - audiosrc_src_process():audio_src.c
- **pcm_read** - streamer_pcm_read():streamer_pcm.c
- **voiceseeker** - audio_proc_sink_pad_chain_handler():audio_proc.c
- **vit** - vitsink_sink_pad_chain_handler():vit_sink.c
- **audio_sink** - audiosink_sink_pad_chain_handler():audio_sink.c
- **pcm_write** - streamer_pcm_write():streamer_pcm.c
- **link** - pad_push():pad.c

Pipeline Microphone -> Speaker

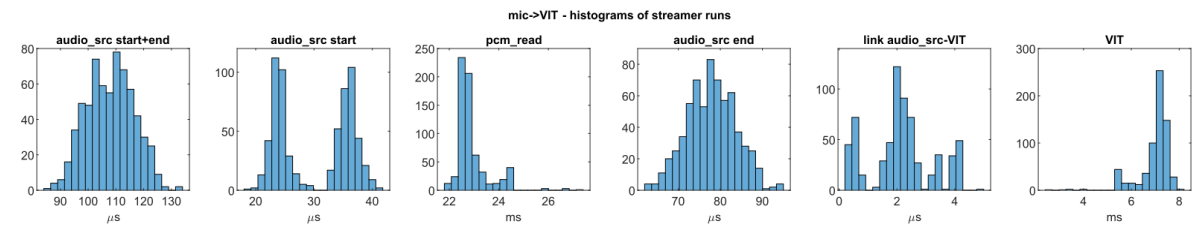
microphone speaker mono	->	stream	audio_src_start	pcm_read	audio_src_end	link audio_sink	audio_src- audio_sink	audio_sink	pcm_write
mean		43 µs	3 µs	29.938 ms	29 µs	<1 µs		10 µs	18 µs
min		26 µs	<1 µs	29.350 ms	19 µs	<1 µs		5 µs	12 µs
max		72 µs	12 µs	29.957 ms	44 µs	1 µs		15 µs	25 µs

microphone speaker stereo	->	stream	audio_src_start	pcm_read	audio_src_end	link audio_sink	audio_src- audio_sink	audio_sink	pcm_write
mean		115 µs	5 µs	29.861 ms	54 µs	2 µs		55 µs	23 µs
min		94 µs	<1 µs	29.768 ms	43 µs	<1 µs		50 µs	12 µs
max		154 µs	14 µs	29.880 ms	67 µs	8 µs		65 µs	49 µs



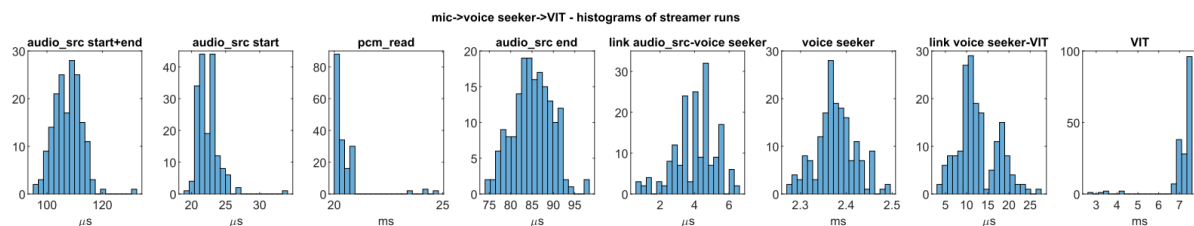
Pipeline Microphone -> VIT

microphone -> VIT	streamer	audio_src_start	pcm_read	audio_src_end	link audio_src-vit	vit
mean	7.380 ms	30 μs	22.624 ms	78 μs	2 μs	7.261 ms
min	2.641 ms	10 μs	2.2265 ms	58 μs	<1 μs	2.559 ms
max	7.780 ms	42 μs	2.7341 ms	94 μs	5 μs	7.624 ms



Pipeline Microphone -> Voice seeker -> VIT

microphone -> voice seeker -> VIT	streamer	audio_src_start	pcm_read	audio_src_end	link audio_src-voiceseeker	voice-seeker	link voiceseeker-vit	vit
mean	9.916 ms	22 μs	20.084 ms	84 μs	4 μs	2.386 ms	13 μs	7.407 ms
min	4.983 ms	19 μs	19.738 ms	72 μs	<1 μs	2.228 ms	2 μs	2.662 ms
max	10.423 ms	34 μs	24.777 ms	100 μs	7 μs	2.522 ms	31 μs	7.729 ms



Maestro on Zephyr

- Based on and tested with Zephyr version, given by tag v4.0.0
- Tested with Zephyr SDK version 16.4
- To see the pre-built documentation, see: [README.html](#). Also see the [documentation section](#).

Maestro sample for recording data from microphone to RAM

Description This sample records data from microphone (alias `dmic0` in devicetree) and stores them to a buffer in RAM.

Currently one PDM channel with fixed 16 kHz sample rate and 16 bit sample width is supported. For configuration options, see `Kconfig` and `prj.conf`.

User Input/Output

- Input:
None.
- Output:
UART Output:
 - Demo result: OK if everything went OK
 - Demo result: FAIL otherwise

Supported platforms Currently tested for:

- RD_RW612_BGA.

Maestro voice detection sample using VIT

Description Records data from microphone (alias `dmic0` in devicetree) and detects voice commands from selected language model. Detected commands are printed via UART.

Language model may be changed via `Kconfig` using `CONFIG_MAESTRO_EXAMPLE_VIT_LANGUAGE` selection. For other configuration options, see example's `Kconfig` and `prj.conf`.

This project requires an NXP board supported by the VIT library.

The example has to be modified if a new board needs to be added. Please create an issue in that case.

User Input/Output

- Input:
None.
- Output:
UART Output:
 - List of voice commands the model can detect (printed immediately after start)
 - `<Specific voice command>` if voice command was detected
 - Demo result: FAIL otherwise

Dependencies

- VIT library: <https://www.nxp.com/design/design-center/software/embedded-software/voice-intelligent-technology-wake-word-and-voice-command-engines:VOICE-INTELLIGENT-TECHNOLOGY>

Supported platforms

 Currently tested for:

- RD_RW612_BGA.

Maestro decoder sample

Description Tests and demonstrates decoder functionality in Maestro pipeline.

Supported decoders:

- MP3
- WAV
- AAC
- FLAC
- OPUS with OGG envelop
- (RAW OPUS - TBD)

Data Input:

- Prepared encoded audio data (part of Maestro repository, folder `zephyr/audioTracks`)
- Prepared decoded audio data (RAW PCM format, part of Maestro repository, folder `zephyr/audioTracks`)

Function:

1. Loads encoded data into source buffer stored in RAM
2. Decodes audio data using selected decoder and stores data in RAM
3. Compares prepared data with decoded data to check if its the same
4. Prints Demo result: OK or Demo result: FAIL via UART

User Input/Output

- Input:
None
- Output:
UART Output
 - Demo result: OK if everything went OK
 - Demo result: FAIL otherwise

Dependencies

- Audio voice component library (pulled in by Maestro's west), containing Decoder libraries

Configuration

- See prj.conf for user input sections
 - Selecting decoder may be done by enabling CONFIG_MAESTRO_EXAMPLE_DECODER_SELECTED in prj.conf file. When no decoder is selected, default one (WAV) is used instead.
 - System settings should be modified (stack size, heap size) based on selected decoder and system capabilities/requirements in prj.conf.
- For other configuration options, see example's Kconfig and prj.conf.

Supported platforms Currently tested for:

- RD_RW612_BGA - Working decoders: FLAC, WAV, OPUS OGG

Maestro encoder sample

Description Tests and demonstrates encoder functionality in Maestro pipeline.

Supported encoders:

- OPUS with OGG envelop - TBD
- RAW OPUS - TBD

Input:

- Prepared decoded audio data (RAW PCM format, part of Maestro repository)
- Prepared encoded audio data (part of Maestro repository)

Function:

1. Loads RAW data into source buffer stored in RAM
2. Encodes audio data using selected encoder and stores data in RAM
3. Compares prepared data with decoded data if same
4. Prints Demo result: OK or Demo result: FAIL via UART

Dependencies

- Audio voice component library (pulled in by Maestro's west), containing Encoder libraries

User Input/Output Input:

- None

Output:

- UART Output
 - Demo result: OK if everything went OK
 - Demo result: FAIL otherwise

Configuration

- See `prj.conf` for user input sections
 - Selecting encoder may be done by enabling `CONFIG_MAESTRO_EXAMPLE_ENCODER_SELECTED` in `prj.conf` file. When no encoder is selected, default one (OPUS) is used instead.
 - System settings should be modified (stack size, heap size) based on selected encoder and system capabilities/requirements in `prj.conf` file.
- For other configuration options, see example's `Kconfig` and `prj.conf`.

Supported platforms

 Currently tested for:

- RD_RW612_BGA - Working encoders: None.

Maestro mem2mem sample

Description Tests basic memory to memory pipeline.

Function:

1. Moves generated data with fixed size of 256B from memory source to memory sink.
2. Compares copied data to check if they're the same.
3. Returns Demo result: OK or Demo result: FAIL via UART.

- [Maestro environment setup](#)
- [Build and run Maestro example](#)
 - [Using command line](#)
 - [Using MCUXpresso for VS Code](#)
- [Folder structure](#)
- [Supported elements and libraries](#)
- [Examples support](#)
- [Creating your own example](#)
- [Documentation](#)
- [FAQ](#)

Maestro environment setup Follow these steps to set up a Maestro development environment on your machine.

1. If you haven't already, please follow [this guide](#) to set up a Zephyr development environment and its dependencies first:
 - Cmake
 - Python
 - Devicetree compiler
 - West
 - Zephyr SDK bundle

2. Get Maestro. You can pick either of the options listed below. If you need help deciding which option is the best fit for your needs, please see the [FAQ](#).

- Freestanding Maestro - This option pulls in only Maestro's necessary dependencies.

Run:

```
1. west init -m <maestro repository url> --mr <revision> --mf west-freestanding.yml
   ↳ <foldername>
2. cd <foldername>
3. west update
```

- Maestro as a Zephyr module

To include Maestro into Zephyr, update Zephyr's west.yml file:

```
projects:
  name: maestro
  url: <maestro repository url>
  revision: <revision with Zephyr support>
  path: modules/audio/maestro
  import: west.yml
```

Then run `west update maestro` command.

Build and run Maestro example These steps will guide you through building and running Maestro samples. You can use either the command line utilizing Zephyr's powerful `west` tool or you can use VS Code's GUI. Detailed steps for both options are listed below.

Using command line See Zephyr's [Building, Flashing and Debugging](#) guide if you aren't familiar with it yet.

1. To **build** a project, run:

```
west build -b <board> -d <output build directory> <path to example> -p
```

For example, this compiles VIT example for `rd_rw612_bga` board:

```
1. cd maestro/zephyr
2. west build -b rd_rw612_bga -d build samples/vit -p
```

2. To **run** a project, run:

```
west flash -d <directory>
```

e.g.:

```
west flash -d build
```

3. To **debug** a project, run:

```
west debug -d <directory>
```

e.g.:

```
west debug -d build
```

Using MCUXpresso for VS Code For this you have to have NXP's [MCUXpresso for VS Code extension](#) installed.

1. Import your topdir as a repository to MCUXPresso for VS Code:

- Open the MCUXpresso Extension. In the *Quickstart Panel* click *Import Repository*.
 - In the displayed menu click *LOCAL* tab and select the folder location of your *topdir*.
 - Click *Import*.
 - The repository is successfully added to the *Installed Repositories* view once the import is successful.
2. To import any project from the imported repository:
 - In the *Quickstart Panel* click *Import Example from Repository*.
 - For **Repository** select *your imported* repository.
 - For **Zephyr SDK** the installed Zephyr SDK is selected automatically. If not, select one.
 - For **Board** select your board (*make sure you've selected the correct revision*).
 - For **Template** select the folder path to your project.
 - Click the *Create* button.
 3. Build the project by clicking the *Build Selected* icon (displayed on hover) in the extension's *Projects* view. After the build, the debug console window displays the memory usage (or compiler errors if any).
 4. Debug the project by clicking the *Debug* (play) icon (displayed on hover) in the extension's *Projects* view.
 5. The execution will pause. To continue execution click *Continue* on the debug options.
 6. In the *SERIAL MONITOR* tab of your console panel, the application prints the Zephyr boot banner during startup and then prints the test results.

Folder structure

```
maestro/
...
zephyr/      All Zephyr related files
samples/     Sample examples
tests/       Tests
audioTracks/ Audio tracks for testing
doc/         Documentation configuration for Sphinx
wrappers/    NXP SDK Wrappers
scripts/     Helper scripts, mostly for testing
module.yml   Defines module name, Cmake and Kconfig locations
CMakeList.txt Defines module's build process
Kconfig      Defines module's configuration
osa/         Deprecated. OSA port for Zephyr
...
```

Supported elements and libraries Here is the list of all features currently supported in Maestro on Zephyr. Our goal is to support all features in Maestro on Zephyr that are already supported in Maestro on NXP's SDK and to extend them further.

Supported elements:

- Memory source
- Memory sink
- Audio source
- Audio sink
- Process sink

- Decoder
- Encoder

Supported decoders:

- WAV
- MP3
- FLAC
- OPUS OGG
- AAC

Supported encoders:

- OPUS RAW

Supported libraries:

- VIT

Examples support All included examples use UART as output. Examples are located in `zephyr/tests` and `zephyr/samples` directories.

List of included examples:

- [Maestro sample for recording data from microphone to RAM](#)
- [Maestro voice detection sample using VIT](#)
- [Maestro encoder sample](#)
- [Maestro decoder sample](#)
- [Maestro mem2mem sample](#)

Examples support for specific boards:

Example	RDRW612BGA	LPCx-presso55s69	MIMXRT1060EVKE	MIMXRT1170EVKB
Record	YES	TO BE TESTED	TO BE TESTED	TO BE TESTED
VIT	YES	TO BE TESTED	TO BE TESTED	TO BE TESTED
Encoder	In progress: OPUS RAW	TO BE TESTED	TO BE TESTED	TO BE TESTED
Decoder	YES - WAV, FLAC, OPUS OGG	TO BE TESTED	TO BE TESTED	TO BE TESTED
Mem2mem	YES	TO BE TESTED	TO BE TESTED	TO BE TESTED

Creating your own example There are two ways to create your own example - you can either one of the included examples as a reference or you can create your own example from scratch by hand.

When creating your own example from scratch, set `CONFIG_MAESTRO_AUDIO_FRAMEWORK=y` in your `prj.conf` file. Then you can start enabling specific elements by setting `CONFIG_MAESTRO_ELEMENT_<NAME>_ENABLE=y`.

However, the recommended way to edit config options is to open `gui-config` (or `menuconfig`) by calling `west build -t guiconfig`. Then you can use the graphical interface to interactively turn on/off the features you need.

Documentation Please note, Maestro documentation is under reconstruction. It is currently mixing several tools and formats.

To see the pre-generated Maestro Zephyr documentation, see `zephyr/doc/doc/README.html`

To generate the Zephyr documentation, go under `zephyr/doc` folder and execute `make html`. Sphinx version `sphinx-build 8.1.3` must be installed. Open `doc/doc/html/README.html` afterwards.

To see Maestro core documentation, go to the Maestro top directory and see `README.md`.

FAQ

1. Should I choose the freestanding version of Maestro or should integrate it into my west instead?
 - Freestanding version of Maestro pulls in all the dependencies it needs including Zephyr itself.
 - Integrating it as a module is easier if you already have your Zephyr environment set up.

Maestro Audio Framework changelog

2.0.0 (newest)

- Added Zephyr port, see [Zephyr README](#).
 - Possible to use standalone version, pulling its own Zephyr and dependencies
 - Possible to import it as a module in your Zephyr project
- Changed build system - newly uses Kconfig and Cmake
- Supports NXP MCUXSDK (previously 2.x)
- Changed folder structure and names to improve readability (description may be found in [README](#))
- Removed audio libraries and placed into audio-voice-components repository
- Added libraries are pulled into the build via Kconfig and Cmake
- Changed Maestro library core - minor changes

1.8.0

- New platforms support: MCX-N5XX-EVK, FRDMMCXN236 and RD-RW612-BGA
- Fixed compilation warnings
- Documentation improvements and updates
 - Added section with processing time information
 - Added application state diagrams
- Various updates and fixes

1.7.0

- Removed EAP support for future SDK releases
- Created new API for audio_sink and audio_src to support USB source, sink
- ASRC library integrated
- License changed to BSD 3-Clause
- Improved pipeline creation API
- Fixed compilation warnings in Opus
- Various other improvements and bug fixes

1.6.0

- Up to 2 parallel pipelines supported
- Synchronous Sample Rate Converter support Added
- Various improvements and bug fixes

1.5.0

- Enabled switching from 2 to 4 channel output during processing
- PadReturn type has been replaced by FlowReturn
- Support of AAC, WAV, FLAC decoders
- Renamed eap element to audio_proc element
- Added audio_proc to VIT pipeline to support VoiceSeeker
- Minor bug fixes

1.4.0

- Use Opusfile lib for Ogg Opus decoder
- Refactor code, fix issues found in unit tests
- Various bug fixes

1.3.0

- Make Maestro framework open source (except mp3 and wav decoder)
- Refactor code, remove unused parts, add comments

1.2.0

- Unified buffering in audio source, audio sink
- Various improvements and bug fixes

1.0_rev0

- Initial version of framework with support for Cortex-M7 platforms

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

Overview The purpose of this document is to describes the [FreeRTOS kernel repo](#) integration into the [NXP MCUXpresso Software Development Kit: mcuxsdk](#). MCUXpresso SDK provides a comprehensive development solutions designed to optimize, ease, and help accelerate embedded system development of applications based on MCUs from NXP. This project involves the FreeRTOS kernel repo fork with:

- cmake and Kconfig support to allow the configuration and build in MCUXpresso SDK ecosystem
- FreeRTOS OS additions, such as [FreeRTOS driver wrappers](#), RTOS ready FatFs file system, and the implementation of FreeRTOS tickless mode

The history of changes in FreeRTOS kernel repo for MCUXpresso SDK are summarized in [CHANGELOG_mcuxsdk.md](#) file.

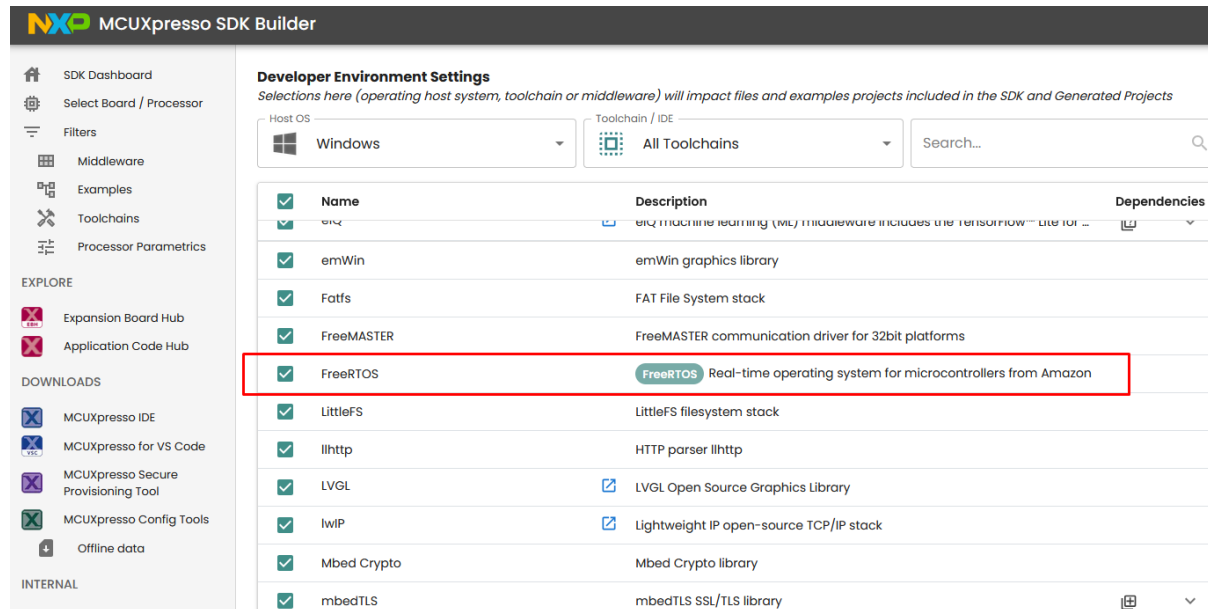
The MCUXpresso SDK framework also contains a set of FreeRTOS examples which show basic FreeRTOS OS features. This makes it easy to start a new FreeRTOS project or begin experimenting with FreeRTOS OS. Selected drivers and middleware are RTOS ready with related FreeRTOS adaptation layer.

FreeRTOS example applications The FreeRTOS examples are written to demonstrate basic FreeRTOS features and the interaction between peripheral drivers and the RTOS.

List of examples The list of freertos_examples, their description and availability for individual supported MCUXpresso SDK development boards can be obtained here: https://mcuxpresso.nxp.com/mcuxsdk/latest/html/examples/freertos_examples/index.html

Location of examples The FreeRTOS examples are located in [mcuxsdk-examples](#) repository, see the `freertos_examples` folder.

Once using MCUXpresso SDK zip packages created via the [MCUXpresso SDK Builder](#) the FreeRTOS kernel library and associated `freertos_examples` are added into final zip package once FreeRTOS components is selected on the Developer Environment Settings page:



The FreeRTOS examples in MCUXpresso SDK zip packages are located in `<MCUXpressoSDK_install_dir>/boards/<board_name>/freertos_examples/` subfolders.

Building a FreeRTOS example application For information how to use the cmake and Kconfig based build and configuration system and how to build `freertos_examples` visit: [MCUXpresso SDK documentation for Build And Configuration MCUXpresso SDK Getting Start Guide](#)

Tip: To list all FreeRTOS example projects and targets that can be built via the west build command, use this `west list_project` command in `mcuxsdk` workspace:

```
west list_project -p examples/freertos_examples
```

FreeRTOS aware debugger plugin NXP provides FreeRTOS task aware debugger for GDB. The plugin is compatible with Eclipse-based (MCUXpressoIDE) and is available after the installation.

Task List (FreeRTOS)							
TCB#	Task Name	Task Handle	Task State	Priority	Stack Usage	Event Object	Runtime
1	task_one	0x1fffecc8	Blocked	1 (1)	0 B / 880 B	MyCountingSemaphore (Rx)	0x0 (0.0%)
2	task_two	0x1ffff130	Blocked	2 (2)	0 B / 888 B	MyCountingSemaphore (Rx)	0x1 (0.1%)
3	IDLE	0x1ffff330	Running	0 (0)	0 B / 296 B		0x3e5 (99.6%)
4	Tmr Svc	0x1ffff6b8	Blocked	17 (17)	28 B / 672 B	TmrQ (Rx)	0x3 (0.3%)

FreeRTOS kernel for MCUXpresso SDK ChangeLog

Changelog FreeRTOS kernel for MCUXpresso SDK All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

[Unreleased]

Added

- Kconfig added CONFIG_FREERTOS_USE_CUSTOM_CONFIG_FRAGMENT config to optionally include custom FreeRTOSConfig fragment include file FreeRTOSConfig_frag.h. File must be provided by application.
- Added missing Kconfig option for configUSE_PICOLIBC_TLS.
- Add correct header files to build when configUSE_NEWLIB_REENTRANT and configUSE_PICOLIBC_TLS is selected in config.

[11.1.0_rev0]

- update amazon freertos version

[11.0.1_rev0]

- update amazon freertos version

[10.5.1_rev0]

- update amazon freertos version

[10.4.3_rev1]

- Apply CM33 security fix from 10.4.3-LTS-Patch-2. See rtos\freertos\freertos_kernel\History.txt
- Apply CM33 security fix from 10.4.3-LTS-Patch-1. See rtos\freertos\freertos_kernel\History.txt

[10.4.3_rev0]

- update amazon freertos version.

[10.4.3_rev0]

- update amazon freertos version.

[9.0.0_rev3]

- New features:
 - Tickless idle mode support for Cortex-A7. Add fsl_tickless_epit.c and fsl_tickless_generic.h in portable/IAR/ARM_CA9 folder.
 - Enabled float context saving in IAR for Cortex-A7. Added configUSE_TASK_FPU_SUPPORT macros. Modified port.c and portmacro.h in portable/IAR/ARM_CA9 folder.
- Other changes:
 - Transformed ARM_CM core specific tickless low power support into generic form under freertos/Source/portable/low_power_tickless/.

[9.0.0_rev2]

- New features:
 - Enabled MCUXpresso thread aware debugging. Add `freertos_tasks_c_additions.h` and `configINCLUDE_FREERTOS_TASK_C_ADDITIONS_H` and `configFREERTOS_MEMORY_SCHEME` macros.

[9.0.0_rev1]

- New features:
 - Enabled `-flto` optimization in GCC by adding `attribute((used))` for `vTaskSwitchContext`.
 - Enabled KDS Task Aware Debugger. Apply FreeRTOS patch to enable `configRECORD_STACK_HIGH_ADDRESS` macro. Modified files are `task.c` and `FreeRTOS.h`.

[9.0.0_rev0]

- New features:
 - Example `freertos_sem_static`.
 - Static allocation support RTOS driver wrappers.
- Other changes:
 - Tickless idle rework. Support for different timers is in separated files (`fsl_tickless_systick.c`, `fsl_tickless_lptmr.c`).
 - Removed configuration option `configSYSTICK_USE_LOW_POWER_TIMER`. Low power timer is now selected by linking of appropriate file `fsl_tickless_lptmr.c`.
 - Removed `configOVERRIDE_DEFAULT_TICK_CONFIGURATION` in RVDS port. Use of `attribute((weak))` is the preferred solution. Not same as `_weak`!

[8.2.3]

- New features:
 - Tickless idle mode support.
 - Added template application for Kinetis Expert (KEx) tool (`template_application`).
- Other changes:
 - Folder structure reduction. Keep only Kinetis related parts.

FreeRTOS kernel Readme

MCUXpresso SDK: FreeRTOS kernel This repository is a fork of FreeRTOS kernel (<https://github.com/FreeRTOS/FreeRTOS-Kernel>)(11.1.0). Modifications have been made to adapt to NXP MCUXpresso SDK. `CMakeLists.txt` and `Kconfig` added to enable FreeRTOS kernel repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository `mcuxsdk-manifests`(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

For more information about the FreeRTOS kernel repo adoption see [README_mcuxsdk.md: FreeRTOS kernel for MCUXpresso SDK](#) document.



Getting started This repository contains FreeRTOS kernel source/header files and kernel ports only. This repository is referenced as a submodule in [FreeRTOS/FreeRTOS](#) repository, which contains pre-configured demo application projects under [FreeRTOS/Demo](#) directory.

The easiest way to use FreeRTOS is to start with one of the pre-configured demo application projects. That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the [FreeRTOS Kernel Quick Start Guide](#) for detailed instructions and other useful links.

Additionally, for FreeRTOS kernel feature information refer to the [Developer Documentation](#), and [API Reference](#).

Also for contributing and creating a Pull Request please refer to *the instructions here*.

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the [FreeRTOS Community Support Forum](#).

To consume FreeRTOS-Kernel

Consume with CMake If using CMake, it is recommended to use this repository using FetchContent. Add the following into your project's main or a subdirectory's CMakeLists.txt:

- Define the source and version/tag you want to use:

```
FetchContent_Declare( freertos_kernel
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Kernel.git
  GIT_TAG        main #Note: Best practice to use specific git-hash or tagged version
)
```

In case you prefer to add it as a git submodule, do:

```
git submodule add https://github.com/FreeRTOS/FreeRTOS-Kernel.git <path of the submodule>
git submodule update --init
```

- Add a freertos_config library (typically an INTERFACE library) The following assumes the directory structure:

– include/FreeRTOSConfig.h

```
add_library(freertos_config INTERFACE)

target_include_directories(freertos_config SYSTEM
  INTERFACE
    include
)

target_compile_definitions(freertos_config
  INTERFACE
    projCOVERAGE_TEST=0
)
```

In case you installed FreeRTOS-Kernel as a submodule, you will have to add it as a subdirectory:

```
add_subdirectory(${FREERTOS_PATH})
```

- Configure the FreeRTOS-Kernel and make it available
 - this particular example supports a native and cross-compiled build option.


```
set( FREERTOS_HEAP "4" CACHE STRING "" FORCE)
# Select the native compile PORT
set( FREERTOS_PORT "GCC_POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  set(FREERTOS_PORT "GCC_ARM_CA9" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_kernel)
```

- In case of cross compilation, you should also add the following to `freertos_config`:

```
target_compile_definitions(freertos_config INTERFACE ${definitions})
target_compile_options(freertos_config INTERFACE ${options})
```

Consuming stand-alone - Cloning this repository

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/FreeRTOS-Kernel.git
```

Using SSH:

```
git clone git@github.com:FreeRTOS/FreeRTOS-Kernel.git
```

Repository structure

- The root of this repository contains the three files that are common to every port - `list.c`, `queue.c` and `tasks.c`. The kernel is contained within these three files. `croutine.c` implements the optional co-routine functionality - which is normally only used on very memory limited systems.
- The `./portable` directory contains the files that are specific to a particular microcontroller and/or compiler. See the readme file in the `./portable` directory for more information.
- The `./include` directory contains the real time kernel header files.
- The `./template_configuration` directory contains a sample `FreeRTOSConfig.h` to help jumpstart a new project. See the *FreeRTOSConfig.h* file for instructions.

Code Formatting FreeRTOS files are formatted using the “`uncrustify`” tool. The configuration file used by `uncrustify` can be found in the [FreeRTOS/CI-CD-GitHub-Actions's uncrustify.cfg](#) file.

Line Endings File checked into the FreeRTOS-Kernel repository use unix-style LF line endings for the best compatibility with git.

For optimal compatibility with Microsoft Windows tools, it is best to enable the git `autocrlf` feature. You can enable this setting for the current repository using the following command:

```
git config core.autocrlf true
```

Git History Optimizations Some commits in this repository perform large refactors which touch many lines and lead to unwanted behavior when using the `git blame` command. You can configure git to ignore the list of large refactor commits in this repository with the following command:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

Spelling and Formatting We recommend using [Visual Studio Code](#), commonly referred to as VSCode, when working on the FreeRTOS-Kernel. The FreeRTOS-Kernel also uses [cSpell](#) as part of its spelling check. The config file for which can be found at [cspell.config.yaml](#). There is additionally a [cSpell plugin for VSCode](#) that can be used as well. `.cSpellWords.txt` contains words that are not traditionally found in an English dictionary. It is used by the spellchecker to verify the various jargon, variable names, and other odd words used in the FreeRTOS code base are correct. If your pull request fails to pass the spelling and you believe this is a mistake, then add the word to `.cSpellWords.txt`. When adding a word please then sort the list, which can be done by running the bash command: `sort -u .cSpellWords.txt -o .cSpellWords.txt`. Note that only the FreeRTOS-Kernel Source Files, *include*, *portable/MemMang*, and *portable/Common* files are checked for proper spelling, and formatting at this time.

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

MCUXpresso SDK: backoffAlgorithm Library This repository is a fork of backoffAlgorithm library (<https://github.com/FreeRTOS/backoffalgorithm>)(1.3.0). Modifications have been made to adapt to NXP MCUXpresso SDK. `CMakeLists.txt` and `Kconfig` added to enable backoffAlgorithm repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository `mcuxsdk-manifests`(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

backoffAlgorithm Library This repository contains the backoffAlgorithm library, a utility library to calculate backoff period using an exponential backoff with jitter algorithm for retrying network operations (like failed network connection with server). This library uses the “Full Jitter” strategy for the exponential backoff with jitter algorithm. More information about the algorithm can be seen in the [Exponential Backoff and Jitter](#) AWS blog.

The backoffAlgorithm library is distributed under the *MIT Open Source License*.

Exponential backoff with jitter is typically used when retrying a failed network connection or operation request with the server. An exponential backoff with jitter helps to mitigate failed network operations with servers, that are caused due to network congestion or high request load on the server, by spreading out retry requests across multiple devices attempting network operations. Besides, in an environment with poor connectivity, a client can get disconnected at any time. A backoff strategy helps the client to conserve battery by not repeatedly attempting reconnections when they are unlikely to succeed.

See memory requirements for this library [here](#).

backoffAlgorithm v1.3.0 source code is part of the FreeRTOS 202210.00 LTS release.

backoffAlgorithm v1.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Reference example The example below shows how to use the backoffAlgorithm library on a POSIX platform to retry a DNS resolution query for amazon.com.

```
#include "backoff_algorithm.h"
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <unistd.h>
#include <time.h>

/* The maximum number of retries for the example code. */
#define RETRY_MAX_ATTEMPTS      ( 5U )

/* The maximum back-off delay (in milliseconds) for between retries in the example. */
#define RETRY_MAX_BACKOFF_DELAY_MS ( 5000U )

/* The base back-off delay (in milliseconds) for retry configuration in the example. */
#define RETRY_BACKOFF_BASE_MS   ( 500U )

int main()
{
    /* Variables used in this example. */
    BackoffAlgorithmStatus_t retryStatus = BackoffAlgorithmSuccess;
    BackoffAlgorithmContext_t retryParams;
    char serverAddress[] = "amazon.com";
    uint16_t nextRetryBackoff = 0;

    int32_t dnsStatus = -1;
    struct addrinfo hints;
    struct addrinfo ** pListHead = NULL;
    struct timespec tp;

    /* Add hints to retrieve only TCP sockets in getaddrinfo. */
    ( void ) memset( &hints, 0, sizeof( hints ) );

    /* Address family of either IPv4 or IPv6. */
    hints.ai_family = AF_UNSPEC;
    /* TCP Socket. */
    hints.ai_socktype = ( int32_t ) SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    /* Initialize reconnect attempts and interval. */
    BackoffAlgorithm_InitializeParams( &retryParams,
                                      RETRY_BACKOFF_BASE_MS,
                                      RETRY_MAX_BACKOFF_DELAY_MS,
                                      RETRY_MAX_ATTEMPTS );

    /* Seed the pseudo random number generator used in this example (with call to
     * rand() function provided by ISO C standard library) for use in backoff period
     * calculation when retrying failed DNS resolution. */

    /* Get current time to seed pseudo random number generator. */
    ( void ) clock_gettime( CLOCK_REALTIME, &tp );
    /* Seed pseudo random number generator with seconds. */
    srand( tp.tv_sec );

    do
    {
        /* Perform a DNS lookup on the given host name. */
        dnsStatus = getaddrinfo( serverAddress, NULL, &hints, pListHead );
    }
```

(continues on next page)

(continued from previous page)

```

/* Retry if DNS resolution query failed. */
if( dnsStatus != 0 )
{
    /* Generate a random number and get back-off value (in milliseconds) for the next retry.
     * Note: It is recommended to use a random number generator that is seeded with
     * device-specific entropy source so that backoff calculation across devices is different
     * and possibility of network collision between devices attempting retries can be avoided.
     *
     * For the simplicity of this code example, the pseudo random number generator, rand()
     * function is used. */
    retryStatus = BackoffAlgorithm_GetNextBackoff( &retryParams, rand(), &nextRetryBackoff );

    /* Wait for the calculated backoff period before the next retry attempt of querying DNS.
     * As usleep() takes nanoseconds as the parameter, we multiply the backoff period by 1000. */
    ( void ) usleep( nextRetryBackoff * 1000U );
}
} while( ( dnsStatus != 0 ) && ( retryStatus != BackoffAlgorithmRetriesExhausted ) );

return dnsStatus;
}

```

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

Additionally, the library uses a header file introduced in ISO C99, *stdint.h*. For compilers that do not provide this header file, the *source/include* directory contains *stdint.readme*, which can be renamed to *stdint.h* to build the backoffAlgorithm library.

For instance, if the example above is copied to a file named *example.c*, *gcc* can be used like so:

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

gcc can also produce an output file to be linked:

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

Building unit tests

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

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

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

Platform Prerequisites

- For running unit tests
 - C89 or later compiler like *gcc*
 - CMake 3.13.0 or later
- For running the coverage target, *gcov* is additionally required.

Steps to build Unit Tests

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

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

4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

MCUXpresso SDK: coreHTTP Client Library

This repository is a fork of coreHTTP Client library (<https://github.com/FreeRTOS/corehttp>)(3.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreHTTP Client repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

coreHTTP Client Library

This repository contains a C language HTTP client library designed for embedded platforms. It has no dependencies on any additional libraries other than the standard C library, [lhttp](#), and a customer-implemented transport interface. This library is distributed under the *MIT Open Source License*.

This library has gone through code quality checks including verification that no function has a [GNU Complexity](#) score over 8. This library has also undergone both static code analysis from [Coverity static analysis](#), and validation of memory safety and data structure invariance through the [CBMC automated reasoning tool](#).

See memory requirements for this library [here](#).

coreHTTP v3.0.0 source code is part of the FreeRTOS 202210.00 LTS release.

coreHTTP v2.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

coreHTTP Config File The HTTP client library exposes configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core_http_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named *core_http_config.h* can be provided by the user application to the library.

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

The HTTP client library can be built by either:

- Defining a `core_http_config.h` file in the application, and adding it to the include directories for the library build. **OR**
- Defining the `HTTP_DO_NOT_USE_CUSTOM_CONFIG` preprocessor macro for the library build.

Building the Library The `httpFilePaths.cmake` file contains the information of all source files and header include paths required to build the HTTP client library.

As mentioned in the *previous section*, either a custom config file (i.e. `core_http_config.h`) OR `HTTP_DO_NOT_USE_CUSTOM_CONFIG` macro needs to be provided to build the HTTP client library.

For a CMake example of building the HTTP library with the `httpFilePaths.cmake` file, refer to the `coverity_analysis` library target in `test/CMakeLists.txt` file.

Building Unit Tests

Platform Prerequisites

- For running unit tests, the following are required:
 - **C90 compiler** like `gcc`
 - **CMake 3.13.0 or later**
 - **Ruby 2.0.0 or later** is required for this repository's [CMock test framework](#).
- For running the coverage target, the following are required:
 - `gcov`
 - `lcov`

Steps to build Unit Tests

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

CBMC To learn more about CBMC and proofs specifically, review the training material [here](#).

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

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

Reference examples The AWS IoT Device SDK for Embedded C repository contains demos of using the HTTP client library [here](#) on a POSIX platform. These can be used as reference examples for the library API.

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

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

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

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

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

4.1.5 corejson

JSON parser.

Readme

MCUXpresso SDK: coreJSON Library This repository is a fork of coreJSON library (<https://github.com/FreeRTOS/corejson>)(3.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreJSON repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

coreJSON Library This repository contains the coreJSON library, a parser that strictly enforces the ECMA-404 JSON standard and is suitable for low memory footprint embedded devices. The coreJSON library is distributed under the *MIT Open Source License*.

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

See memory requirements for this library [here](#).

coreJSON v3.2.0 source code is part of the FreeRTOS 202210.00 LTS release.

coreJSON v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Reference example


```

#include <stdio.h>
#include "core_json.h"

int main()
{
    // Variables used in this example.
    JSONStatus_t result;
    char buffer[] = "{\"foo\":\"abc\", \"bar\":{\"foo\":\"xyz\"}}";
    size_t bufferLength = sizeof( buffer ) - 1;
    char queryKey[] = "bar.foo";
    size_t queryKeyLength = sizeof( queryKey ) - 1;
    char * value;
    size_t valueLength;

    // Calling JSON_Validate() is not necessary if the document is guaranteed to be valid.
    result = JSON_Validate( buffer, bufferLength );

    if( result == JSONSuccess )
    {
        result = JSON_Search( buffer, bufferLength, queryKey, queryKeyLength,
                             &value, &valueLength );
    }

    if( result == JSONSuccess )
    {
        // The pointer "value" will point to a location in the "buffer".
        char save = value[ valueLength ];
        // After saving the character, set it to a null byte for printing.
        value[ valueLength ] = '\0';
        // "Found: bar.foo -> xyz" will be printed.
        printf( "Found: %s -> %s\n", queryKey, value );
        // Restore the original character.
        value[ valueLength ] = save;
    }

    return 0;
}

```

A search may descend through nested objects when the queryKey contains matching key strings joined by a separator, .. In the example above, bar has the value { "foo": "xyz" }. Therefore, a search for query key bar.foo would output xyz.

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

Additionally, the library uses 2 header files introduced in ISO C99, *stdbool.h* and *stdint.h*. For compilers that do not provide this header file, the *source/include* directory contains *stdbool.readme* and *stdint.readme*, which can be renamed to *stdbool.h* and *stdint.h* respectively.

For instance, if the example above is copied to a file named *example.c*, *gcc* can be used like so:

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

gcc can also produce an output file to be linked:

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

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

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

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

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

Building unit tests

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

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

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

Platform Prerequisites

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

Steps to build Unit Tests

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

CBMC To learn more about CBMC and proofs specifically, review the training material [here](#).

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

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

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

4.1.6 coremqtt

MQTT publish/subscribe messaging library.

MCUXpresso SDK: coreMQTT Library

This repository is a fork of coreMQTT library (<https://github.com/FreeRTOS/coremqtt>)(2.1.1). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

coreMQTT Client Library

This repository contains the coreMQTT library that has been optimized for a low memory footprint. The coreMQTT library is compliant with the [MQTT 3.1.1](#) standard. It has no dependencies on any additional libraries other than the standard C library, a customer-implemented network transport interface, and *optionally* a user-implemented platform time function. This library is distributed under the *MIT Open Source License*.

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

See memory requirements for this library [here](#).

coreMQTT v2.1.1 source code is part of the FreeRTOS 202210.01 LTS release.

MQTT Config File The MQTT client library exposes build configuration macros that are required for building the library. A list of all the configurations and their default values are defined in *core_mqtt_config_defaults.h*. To provide custom values for the configuration macros, a custom config file named *core_mqtt_config.h* can be provided by the application to the library.

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

Thus, the MQTT library can be built by either:

- Defining a *core_mqtt_config.h* file in the application, and adding it to the include directories list of the library
- OR**
- Defining the `MQTT_DO_NOT_USE_CUSTOM_CONFIG` preprocessor macro for the library build.

Sending metrics to AWS IoT When establishing a connection with AWS IoT, users can optionally report the Operating System, Hardware Platform and MQTT client version information of their device to AWS. This information can help AWS IoT provide faster issue resolution and technical support. If users want to report this information, they can send a specially formatted string (see below) in the username field of the MQTT CONNECT packet.

Format

The format of the username string with metrics is:

```
<Actual_Username>?SDK=<OS_Name>&Version=<OS_Version>&Platform=<Hardware_Platform>&MQTTLib=<MQTT_Library_name>@<MQTT_Library_version>
```

Where

- <Actual_Username> is the actual username used for authentication, if username and password are used for authentication. When username and password based authentication is not used, this is an empty value.
- <OS_Name> is the Operating System the application is running on (e.g. FreeRTOS)
- <OS_Version> is the version number of the Operating System (e.g. V10.4.3)
- <Hardware_Platform> is the Hardware Platform the application is running on (e.g. WinSim)
- <MQTT_Library_name> is the MQTT Client library being used (e.g. coreMQTT)
- <MQTT_Library_version> is the version of the MQTT Client library being used (e.g. 1.0.2)

Example

- Actual_Username = "iotuser", OS_Name = FreeRTOS, OS_Version = V10.4.3, Hardware_Platform_Name = WinSim, MQTT_Library_Name = coremqtt, MQTT_Library_version = 2.1.1. If username is not used, then "iotuser" can be removed.

```
/* Username string:
 * iotuser?SDK=FreeRTOS&Version=v10.4.3&Platform=WinSim&MQTTLib=coremqtt@2.1.1
 */

#define OS_NAME           "FreeRTOS"
#define OS_VERSION        "V10.4.3"
#define HARDWARE_PLATFORM_NAME  "WinSim"
#define MQTT_LIB          "coremqtt@2.1.1"

#define USERNAME_STRING   "iotuser?SDK=" OS_NAME "&Version=" OS_VERSION "&Platform=" HARDWARE_PLATFORM_NAME "&MQTTLib=" MQTT_LIB
#define USERNAME_STRING_LENGTH  ( ( uint16_t ) ( sizeof( USERNAME_STRING ) - 1 ) )

MQTTConnectInfo_t connectInfo;
connectInfo.userName = USERNAME_STRING;
connectInfo.userNameLength = USERNAME_STRING_LENGTH;
mqttStatus = MQTT_Connect( pMqttContext, &connectInfo, NULL, CONNACK_RECV_TIMEOUT_MS,
↳ pSessionPresent );
```

Upgrading to v2.0.0 and above With coreMQTT versions >=v2.0.0, there are breaking changes. Please refer to the *coreMQTT version >=v2.0.0 Migration Guide*.

Building the Library The *mqttFilePaths.cmake* file contains the information of all source files and the header include path required to build the MQTT library.

Additionally, the MQTT library requires two header files that are not part of the ISO C90 standard library, *stdbool.h* and *stdint.h*. For compilers that do not provide these header files, the

source/include directory contains the files *stdbool.readme* and *stdint.readme*, which can be renamed to *stdbool.h* and *stdint.h*, respectively, to provide the type definitions required by MQTT.

As mentioned in the previous section, either a custom config file (i.e. *core_mqtt_config.h*) OR *MQTT_DO_NOT_USE_CUSTOM_CONFIG* macro needs to be provided to build the MQTT library.

For a CMake example of building the MQTT library with the *mqttFilePaths.cmake* file, refer to the *coverity_analysis* library target in *test/CMakeLists.txt* file.

Building Unit Tests

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

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

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

Platform Prerequisites

- Docker

or the following:

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

Steps to build Unit Tests

1. If using docker, launch the container:
 1. `docker build -t coremqtt .`
 2. `docker run -it -v "$PWD":/workspaces/coreMQTT -w /workspaces/coreMQTT coremqtt`
2. Go to the root directory of this repository. (Make sure that the **CMock** submodule is cloned as described [above](#))
3. Run the *cmake* command: `cmake -S test -B build`
4. Run this command to build the library and unit tests: `make -C build all`
5. The generated test executables will be present in *build/bin/tests* folder.
6. Run `cd build && ctest` to execute all tests and view the test run summary.

CBMC To learn more about CBMC and proofs specifically, review the training material [here](#).

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

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

Reference examples Please refer to the demos of the MQTT client library in the following locations for reference examples on POSIX and FreeRTOS platforms:

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

Documentation

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

Location
AWS IoT Device SDK for Embedded C
FreeRTOS.org

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

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

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

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

4.1.7 coremqtt-agent

The coreMQTT Agent library is a high level API that adds thread safety to the coreMQTT library.

Readme

MCUXpresso SDK: coreMQTT Agent Library This repository is a fork of coreMQTT Agent library (<https://github.com/FreeRTOS/coremqtt-agent>)(1.2.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable coreMQTT Agent repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

coreMQTT Agent Library The coreMQTT Agent library is a high level API that adds thread safety to the [coreMQTT](#) library. The library provides thread safe equivalents to the coreMQTT's APIs, greatly simplifying its use in multi-threaded environments. The coreMQTT Agent library manages the MQTT connection by serializing the access to the coreMQTT library and reducing implementation overhead (e.g., removing the need for the application to repeatedly call to MQTT_ProcessLoop). This allows your multi-threaded applications to share the same MQTT connection, and enables you to design an embedded application without having to worry about coreMQTT thread safety.

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

See memory requirements for this library [here](#).

Cloning this repository This repo uses [Git Submodules](#) to bring in dependent components.

To clone using HTTPS:

```
git clone https://github.com/FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

Using SSH:

```
git clone git@github.com:FreeRTOS/coreMQTT-Agent.git --recurse-submodules
```

If you have downloaded the repo without using the `--recurse-submodules` argument, you need to run:

```
git submodule update --init --recursive
```

coreMQTT Agent Library Configurations The MQTT Agent library uses the same `core_mqtt_config.h` configuration file as coreMQTT, with the addition of configuration constants listed at the top of `core_mqtt_agent.h` and `core_mqtt_agent_command_functions.h`. Documentation for these configurations can be found [here](#).

To provide values for these configuration values, they must be either:

- Defined in `core_mqtt_config.h` used by coreMQTT **OR**
- Passed as compile time preprocessor macros

Porting the coreMQTT Agent Library In order to use the MQTT Agent library on a platform, you need to supply thread safe functions for the agent's *messaging interface*.

Messaging Interface Each of the following functions must be thread safe.

Function Pointer	Description
MQTTAgentMessageSend_t	A function that sends commands (as MQTTAgentCommand_t * pointers) to be received by MQTTAgent_CommandLoop. This can be implemented by pushing to a thread safe queue.
MQTTAgentMessageRecv_t	A function used by MQTTAgent_CommandLoop to receive MQTTAgentCommand_t * pointers that were sent by API functions. This can be implemented by receiving from a thread safe queue.
MQTTAgentCommandGet_t	A function that returns a pointer to an allocated MQTTAgentCommand_t structure, which is used to hold information and arguments for a command to be executed in MQTTAgent_CommandLoop(). If using dynamic memory, this can be implemented using malloc().
MQTTAgentCommandRelease_t	A function called to indicate that a command structure that had been allocated with the MQTTAgentCommandGet_t function pointer will no longer be used by the agent, so it may be freed or marked as not in use. If using dynamic memory, this can be implemented with free().

Reference implementations for the interface functions can be found in the [reference examples](#) below.

Additional Considerations

Static Memory If only static allocation is used, then the MQTTAgentCommandGet_t and MQTTAgentCommandRelease_t could instead be implemented with a pool of MQTTAgentCommand_t structures, with a queue or semaphore used to control access and provide thread safety. The below [reference examples](#) use static memory with a command pool.

Subscription Management The MQTT Agent does not track subscriptions for MQTT topics. The receipt of any incoming PUBLISH packet will result in the invocation of a single MQTTAgentIncomingPublishCallback_t callback, which is passed to MQTTAgent_Init() for initialization. If it is desired for different handlers to be invoked for different incoming topics, then the publish callback will have to manage subscriptions and fan out messages. A platform independent subscription manager example is implemented in the [reference examples](#) below.

Building the Library You can build the MQTT Agent source files that are in the *source* directory, and add *source/include* to your compiler's include path. Additionally, the MQTT Agent library requires the coreMQTT library, whose files follow the same *source/* and *source/include* pattern as the agent library; its build instructions can be found [here](#).

If using CMake, the *mqttAgentFilePaths.cmake* file contains the above information of the source files and the header include path from this repository. The same information is found for coreMQTT from *mqttFilePaths.cmake* in the *coreMQTT submodule*.

For a CMake example of building the MQTT Agent library with the *mqttAgentFilePaths.cmake* file, refer to the *coverity_analysis* library target in *test/CMakeLists.txt* file.

Building Unit Tests

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

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

Unit Test Platform Prerequisites

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

Steps to build Unit Tests

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

CBMC To learn more about CBMC and proofs specifically, review the training material [here](#).

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

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

Reference examples Please refer to the demos of the MQTT Agent library in the following locations for reference examples on FreeRTOS platforms:

Location
coreMQTT Agent Demos
FreeRTOS/FreeRTOS

Documentation The MQTT Agent API documentation can be found [here](#).

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

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

Getting help You can use your Github login to get support from both the FreeRTOS community and directly from the primary FreeRTOS developers on our [active support forum](#). You can find a list of [frequently asked questions](#) [here](#).

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

License This library is licensed under the MIT License. See the *LICENSE* file.

4.1.8 corepkcs11

PKCS #11 key management library.

Readme

MCUXpresso SDK: corePKCS11 Library This repository is a fork of PKCS #11 key management library (<https://github.com/FreeRTOS/corePKCS11/tree/v3.5.0>)(v3.5.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable corepkcs11 repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

corePKCS11 Library PKCS #11 is a standardized and widely used API for manipulating common cryptographic objects. It is important because the functions it specifies allow application software to use, create, modify, and delete cryptographic objects, without ever exposing those objects to the application's memory. For example, FreeRTOS AWS reference integrations use a small subset of the PKCS #11 API to, among other things, access the secret (private) key necessary to create a network connection that is authenticated and secured by the [Transport Layer Security \(TLS\)](#) protocol – without the application ever ‘seeing’ the key.

The Cryptoki or PKCS #11 standard defines a platform-independent API to manage and use cryptographic tokens. The name, “PKCS #11”, is used interchangeably to refer to the API itself and the standard which defines it.

This repository contains a software based mock implementation of the PKCS #11 interface (API) that uses the cryptographic functionality provided by Mbed TLS. Using a software mock enables rapid development and flexibility, but it is expected that the mock be replaced by an implementation specific to your chosen secure key storage in production devices.

Only a subset of the PKCS #11 standard is implemented, with a focus on operations involving asymmetric keys, random number generation, and hashing.

The targeted use cases include certificate and key management for TLS authentication and code-sign signature verification, on small embedded devices.

corePKCS11 is implemented on PKCS #11 v2.4.0, the full PKCS #11 standard can be found on the [oasis website](#).

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

See memory requirements for this library [here](#).

corePKCS11 v3.5.0 source code is part of the FreeRTOS 202210.00 LTS release.

corePKCS11 v3.0.0 source code is part of the FreeRTOS 202012.00 LTS release.

Purpose Generally vendors for secure cryptoprocessors such as Trusted Platform Module (TPM), Hardware Security Module (HSM), Secure Element, or any other type of secure hardware enclave, distribute a PKCS #11 implementation with the hardware. The purpose of the corePKCS11 software only mock library is therefore to provide a non hardware specific PKCS #11 implementation that allows for rapid prototyping and development before switching to a cryptoprocessor specific PKCS #11 implementation in production devices.

Since the PKCS #11 interface is defined as part of the PKCS #11 [specification](#) replacing this library with another implementation should require little porting effort, as the interface will not change. The system tests distributed in this repository can be leveraged to verify the behavior of a different implementation is similar to corePKCS11.

corePKCS11 Configuration The corePKCS11 library exposes preprocessor macros which must be defined prior to building the library. A list of all the configurations and their default values are defined in the doxygen documentation for this library.

Build Prerequisites

Library Usage For building the library the following are required:

- **A C99 compiler**
- **mbedcrypto** library from [mbedtls](#) version 2.x or 3.x.
- **pkcs11 API header(s)** available from [OASIS](#) or [OpenSC](#)

Optionally, variables from the pkcsFilePaths.cmake file may be referenced if your project uses cmake.

Integration and Unit Tests In order to run the integration and unit test suites the following are dependencies are necessary:

- **C Compiler**
- **CMake 3.13.0 or later**
- **Ruby 2.0.0 or later** required by CMock.
- **Python 3** required for configuring mbedtls.
- **git** required for fetching dependencies.
- **GNU Make** or **Ninja**

The *mbedtls*, *CMock*, and *Unity* libraries are downloaded and built automatically using the cmake FetchContent feature.

Coverage Measurement and Instrumentation The following software is required to run the coverage target:

- Linux, MacOS, or another POSIX-like environment.
- A recent version of **GCC** or **Clang** with support for gcov-like coverage instrumentation.
- **gcov** binary corresponding to your chosen compiler
- **lcov** from the [Linux Test Project](#)
- **perl** needed to run the lcov utility.

Coverage builds are validated on recent versions of Ubuntu Linux.

Running the Integration and Unit Tests

1. Navigate to the root directory of this repository in your shell.
2. Run **cmake** to construct a build tree: `cmake -S test -B build`
 - You may specify your preferred build tool by appending `-G'Unix Makefiles'` or `-GNinja` to the command above.
 - You may append `-DUNIT_TESTS=0` or `-DSYSTEM_TESTS=0` to disable Unit Tests or Integration Tests respectively.
3. Build the test binaries: `cmake --build ./build --target all`
4. Run `ctest --test-dir ./build` or `cmake --build ./build --target test` to run the tests without capturing coverage.
5. Run `cmake --build ./build --target coverage` to run the tests and capture coverage data.

CBMC To learn more about CBMC and proofs specifically, review the training material [here](#).

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

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

Reference examples The FreeRTOS-Labs repository contains demos using the PKCS #11 library [here](#) using FreeRTOS on the Windows simulator platform. These can be used as reference examples for the library API.

Porting Guide Documentation for porting corePKCS11 to a new platform can be found on the AWS [docs](#) web page.

corePKCS11 is not meant to be ported to projects that have a TPM, HSM, or other hardware for offloading crypto-processing. This library is specifically meant to be used for development and prototyping.

Related Example Implementations These projects implement the PKCS #11 interface on real hardware and have similar behavior to corePKCS11. It is preferred to use these, over corePKCS11, as they allow for offloading Cryptography to separate hardware.

- ARM's [Platform Security Architecture](#).
- Microchip's [cryptoauthlib](#).
- Infineon's [Optiga Trust X](#).

Documentation

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

Location
AWS IoT Device SDK for Embedded C FreeRTOS.org

Note that the latest included version of corePKCS11 may differ across repositories.

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

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

Security See *CONTRIBUTING* for more information.

License This library is licensed under the MIT-0 License. See the LICENSE file.

4.1.9 freertos-plus-tcp

Open source RTOS FreeRTOS Plus TCP.

Readme

MCUXpresso SDK: FreeRTOS-Plus-TCP Library This repository is a fork of FreeRTOS-Plus-TCP library (<https://github.com/FreeRTOS/freertos-plus-tcp>)(4.0.0). Modifications have been made to adapt to NXP MCUXpresso SDK. CMakeLists.txt and Kconfig added to enable FreeRTOS-Plus-TCP repo sources build in MCUXpresso SDK. It is part of the MCUXpresso SDK overall delivery which is composed of several sub-repositories/projects. Navigate to the top/parent repository mcuxsdk-manifests(<https://github.com/nxp-mcuxpresso/mcuxsdk-manifests>) for the complete delivery of MCUXpresso SDK.

Introduction This branch contains unified IPv4 and IPv6 functionalities. Refer to the Getting started Guide (found [here](#)) for more details.

FreeRTOS-Plus-TCP Library FreeRTOS-Plus-TCP is a lightweight TCP/IP stack for FreeRTOS. It provides a familiar Berkeley sockets interface, making it as simple to use and learn as possible. FreeRTOS-Plus-TCP's features and RAM footprint are fully scalable, making FreeRTOS-Plus-TCP equally applicable to smaller lower throughput microcontrollers as well as larger higher throughput microprocessors.

This library has undergone static code analysis and checks for compliance with the [MISRA coding standard](#). Any deviations from the MISRA C:2012 guidelines are documented under [MISRA Deviations](#). The library is validated for memory safety and data structure invariance through the [CBMC automated reasoning tool](#) for the functions that parse data originating from the network. The library is also protocol tested using Maxwell protocol tester for both IPv4 and IPv6.

Getting started The easiest way to use the 4.0.0 version of FreeRTOS-Plus-TCP is to refer the Getting started Guide (found [here](#)) Another way is to start with the pre-configured demo application project (found in [this directory](#)). That way you will have the correct FreeRTOS source files included, and the correct include paths configured. Once a demo application is building and executing you can remove the demo application files, and start to add in your own application source files. See the [FreeRTOS Kernel Quick Start Guide](#) for detailed instructions and other useful links.

Additionally, for FreeRTOS-Plus-TCP source code organization refer to the [Documentation](#), and [API Reference](#).

Getting help If you have any questions or need assistance troubleshooting your FreeRTOS project, we have an active community that can help on the [FreeRTOS Community Support Forum](#). Please also refer to [FAQ](#) for frequently asked questions.

Also see the [Submitting a bug/feature request](#) section of CONTRIBUTING.md for more details.

Note: All the remaining sections are generic and applies to all the versions from V3.0.0 onwards.

Upgrading to V3.0.0 and V3.1.0 In version 3.0.0 or 3.1.0, the folder structure of FreeRTOS-Plus-TCP has changed and the files have been broken down into smaller logically separated modules. This change makes the code more modular and conducive to unit-tests. FreeRTOS-Plus-TCP V3.0.0 improves the robustness, security, and modularity of the library. Version 3.0.0 adds comprehensive unit test coverage for all lines and branches of code and has undergone protocol testing, and penetration testing by AWS Security to reduce the exposure to security vulnerabilities. Additionally, the source files have been moved to a `source` directory. This change requires modification of any existing project(s) to include the modified source files and directories. There are examples on how to use the new files and directory structure. For an example based on the Xilinx Zynq-7000, use the code in this [branch](#) and follow these [instructions](#) to build and run the demo.

FreeRTOS-Plus-TCP V3.1.0 source code(.c .h) is part of the FreeRTOS 20210.00 LTS release.

Generating pre V3.0.0 folder structure for backward compatibility: If you wish to continue using a version earlier than V3.0.0 i.e. continue to use your existing source code organization, a script is provided to generate the folder structure similar to [this](#).

Note: After running the script, while the `.c` files will have same names as the pre V3.0.0 source, the files in the `include` directory will have different names and the number of files will differ as well. This should, however, not pose any problems to most projects as projects generally include all files in a given directory.

Running the script to generate pre V3.0.0 folder structure: For running the script, you will need Python version > 3.7. You can download/install it from [here](#).

Once python is downloaded and installed, you can verify the version from your terminal/command window by typing `python --version`.

To run the script, you should switch to the FreeRTOS-Plus-TCP directory that was created using the *Cloning this repository* step above. And then run `python <Path/to/the/script>/GenerateOriginalFiles.py`.

To consume FreeRTOS+TCP

Consume with CMake If using CMake, it is recommended to use this repository using FetchContent. Add the following into your project's main or a subdirectory's CMakeLists.txt:

- Define the source and version/tag you want to use:

```
FetchContent_Declare( freertos_plus_tcp
  GIT_REPOSITORY https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git
  GIT_TAG        master #Note: Best practice to use specific git-hash or tagged version
  GIT_SUBMODULES "" # Don't grab any submodules since not latest
)
```

- Configure the FreeRTOS-Kernel and make it available
 - this particular example supports a native and cross-compiled build option.

```

set( FREERTOS_PLUS_FAT_DEV_SUPPORT OFF CACHE BOOL "" FORCE)
# Select the native compile PORT
set( FREERTOS_PLUS_FAT_PORT "POSIX" CACHE STRING "" FORCE)
# Select the cross-compile PORT
if (CMAKE_CROSSCOMPILING)
  # Eg. Zynq 2019_3 version of port
  set(FREERTOS_PLUS_FAT_PORT "ZYNQ_2019_3" CACHE STRING "" FORCE)
endif()

FetchContent_MakeAvailable(freertos_plus_tcp)

```

Consuming stand-alone This repository uses [Git Submodules](#) to bring in dependent components.

Note: If you download the ZIP file provided by GitHub UI, you will not get the contents of the submodules. (The ZIP file is also not a valid Git repository)

To clone using HTTPS:

```

git clone https://github.com/FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Using SSH:

```

git clone git@github.com:FreeRTOS/FreeRTOS-Plus-TCP.git ./FreeRTOS-Plus-TCP
cd ./FreeRTOS-Plus-TCP
git submodule update --checkout --init --recursive tools/CMock test/FreeRTOS-Kernel

```

Porting The porting guide is available on [this page](#).

Repository structure This repository contains the FreeRTOS-Plus-TCP repository and a number of supplementary libraries for testing/PR Checks. Below is the breakdown of what each directory contains:

- tools
 - This directory contains the tools and related files (CMock/uncrustify) required to run tests/checks on the TCP source code.
- tests
 - This directory contains all the tests (unit tests and CBMC) and the dependencies ([FreeRTOS-Kernel/Litani-port](#)) the tests require.
- source/portable
 - This directory contains the portable files required to compile the FreeRTOS-Plus-TCP source code for different hardware/compilers.
- source/include
 - The include directory has all the ‘core’ header files of FreeRTOS-Plus-TCP source.
- source
 - This directory contains all the [.c] source files.

Note At this time it is recommended to use `BufferAllocation_2.c` in which case it is essential to use the `heap_4.c` memory allocation scheme. See [memory management](#).

Kernel sources The FreeRTOS Kernel Source is in [FreeRTOS/FreeRTOS-Kernel repository](#), and it is consumed by testing/PR checks as a submodule in this repository.

The version of the FreeRTOS Kernel Source in use could be accessed at `./test/FreeRTOS-Kernel` directory.

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

To learn more about CBMC and proofs specifically, review the training material [here](#).

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