



# MCUXpresso SDK Documentation

Release 26.03.00-pvw2



NXP  
Feb 20, 2026



# Table of contents

<b>1</b>	<b>LPCXpresso802</b>	<b>3</b>
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	59
1.3.1	Getting Started with MCUXpresso SDK Repository	59
1.4	Release Notes	66
1.4.1	MCUXpresso SDK Release Notes	66
1.5	ChangeLog	68
1.5.1	MCUXpresso SDK Changelog	68
1.6	Driver API Reference Manual	93
1.7	Middleware Documentation	93
1.7.1	FreeMASTER	93
<b>2</b>	<b>LPC802</b>	<b>95</b>
2.1	Clock Driver	95
2.2	CRC: Cyclic Redundancy Check Driver	102
2.3	CTIMER: Standard counter/timers	105
2.4	I2C: Inter-Integrated Circuit Driver	115
2.5	I2C Driver	115
2.6	I2C Master Driver	116
2.7	I2C Slave Driver	125
2.8	IAP: In Application Programming Driver	134
2.9	Common Driver	140
2.10	LPC_ACOMP: Analog comparator Driver	154
2.11	ADC: 12-bit SAR Analog-to-Digital Converter Driver	157
2.12	GPIO: General Purpose I/O	168
2.13	IOCON: I/O pin configuration	170
2.14	MRT: Multi-Rate Timer	171
2.15	PINT: Pin Interrupt and Pattern Match Driver	175
2.16	Power Driver	184
2.17	Reset Driver	188
2.18	SPI: Serial Peripheral Interface Driver	191
2.19	SPI Driver	191
2.20	SWM: Switch Matrix Module	203
2.21	SYSCON: System Configuration	207
2.22	USART: Universal Asynchronous Receiver/Transmitter Driver	208
2.23	USART Driver	208
2.24	WKT: Self-wake-up Timer	221
2.25	WWDT: Windowed Watchdog Timer Driver	223
<b>3</b>	<b>Middleware</b>	<b>227</b>
3.1	Motor Control	227
3.1.1	FreeMASTER	227
<b>4</b>	<b>RTOS</b>	<b>265</b>
4.1	FreeRTOS	265

4.1.1	FreeRTOS kernel	265
4.1.2	FreeRTOS drivers	265
4.1.3	backoffalgorithm	265
4.1.4	corehttp	265
4.1.5	corejson	265
4.1.6	coremqtt	266
4.1.7	corepkcs11	266
4.1.8	freertos-plus-tcp	266

This documentation contains information specific to the lpcxpresso802 board.

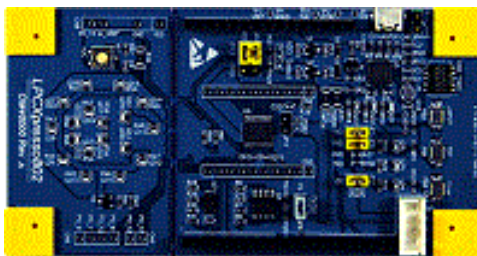


# Chapter 1

## LPCXpresso802

### 1.1 Overview

LPC800 series boards and devices are fully supported by NXP's MCUXpresso suite of free software and tools, which include an Eclipse-based IDE, configuration tools and extensive SDK drivers/examples available at <https://mcuxpresso.nxp.com>. All boards in this series include an on-board CMSIS-DAP debug probe based on the LPC11U35 debug probe, with the option for an external debug probe such as those from SEGGER and PE Micro. Popular Arduino UNO shield boards can be used on these boards, enabling quick and easy prototyping. The LPC800 series is fully supported by NXP's MCUXpresso suite <<https://www.nxp.com/mcuxpresso>> 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, and these IDEs are also fully supported by the MCUXpresso pin, clock and peripheral configuration tools.



MCU device and part on board is shown below:

- Device: LPC802
- PartNumber: LPC802M001JDH20

### 1.2 Getting Started with MCUXpresso SDK Package

#### 1.2.1 Getting Started with MCUXpresso SDK Package

Starting with version 25.09.00, MCUXpresso SDK introduced two package versions for offline development:

- **Classic SDK Package:** Traditional board-specific packages with pre-configured IDE projects for MCUXpresso IDE, IAR, Keil, and other toolchains.
- **Repository-Layout SDK Package:** Board-specific packages that maintain the same structure and build system as the GitHub Repository SDK, providing offline access to

the repository SDK development experience. Available when selecting the ARMGCC toolchain.

**From version 25.12.00 onward:**

- When you select ARMGCC, the SDK download will use the Repository-Layout version.
- For all other toolchains, the SDK download will remain in the Classic version.

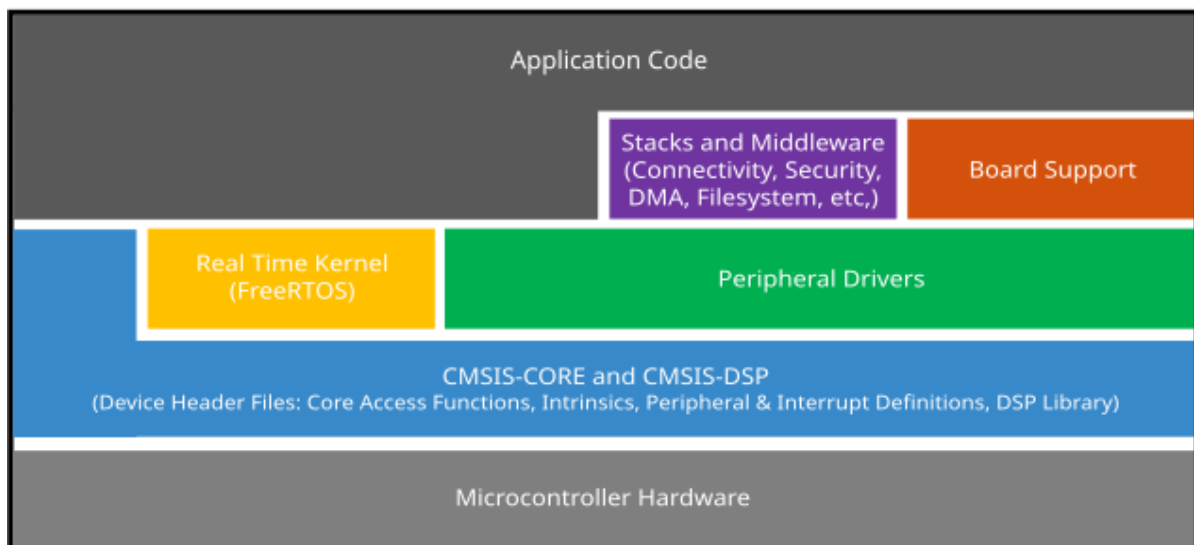
Note: The Repository-Layout SDK package was first introduced in version 25.09.00, but initially only for MCXW23x platforms.

## Classic 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):

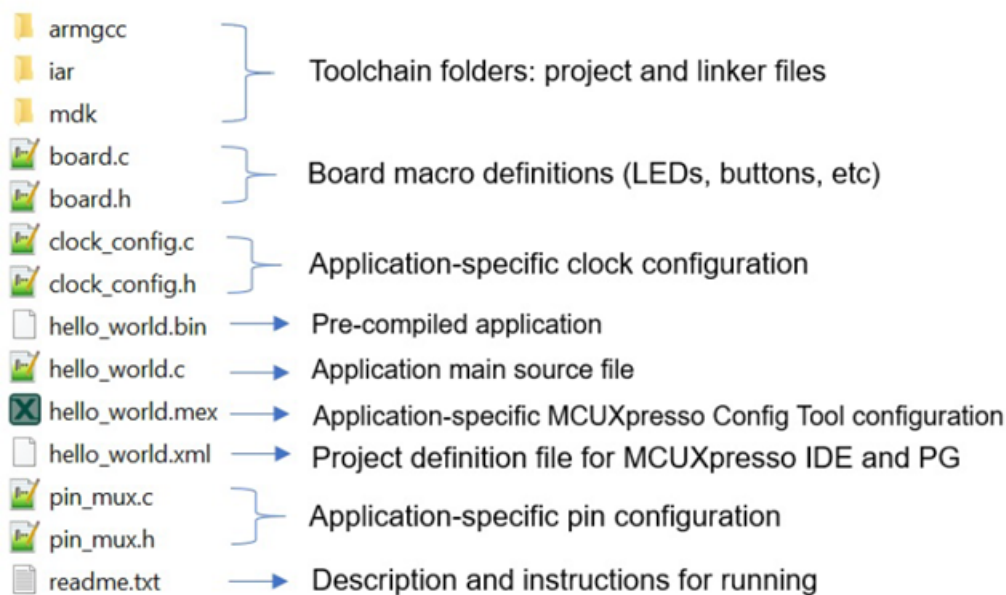
- `cmsis_driver_examples`: Simple applications intended to show how to use CMSIS drivers.
- `demo_apps`: Full-featured applications that highlight key functionality and use cases of the target MCU. These applications typically use multiple MCU peripherals and may leverage stacks and middleware.

- `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.

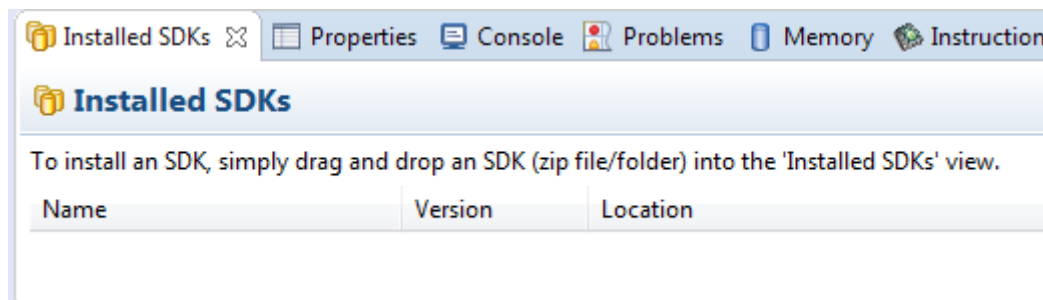
**Run a demo using MCUXpresso IDE** **Note:** Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

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.

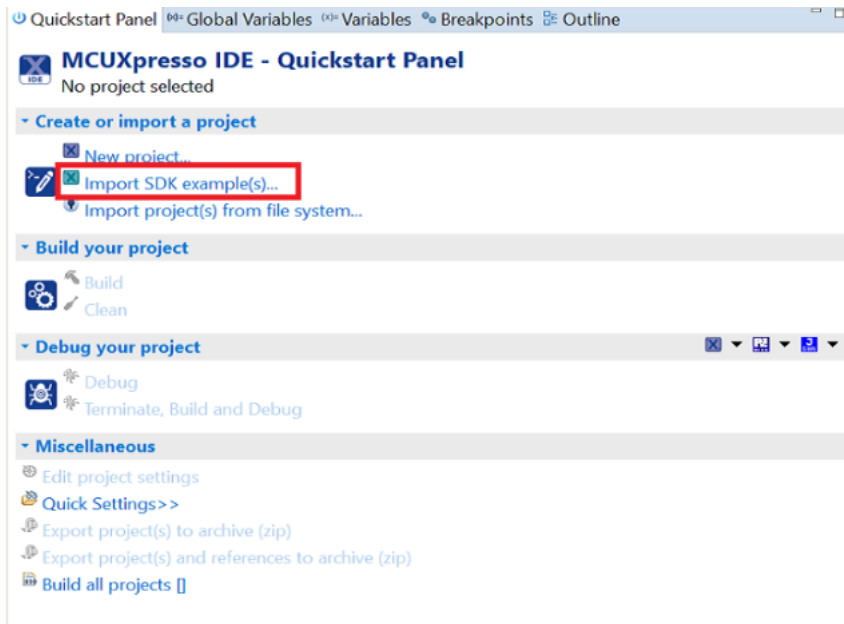
**Select the workspace location** Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the 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.

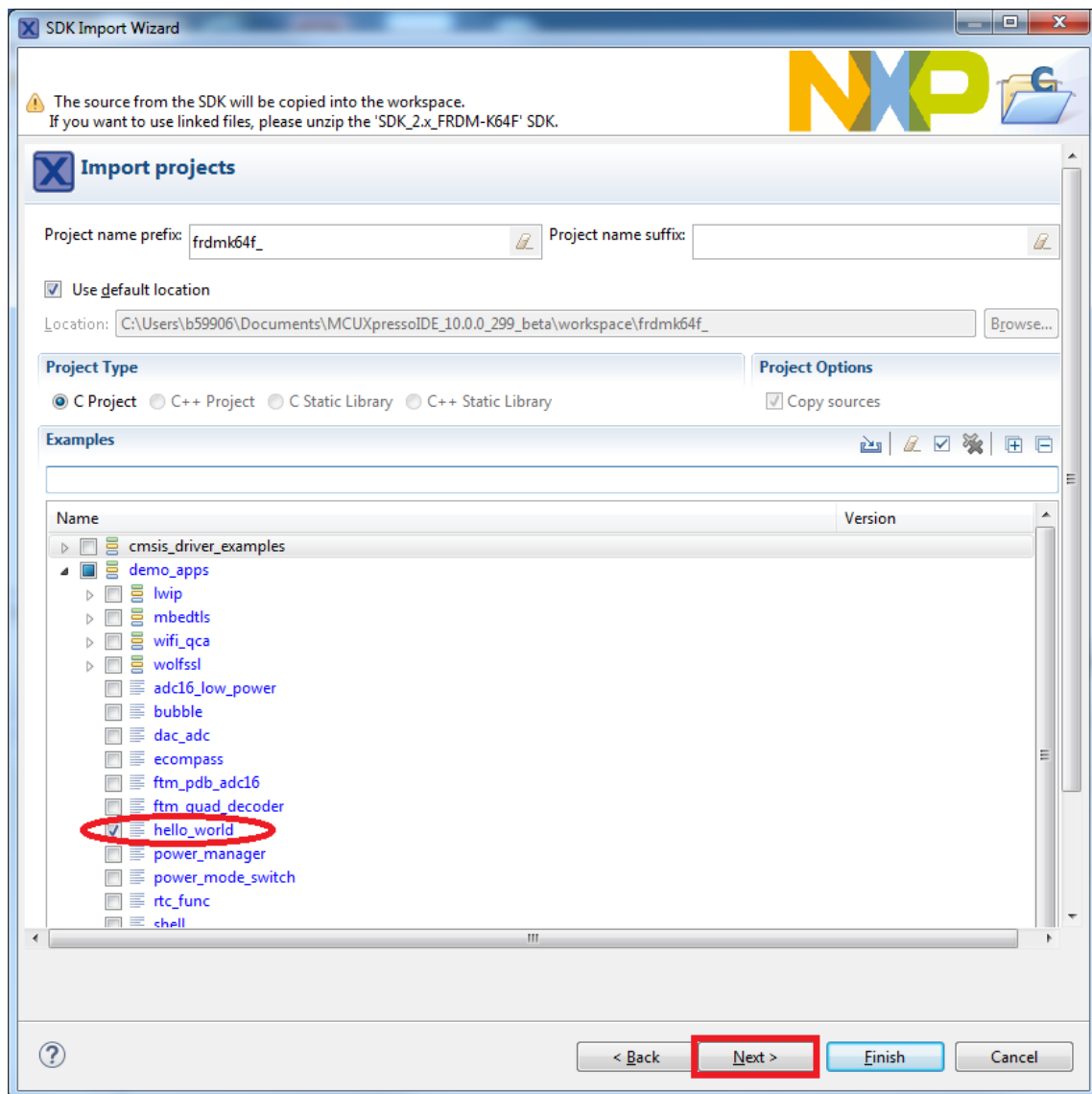
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



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



3. Expand the demo\_apps folder and select hello\_world.
4. Click **Next**.



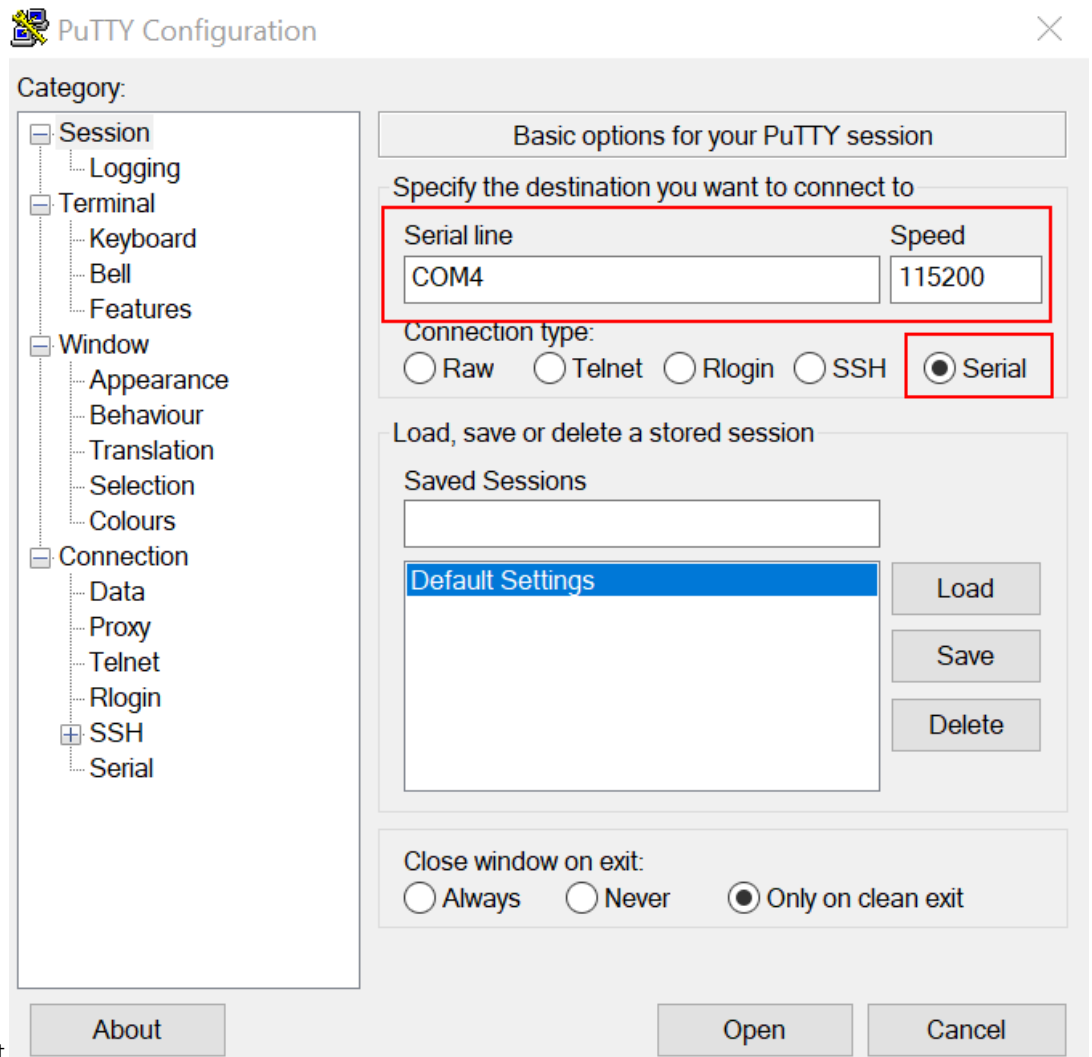
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminal for demo applications such as `adc_basic`, `adc_burst`, `adc_dma`, and `adc_interrupt`. Otherwise, it is not necessary to select this option. Then, click **Finish**.

**Run an example application** For more information on debug probe support in the MCUXpresso IDE, see [community.nxp.com](http://community.nxp.com).

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

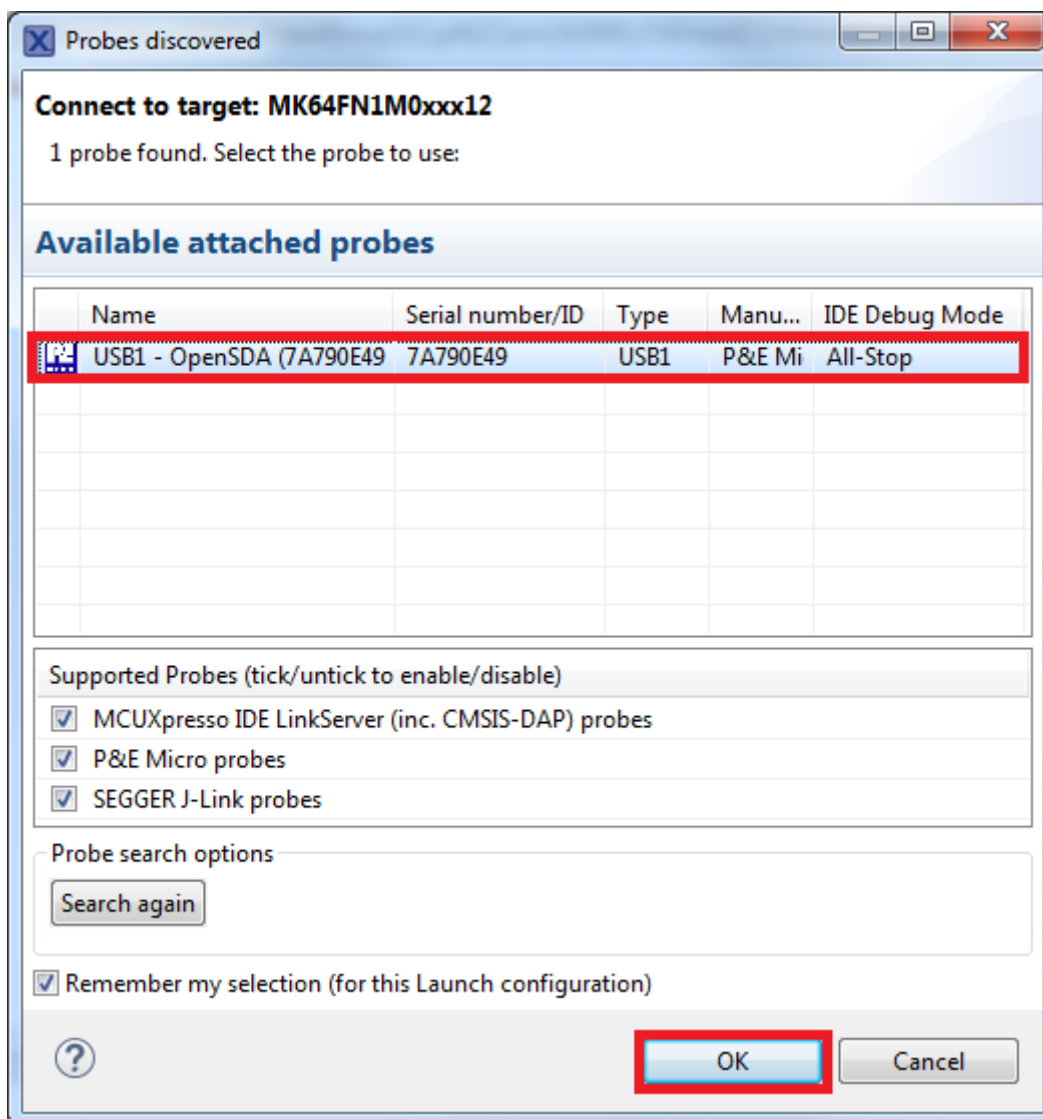
1. Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
2. Connect the development platform to your PC via a USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM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

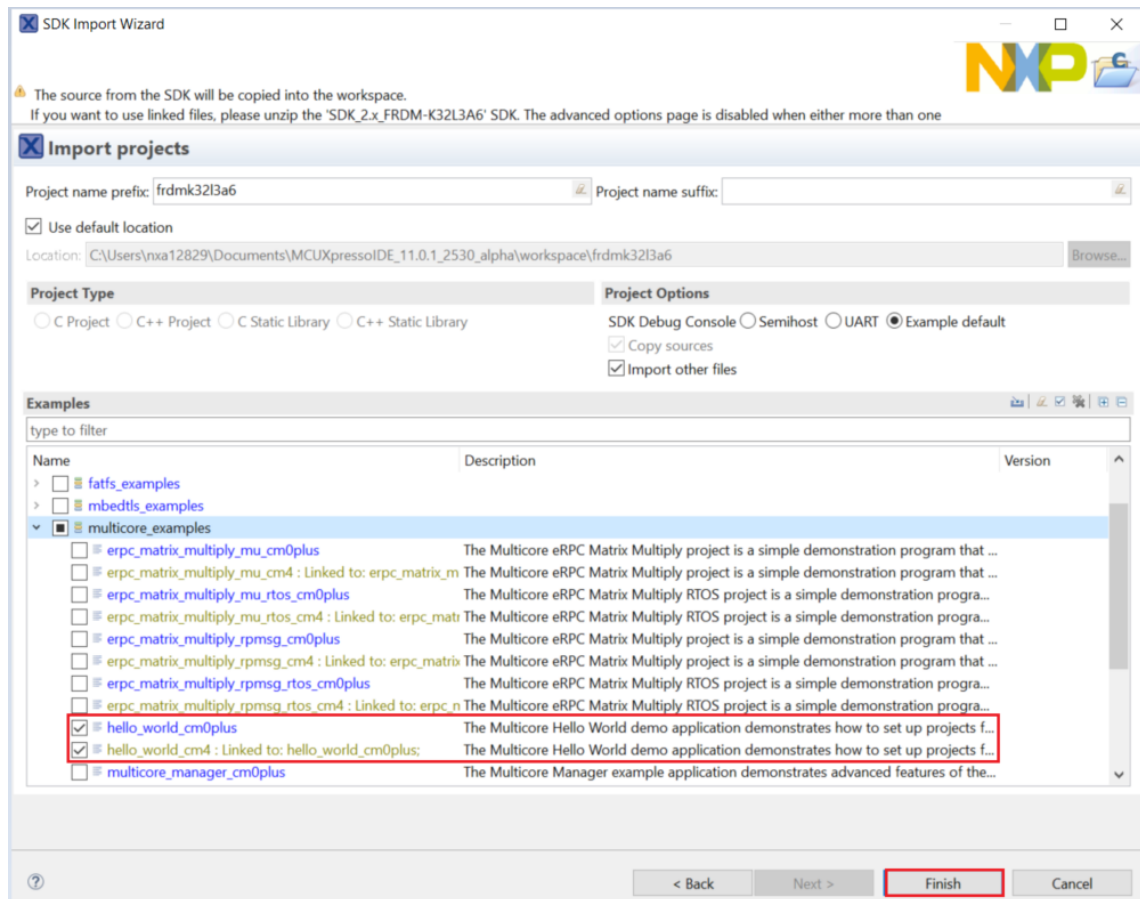


The `hello_world` application is now running and a banner is displayed on the terminal. If not, check your terminal settings and connections.

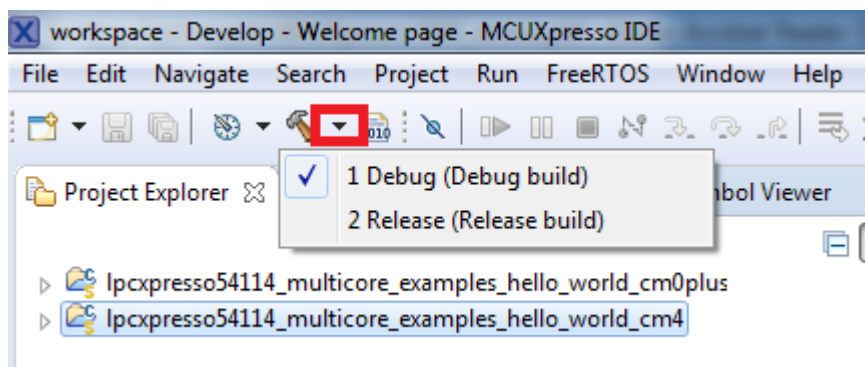


**Build a multicore example application** This section describes the steps required to configure MCUXpresso IDE to build, run, and debug multicore example applications. The following steps can be applied to any multicore example application in the MCUXpresso SDK. Here, the dual-core version of hello\_world 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**.

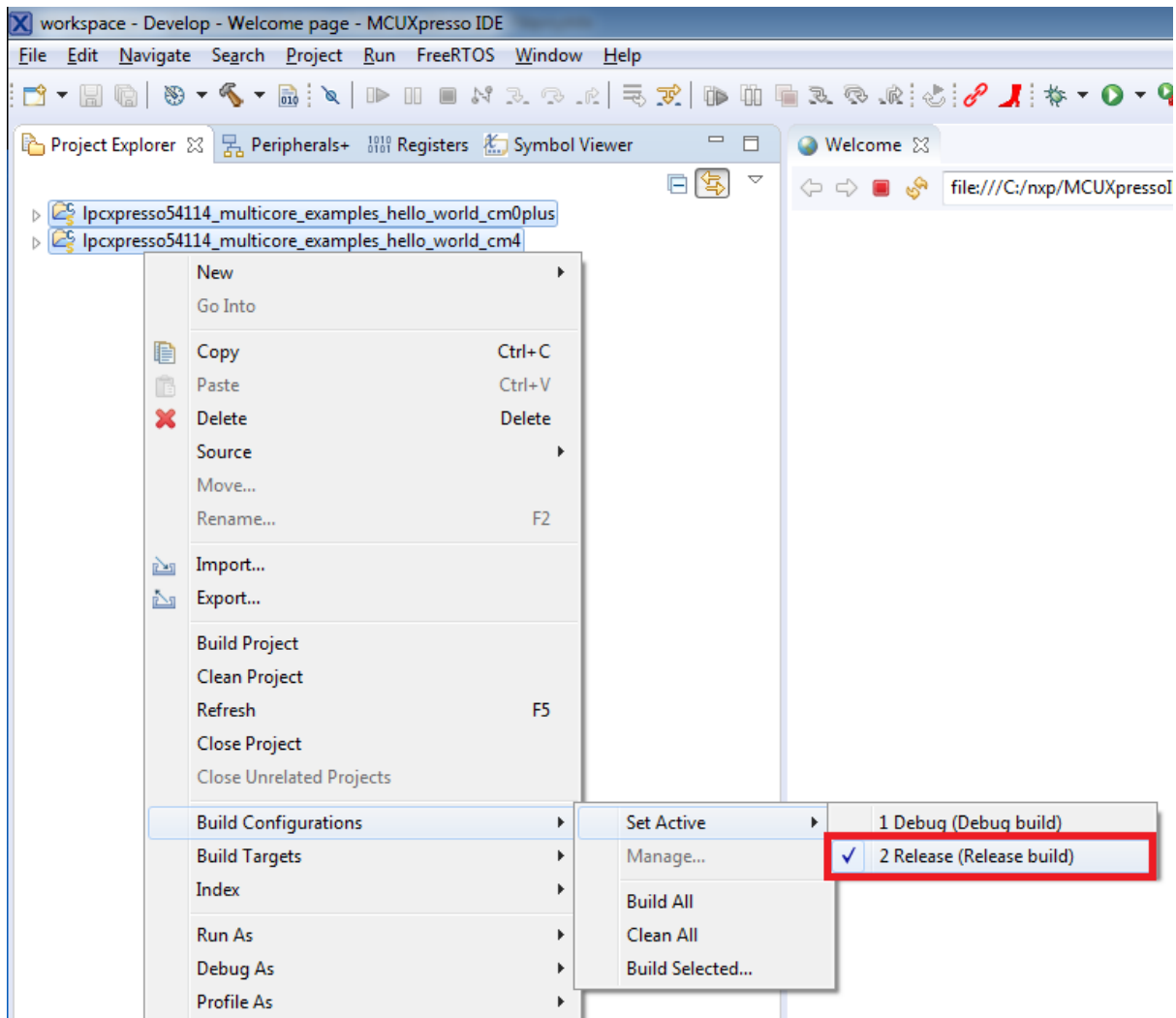


- Now, two projects should be imported into the workspace. To start building the multicore application, highlight the `lpcxpresso54114_multicore_examples_hello_world_cm4` project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by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

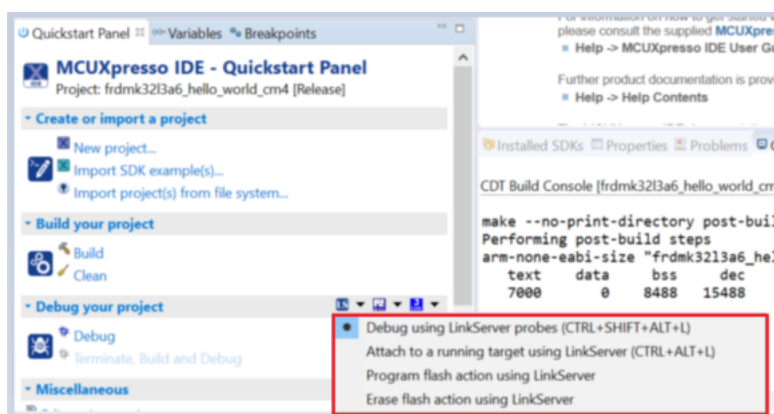


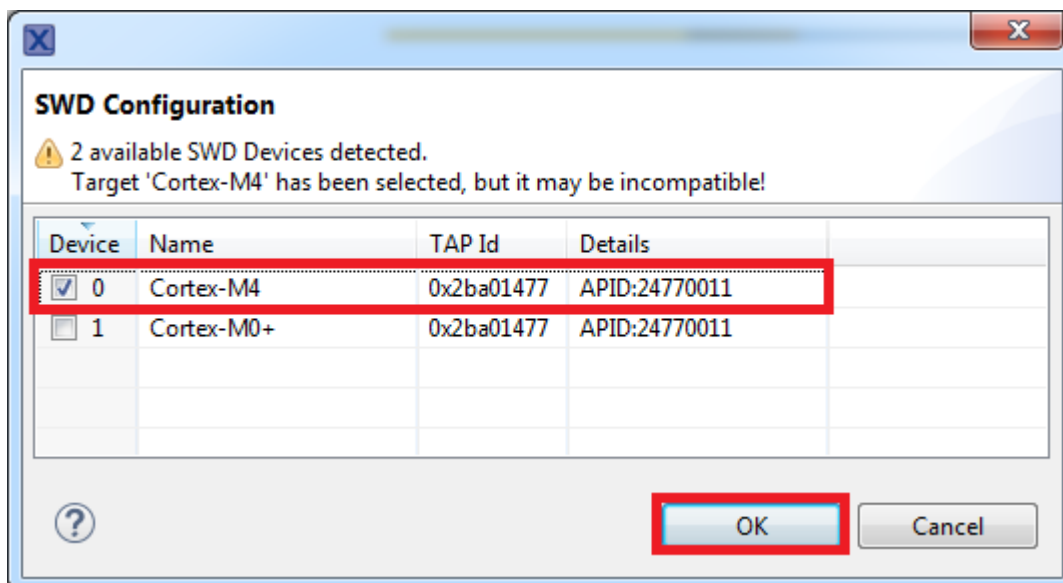
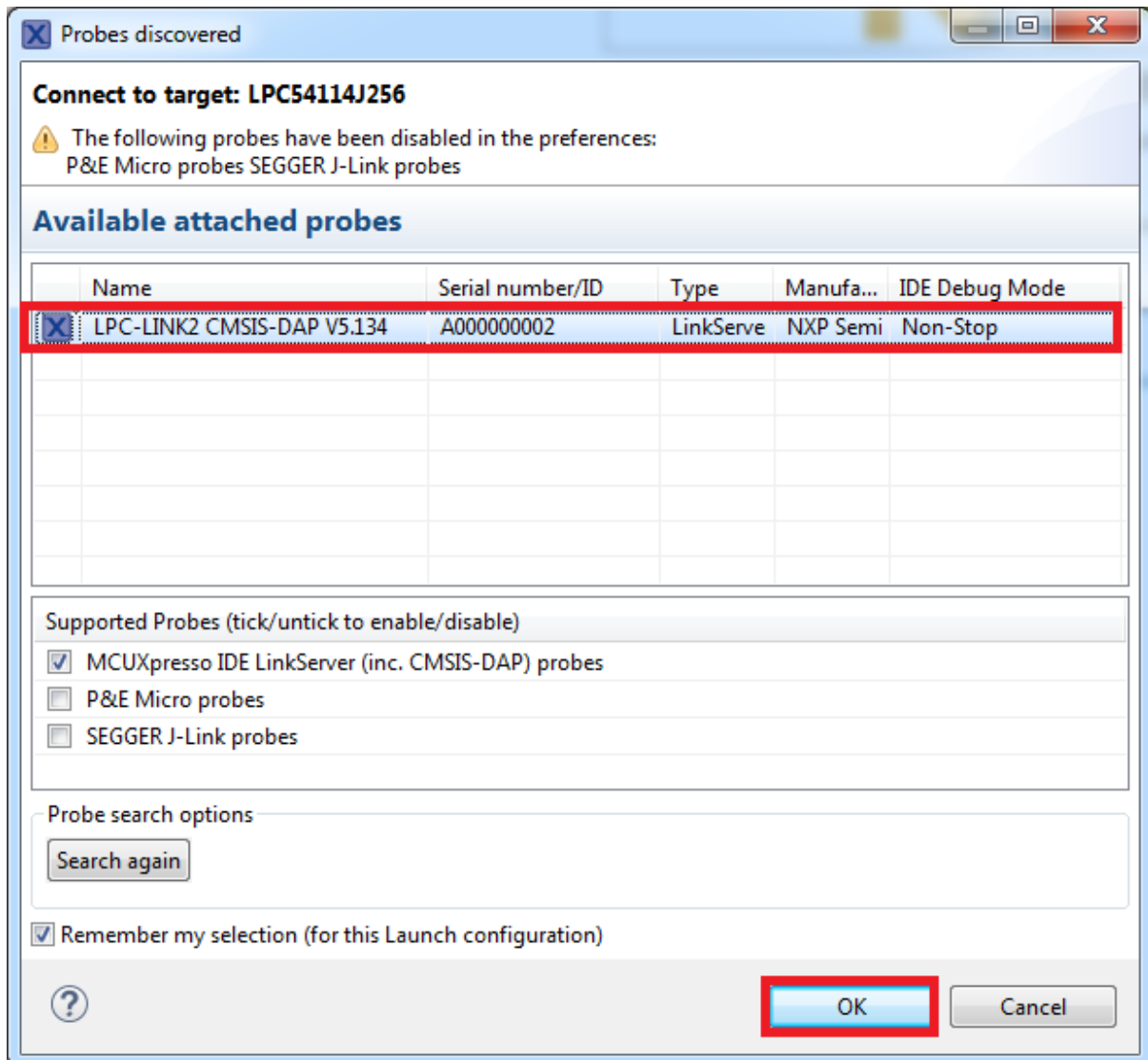
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

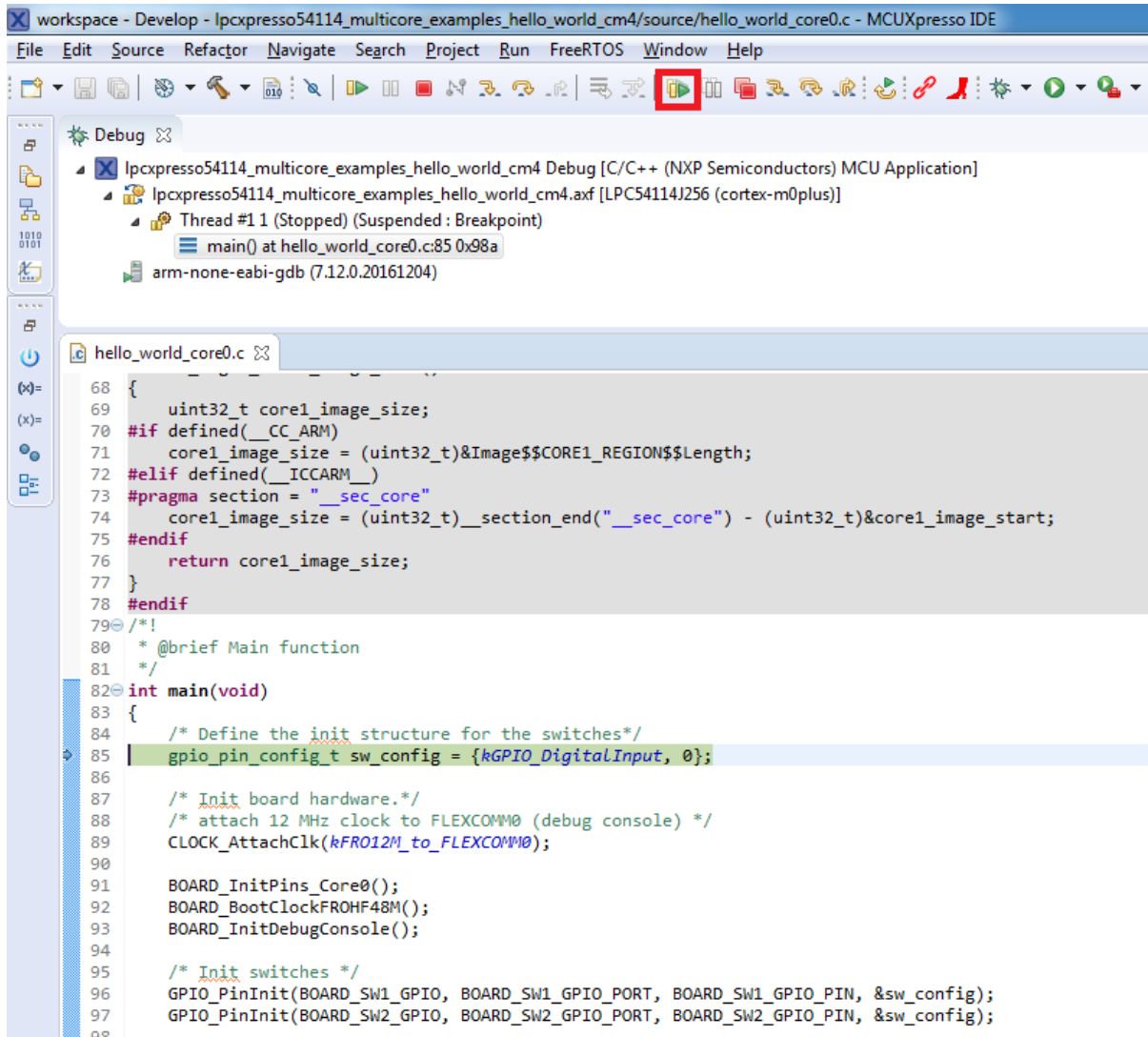
**Note:** When the **Release** build is requested, it is necessary to change the build configuration of both the primary and auxiliary core application projects first. To do this, select both projects in the Project Explorer view and then right click which displays the context-sensitive menu. Select **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.



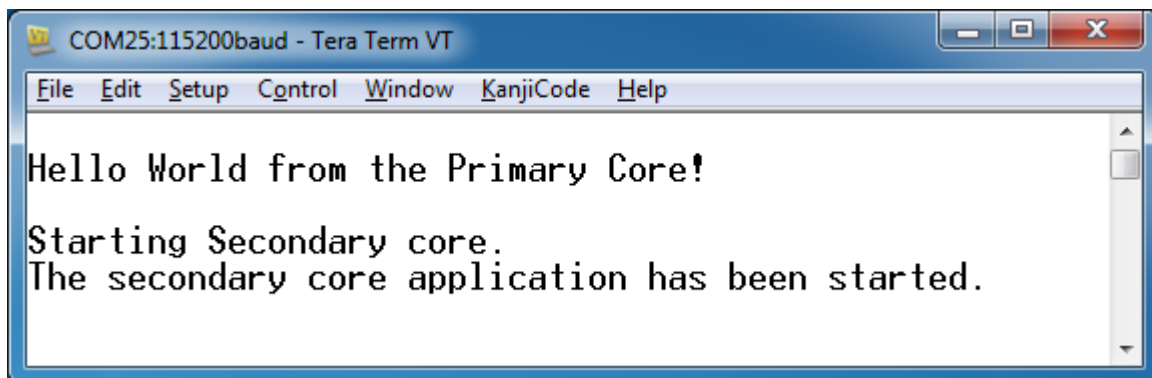
**Run a multicore example application** The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run 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.





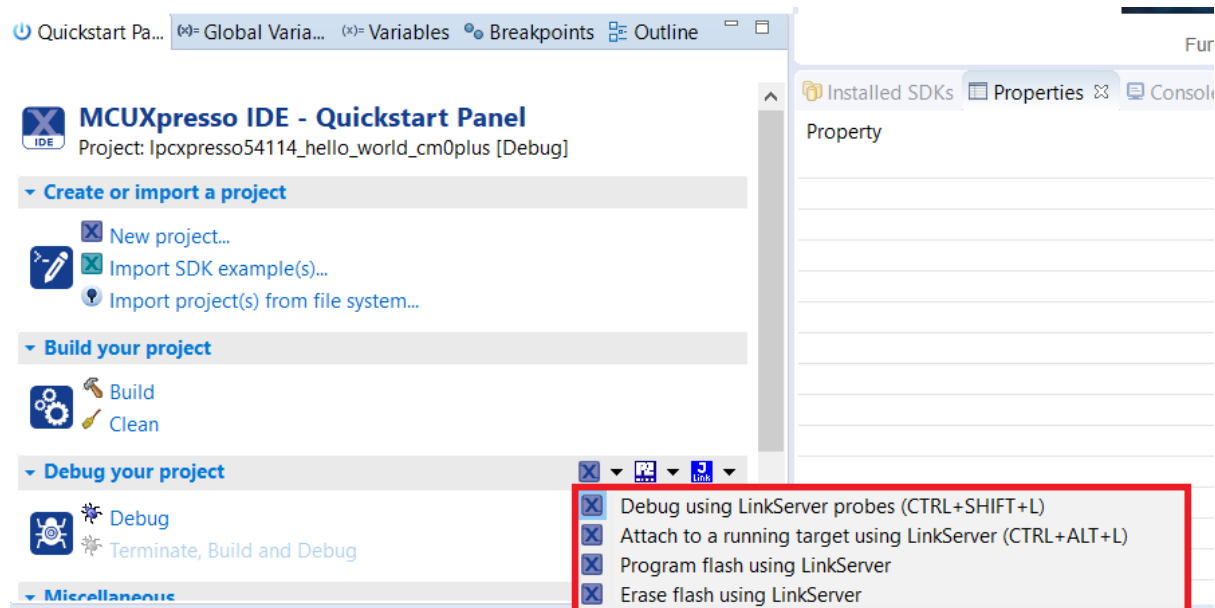


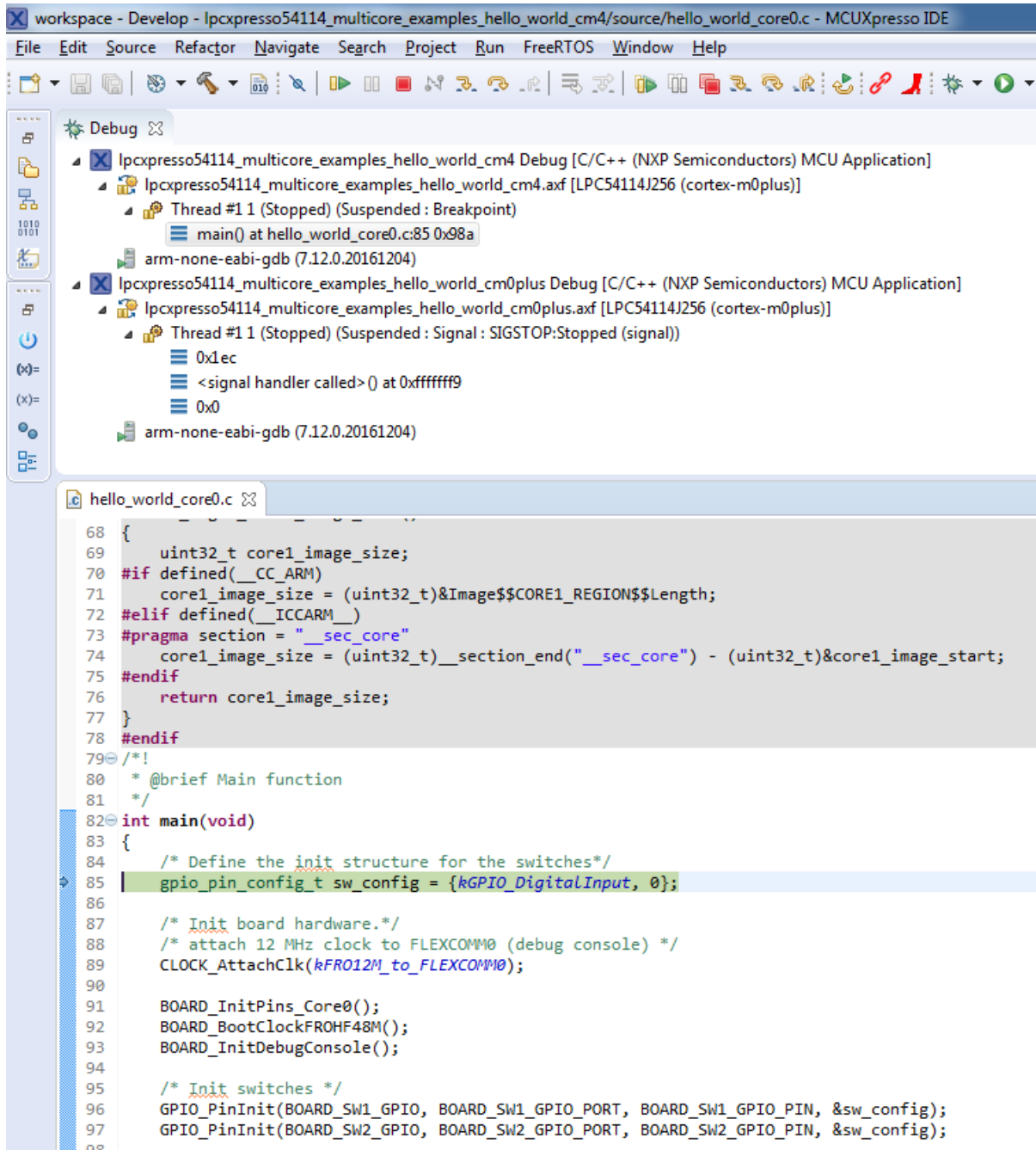
After clicking the “Resume All Debug sessions” button, the `hello_world` multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.



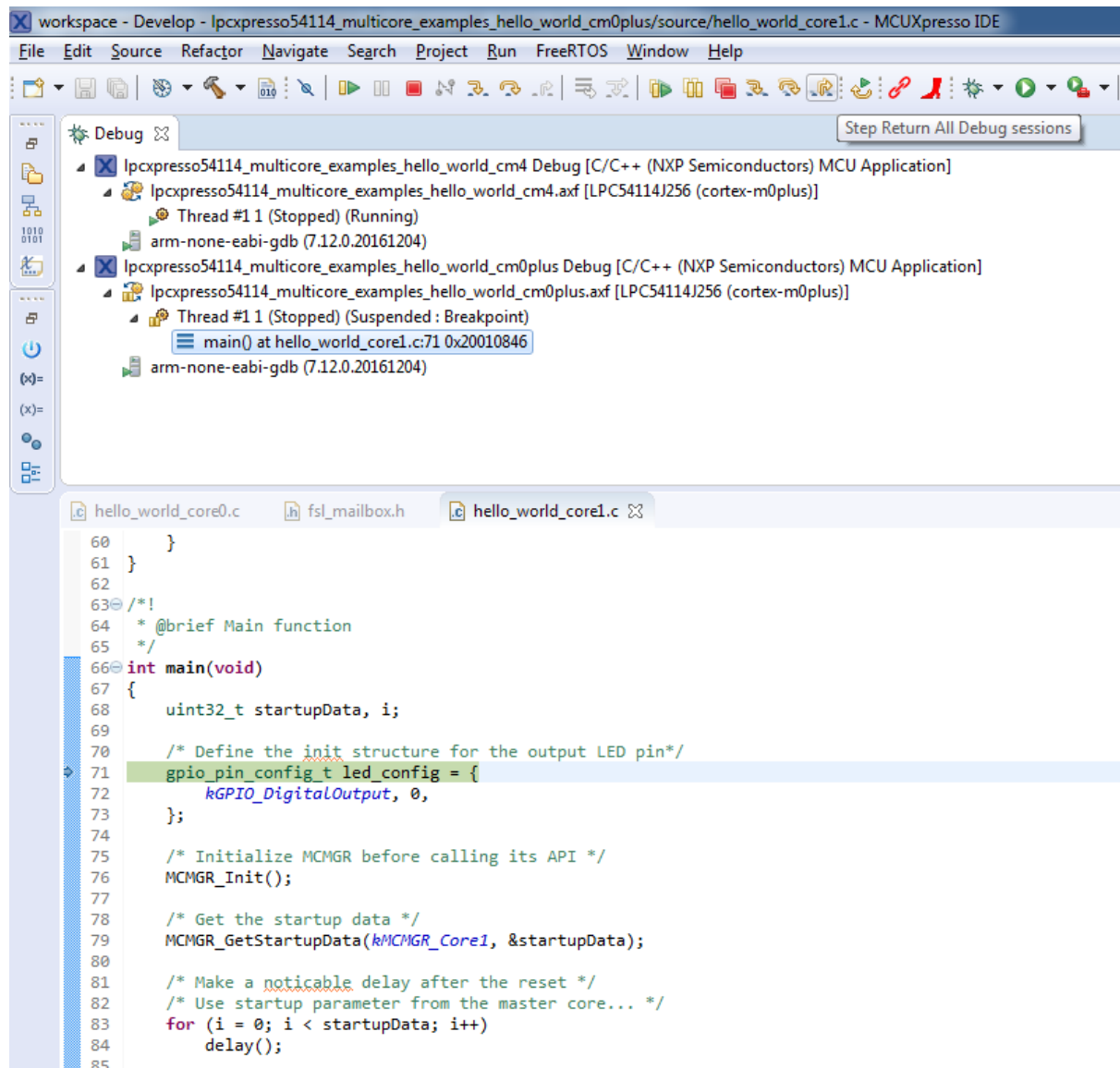
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps 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.

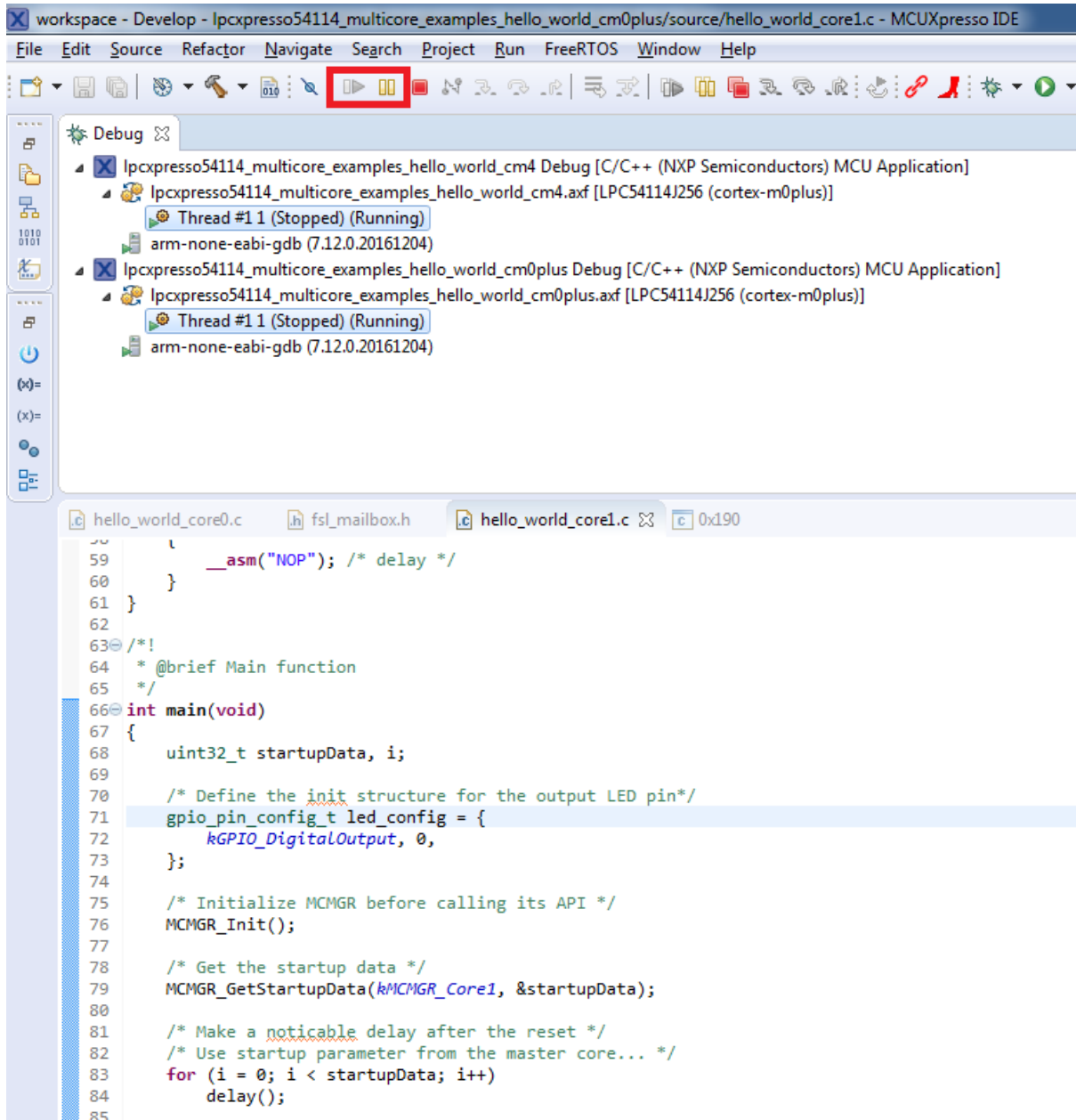


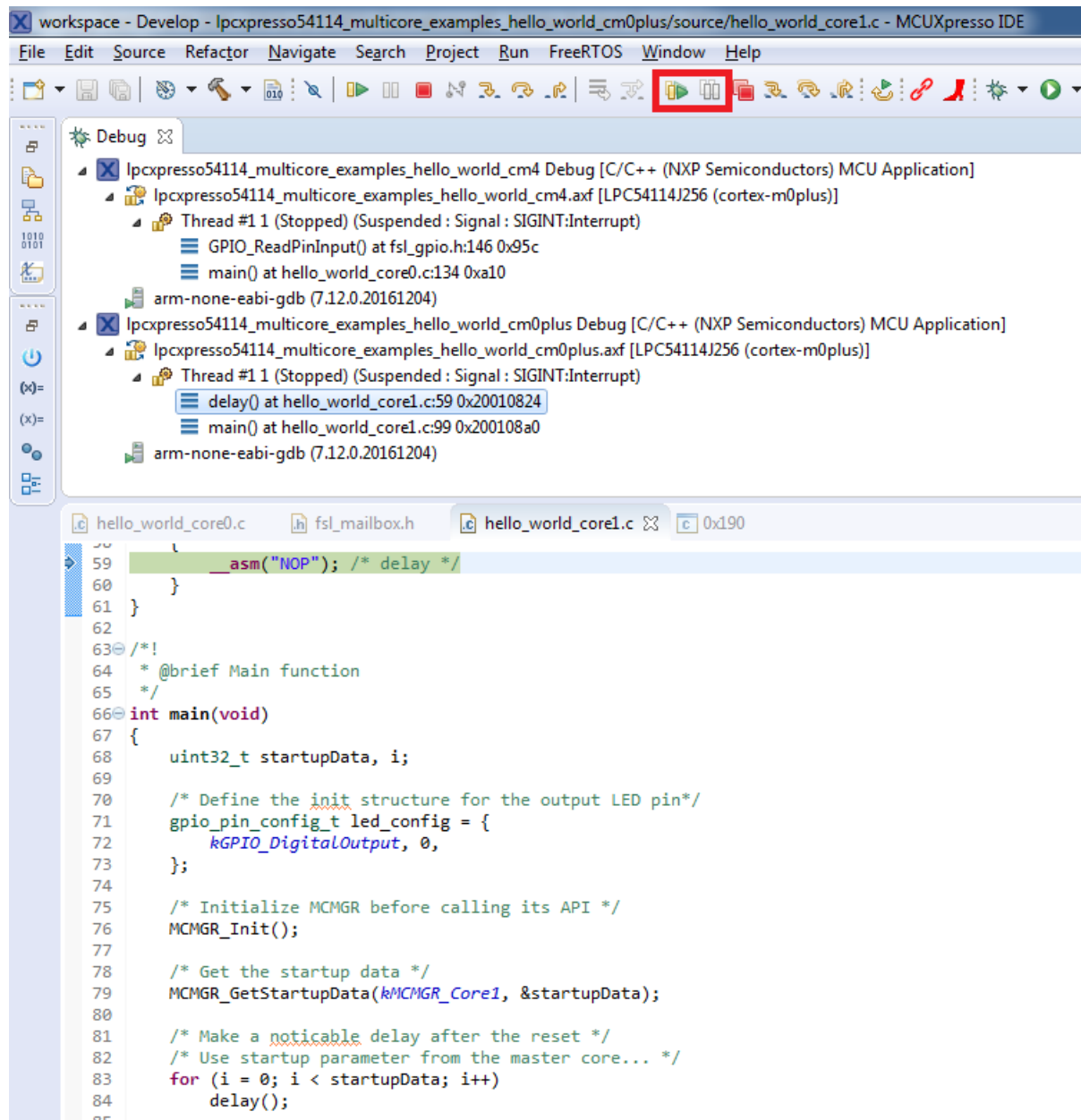


Now, the two debug sessions should be opened, and the debug controls can be used for both debug sessions depending on the debug session selection. Keep the primary core debug session selected by clicking the “Resume” button. The hello\_world multicore application then starts running. The primary core 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.



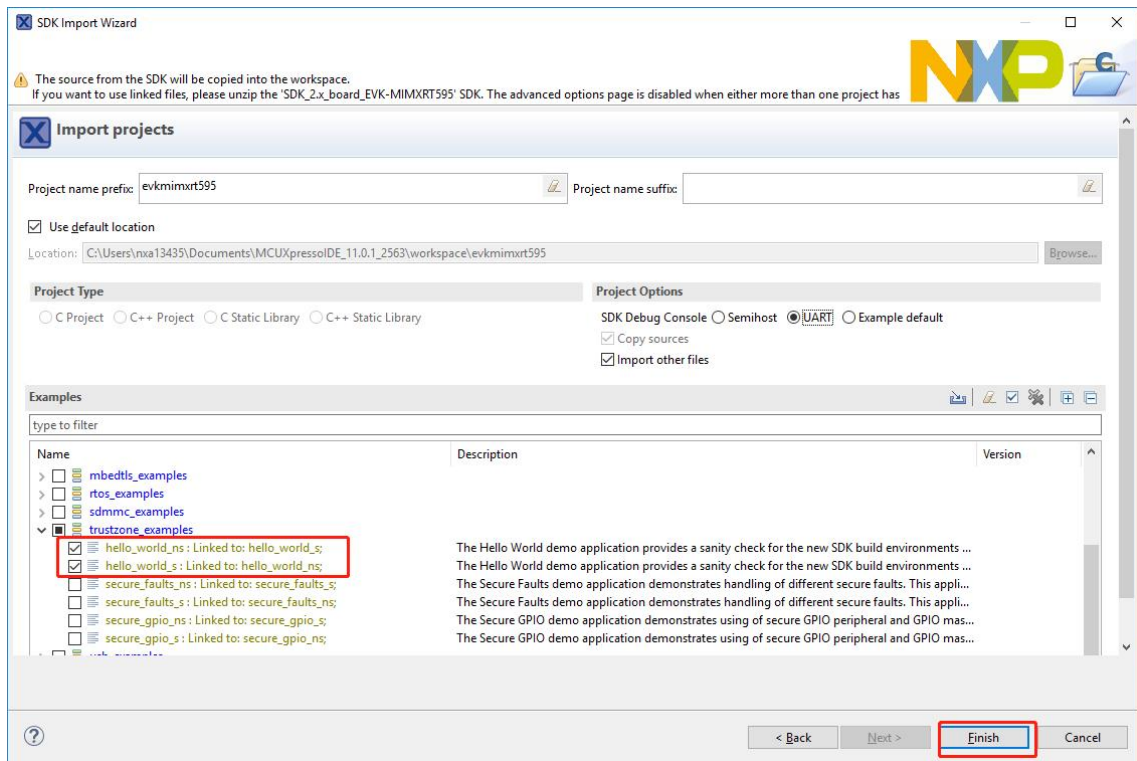
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the “Suspend” / “Resume” control button,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



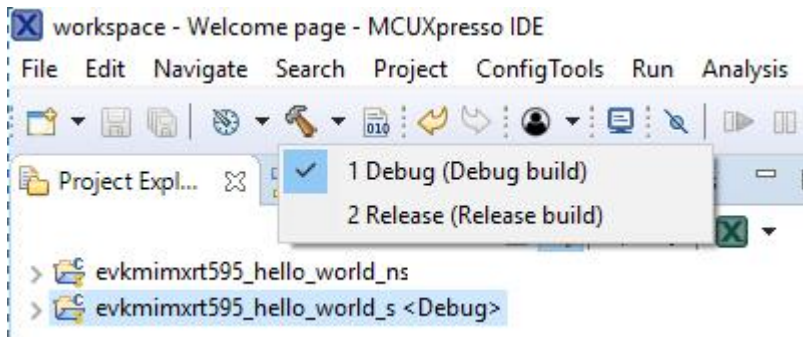


**Build a TrustZone example application** This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the `hello_world` example application targeted for the MIMXRT595-EVK hardware platform is used as an example,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

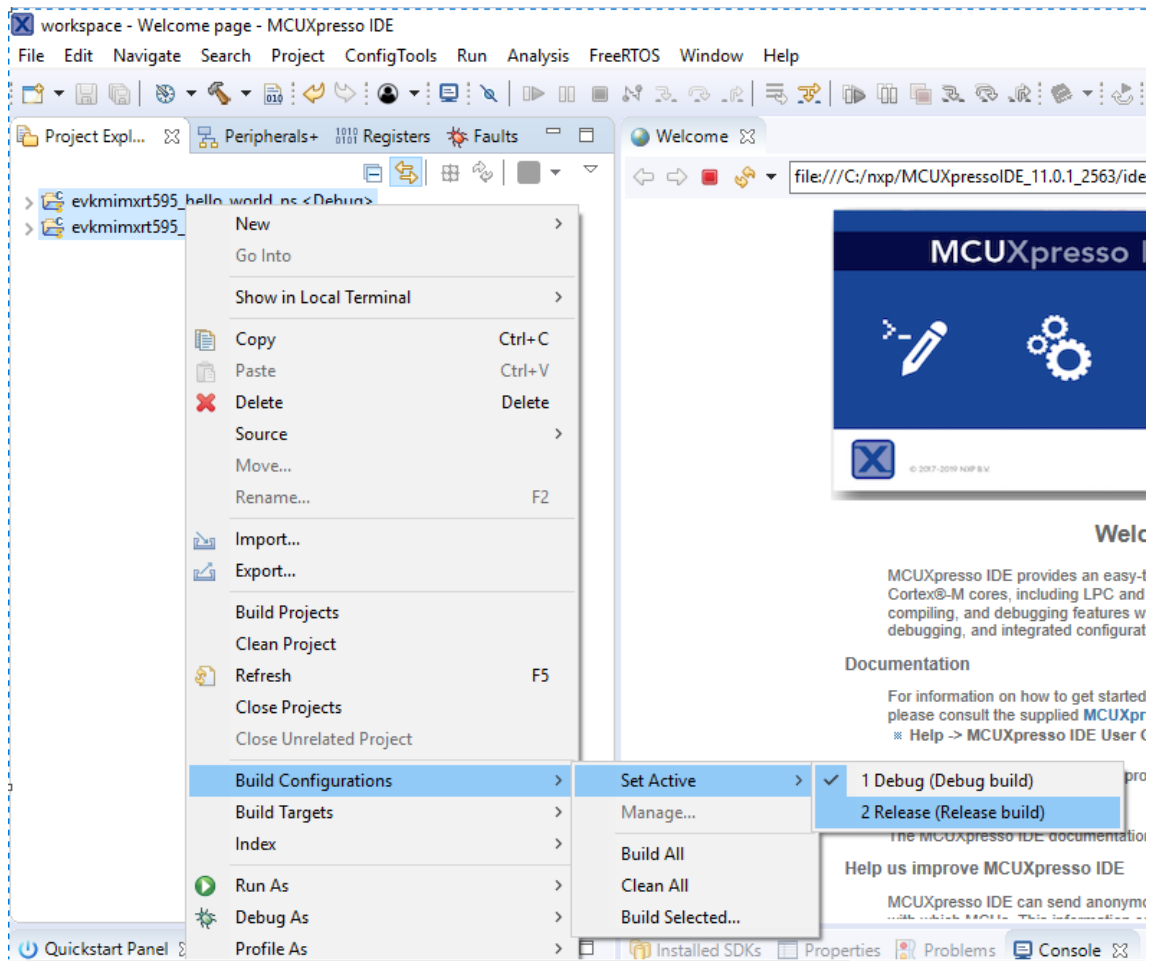


3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the `evkmimxrt595_hello_world_s` project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward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



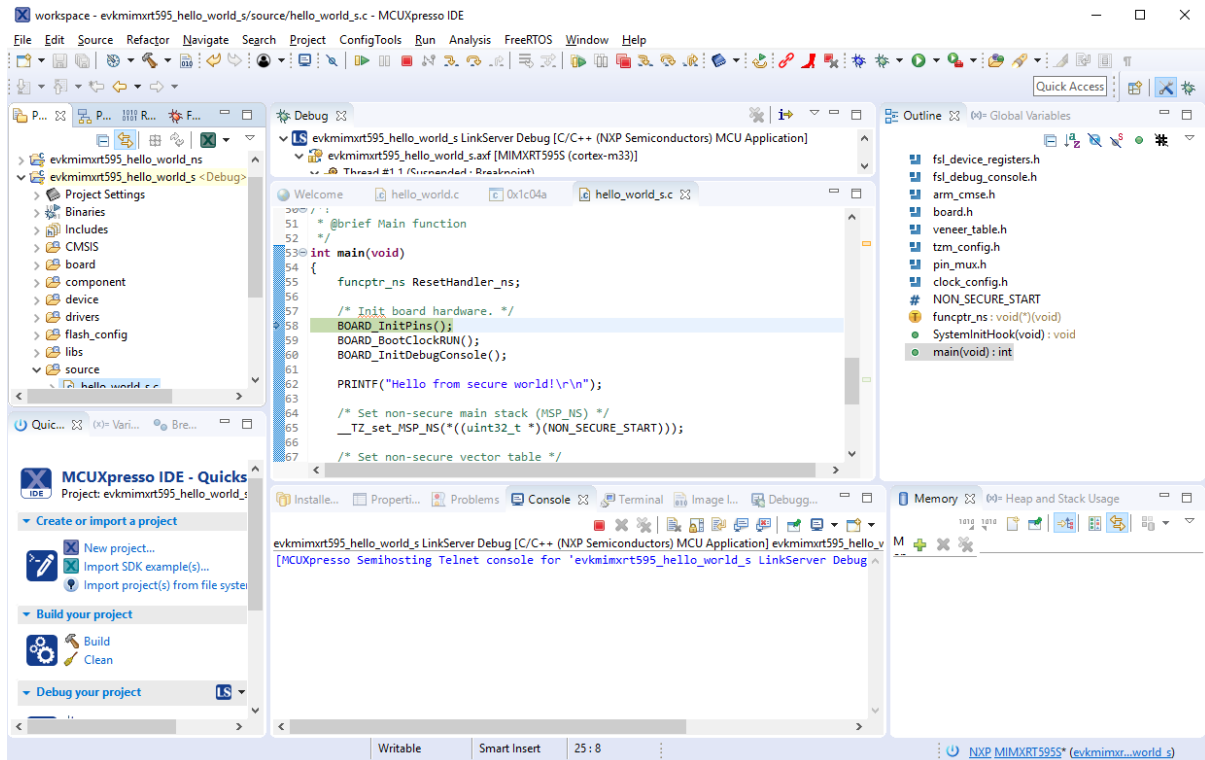
The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when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

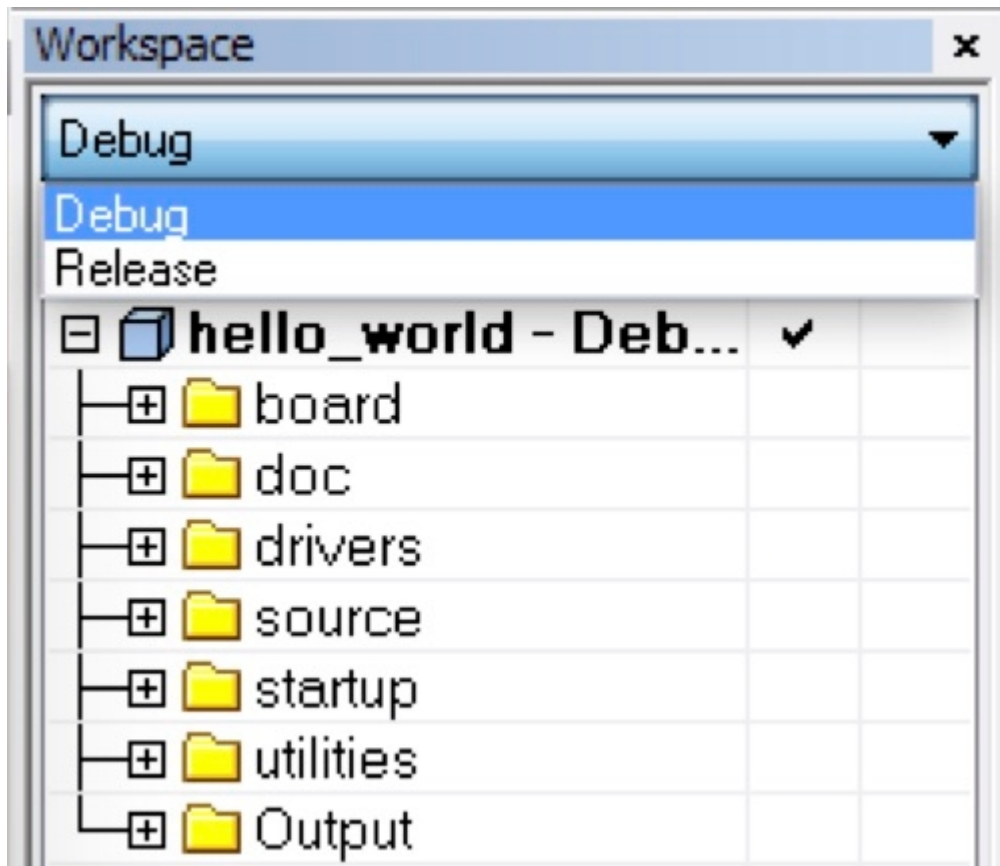
1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

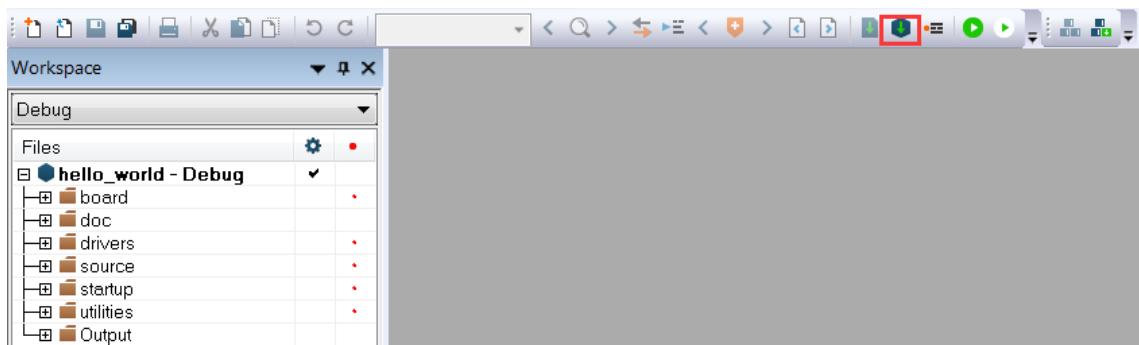
Other example applications may have additional folders in their path.

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

For this example, select **hello\_world – debug**.



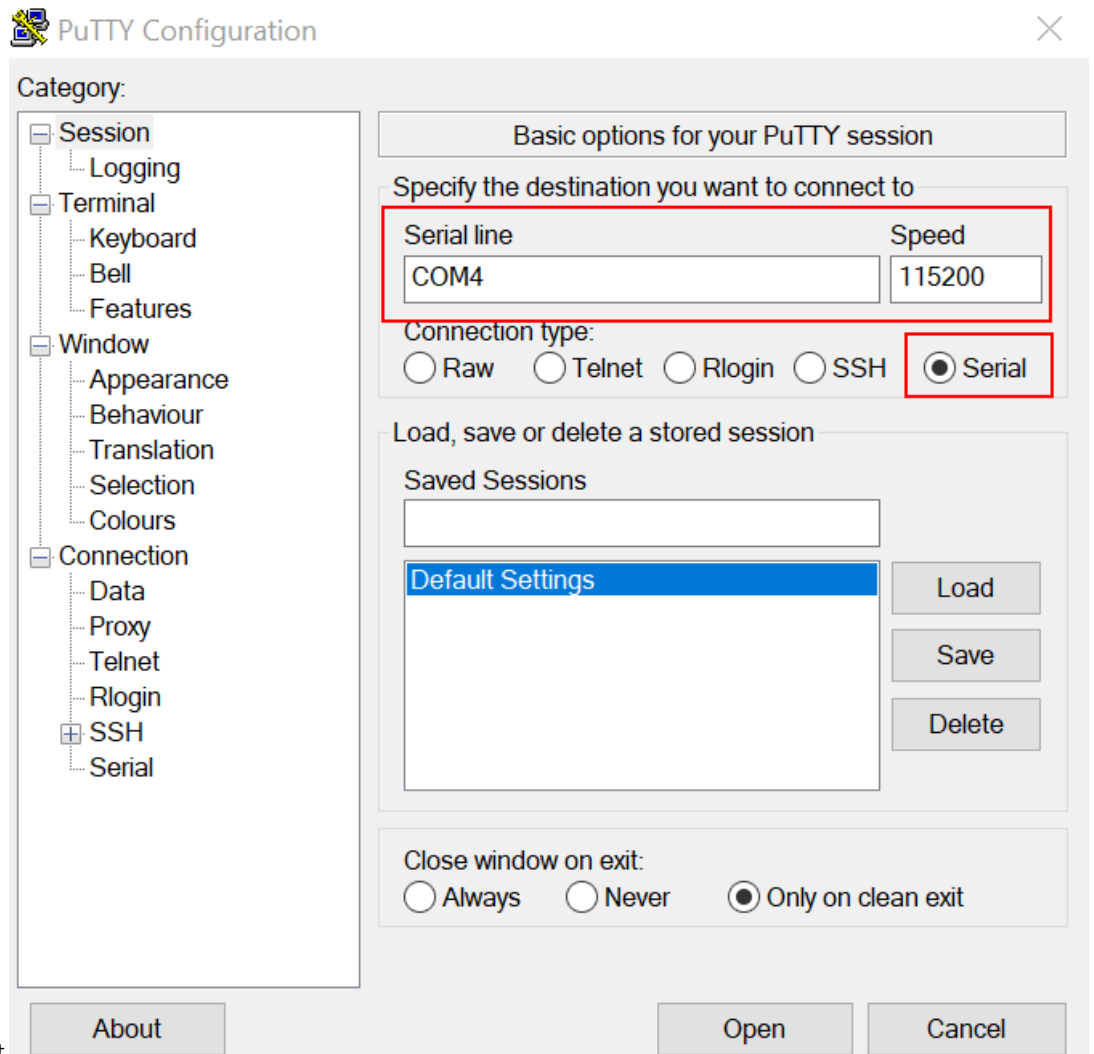
3. To build the demo application, click **Make**, highlighted in red in following figure.



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

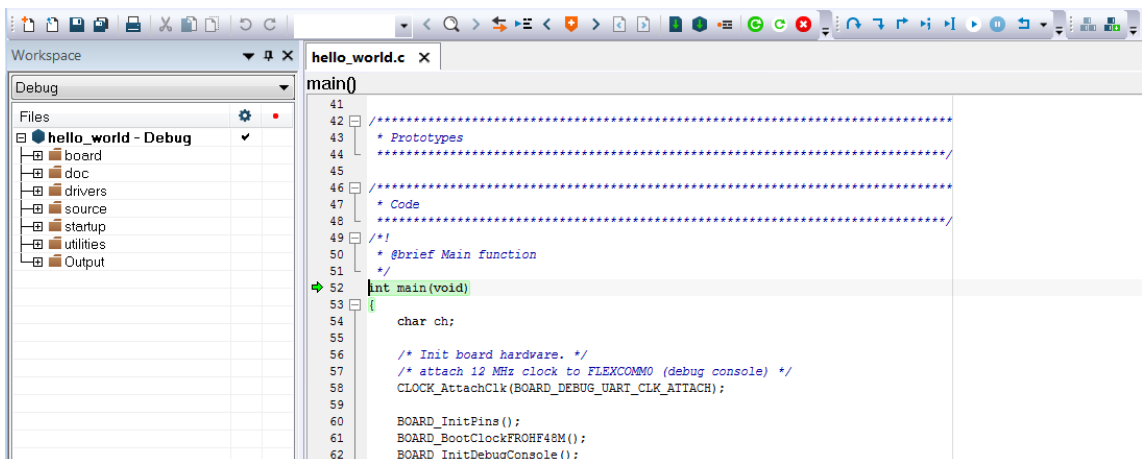


4. 1 stop bit

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.



- 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
```

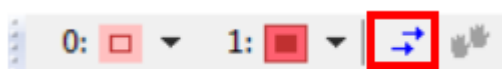
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/iar/hello_world_cm4.eww
```

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.

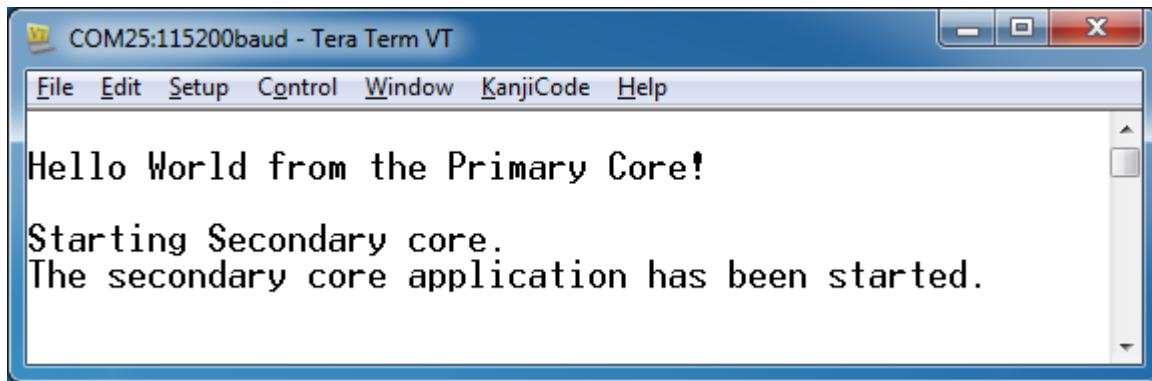
**Run a multicore example application** The primary core debugger handles flashing both primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 4 as described in **Run an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

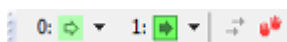
Run both cores by clicking the “Start all cores” button to start the multicore application.



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:

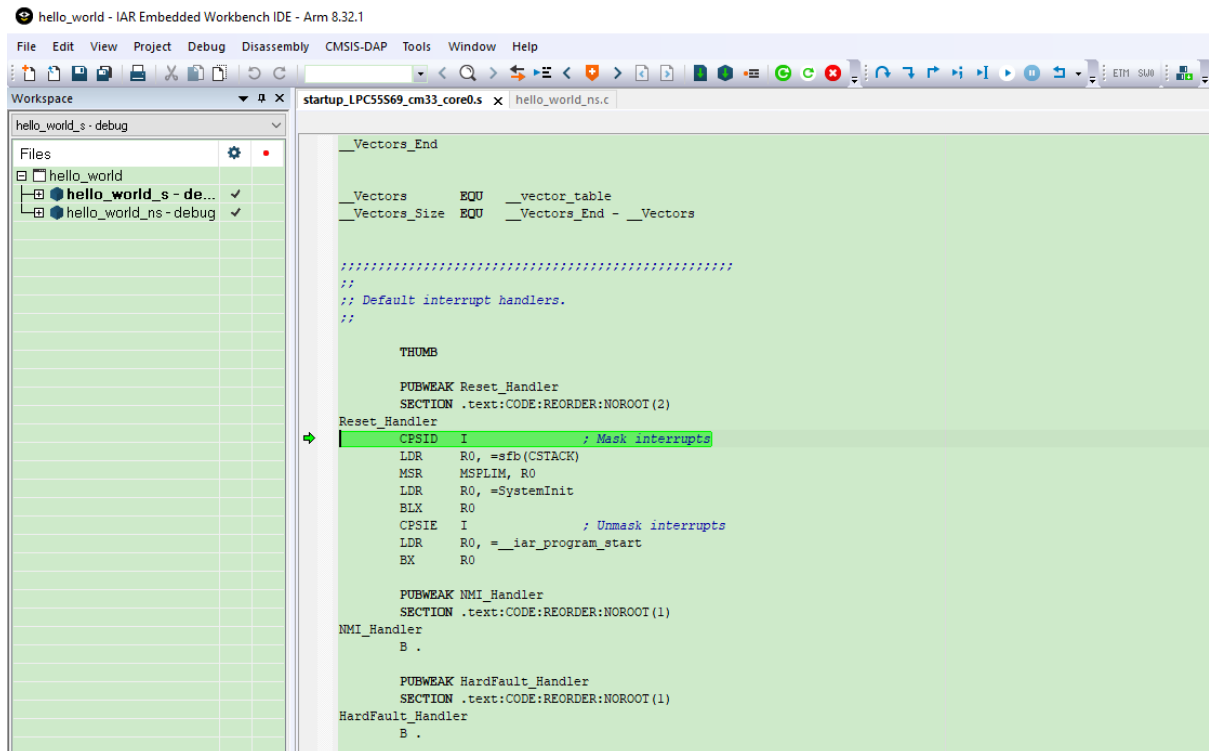
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/iar/hello_world_  
↔ns.eww
```

```
<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
```

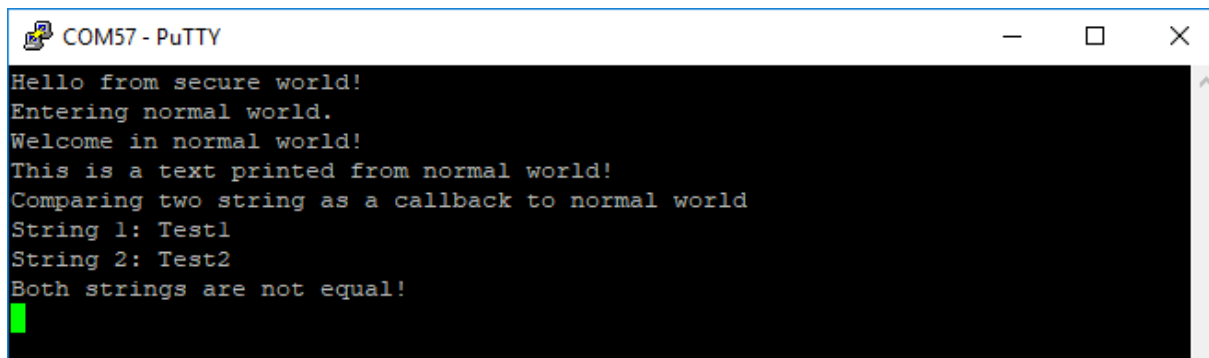
This project `hello_world.eww` contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another. Build both applications separately by clicking **Make**. It is requested to build the application for the secure project first, because 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.

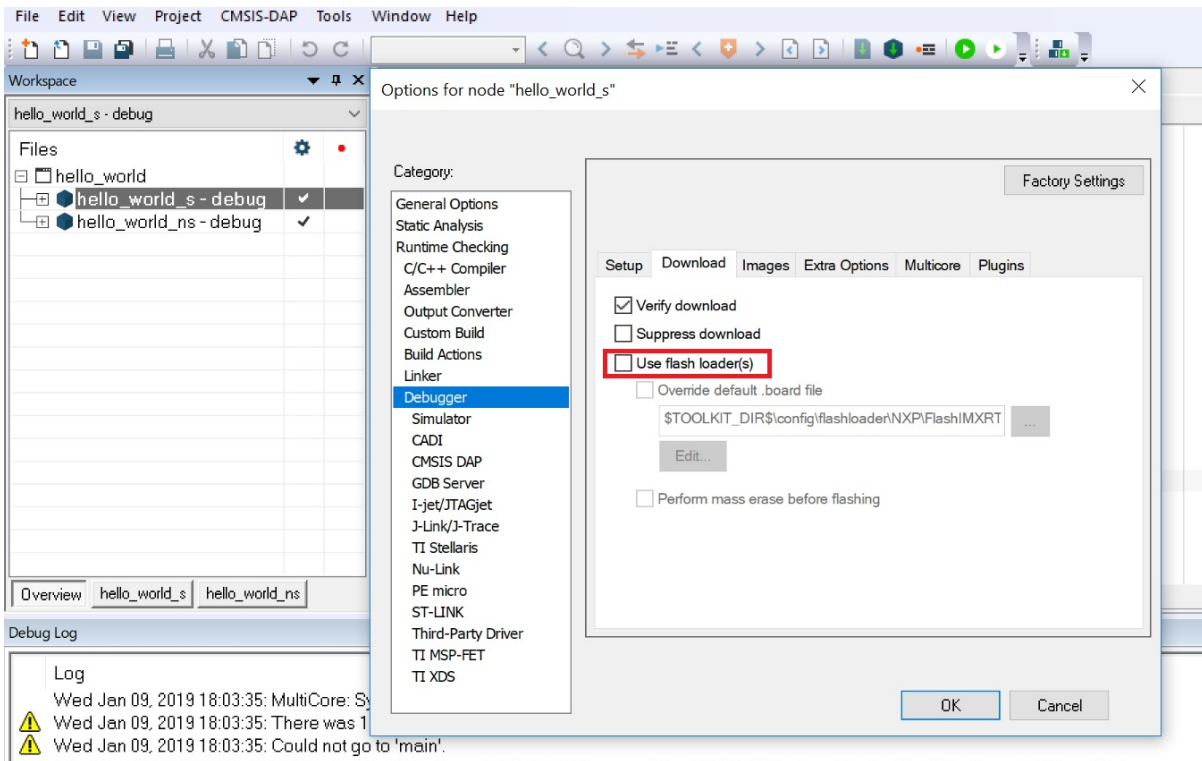


Run the code by clicking **Go** to start the application.

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.



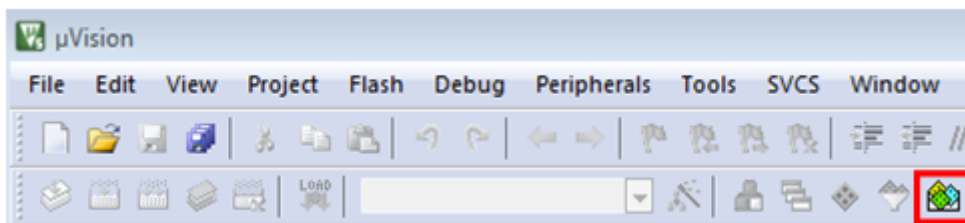
**Note:** If the application is running in RAM (debug/release build target), in **Options\*\*>\*\*Debugger > Download** tab, disable **Use flash loader(s)**. This can avoid the `__ns` download issue on i.MXRT500.



**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:

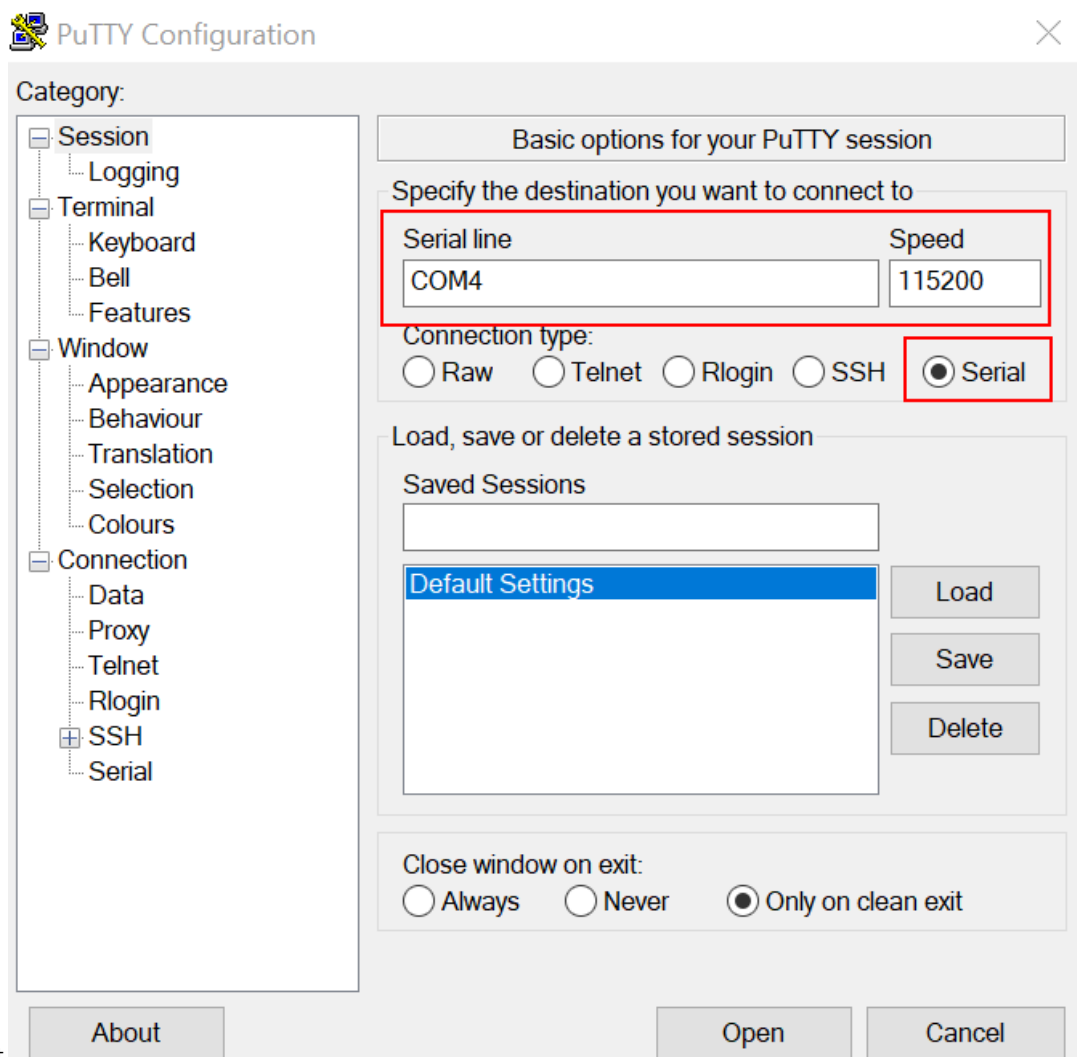
- To build the demo project, select **Rebuild**, highlighted in red.



- The build completes without errors.

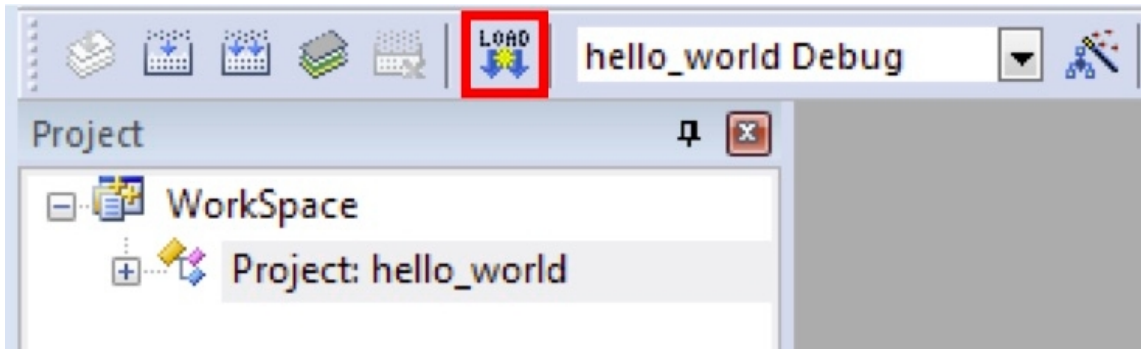
**Run an example application** To download and run the application, perform these steps:

- 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

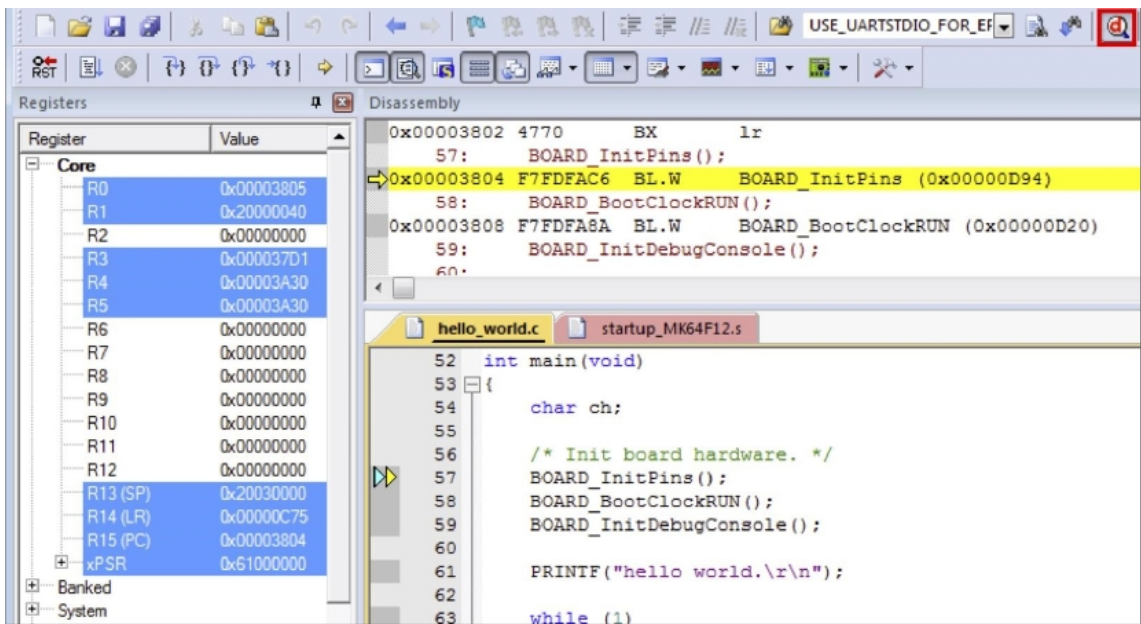


- 1 stop bit

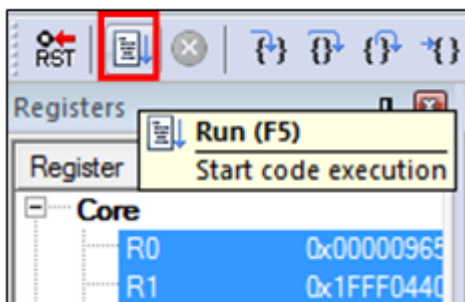
- In  $\mu$ Vision, after the application is built, click the **Download** button to download the application to the target.



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/ $\mu$ Vision workspaces are located in this folder:

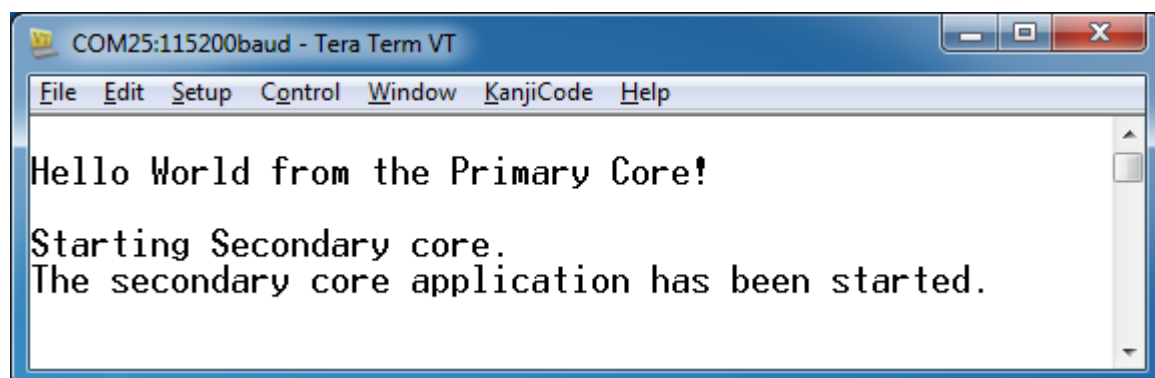
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/mdk/hello_world_
↪cm0plus.uvmpw
```

```
<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.

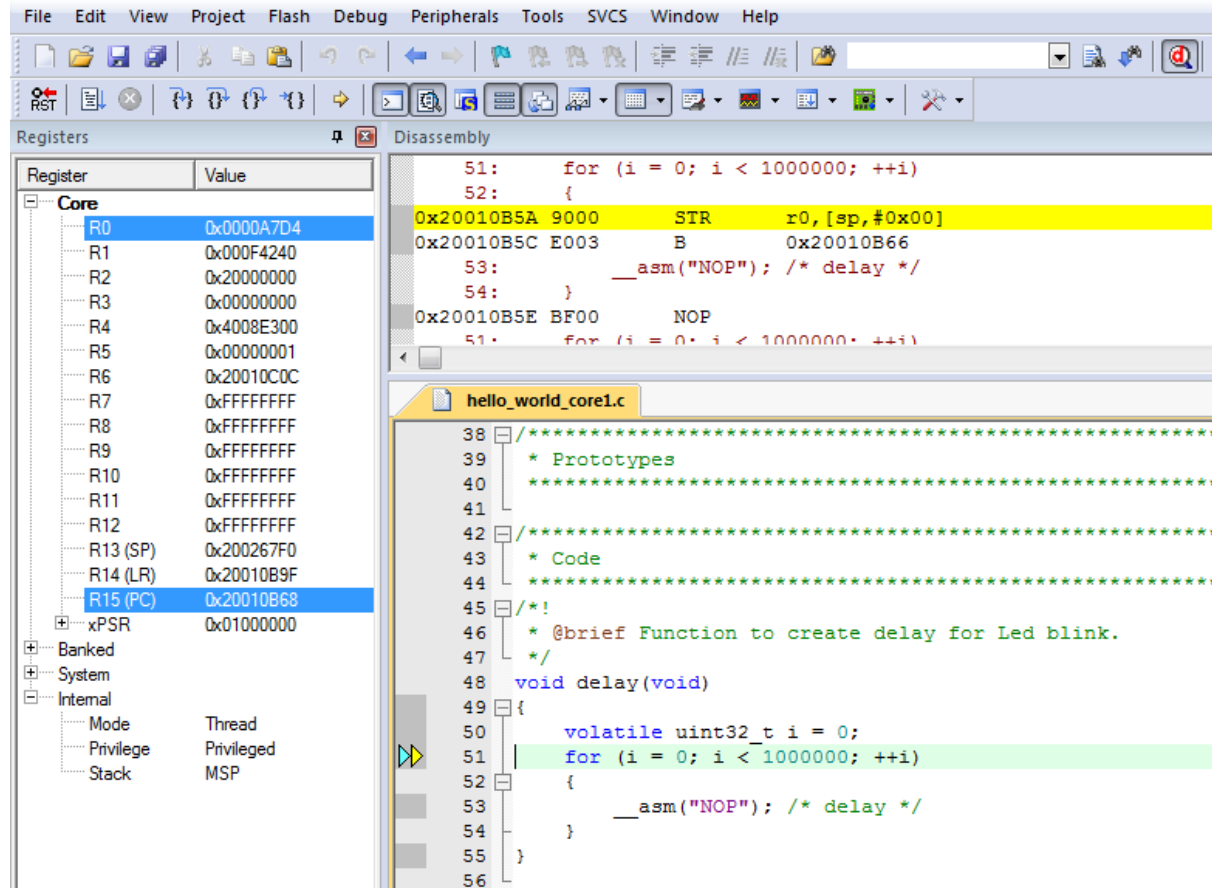
**Run a multicore example application** The primary core debugger flashes both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 5 as described in **Run an example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

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  $\mu$ 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](http://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/ $\mu$ 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
```

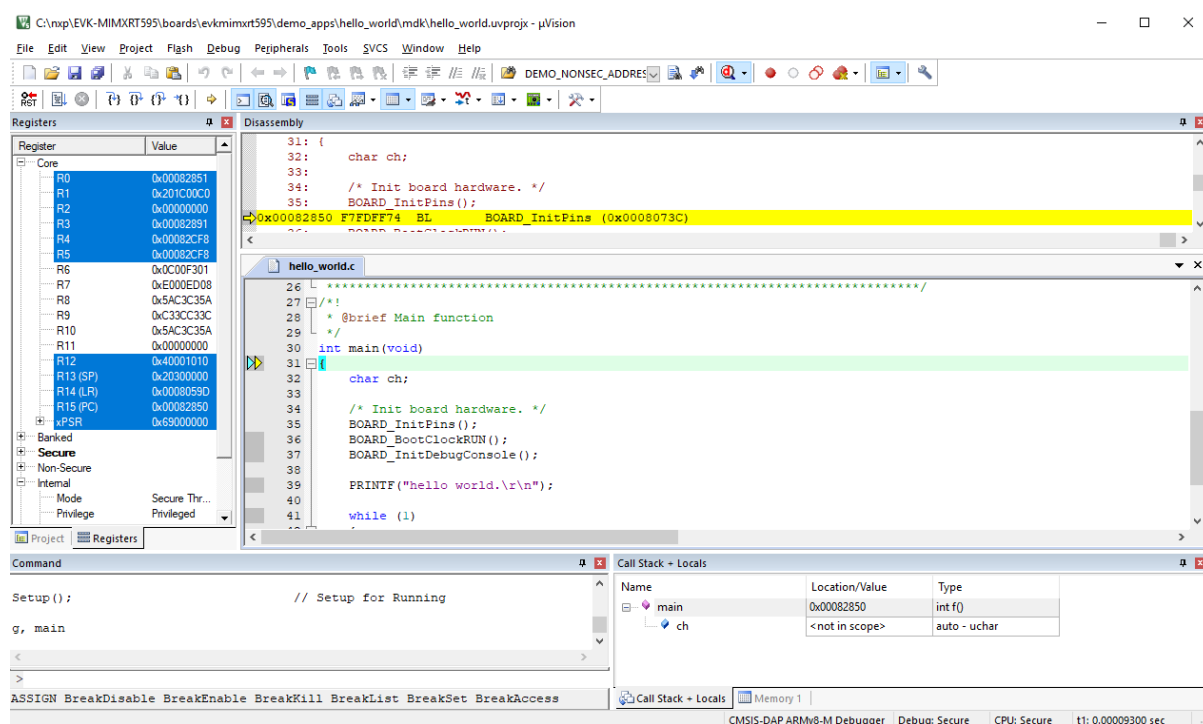
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world.  
↪ uvmpw
```

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.

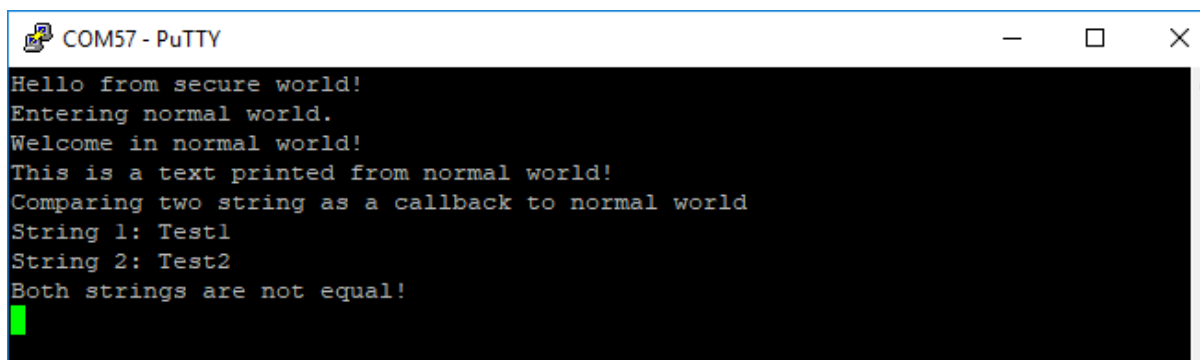
**Run a TrustZone example application** The secure project is configured to download both secure and non-secure output files so debugging can be fully managed from the secure project.

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  $\mu$ 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 ARMGCC / VSCODE** This section describes the steps to run an example application from the SDK archive using the ARMGCC / VSCODE toolchain.

Refer to the [running a demo using MCUXpresso VSC](#) section for detailed instructions on setting up and configuring your project in Visual Studio Code.

Refer to the [CLI](#) section for detailed instructions on building and running your project from the command line.

**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
<b>Pins tool</b>	For configuration of pin routing and pin electrical properties.	
<b>Clock tool</b>	For system clock configuration	
<b>Peripherals tools</b>	For configuration of other peripherals	
<b>TEE tool</b>	Configures access policies for memory area and peripherals helping to protect and isolate sensitive parts of the application.	
<b>Device Configuration tool</b>	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](http://www.nxp.com/mcuxpresso). Recommended for customers using IAR Embedded Workbench, Keil MDK  $\mu$ Vision, or Arm GCC.
- **Online version** available on [mcuxpresso.nxp.com](http://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.

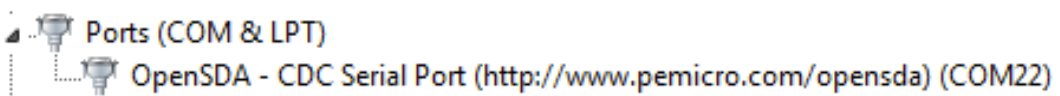
2. **Windows:** To determine the COM port open Device Manager in the Windows operating system. Click the **Start** menu and type **Device Manager** in the search bar.

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.

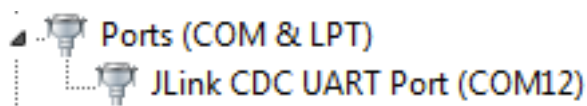
1. **CMSIS-DAP/mbed/DAPLink** interface:



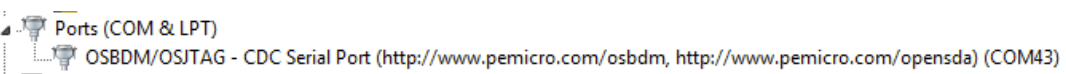
2. **P&E Micro:**



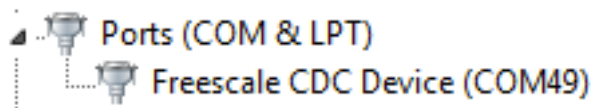
3. **J-Link:**



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](http://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](http://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](http://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](http://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/mbed 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.**

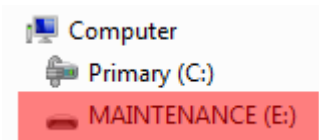
- For boards with CMSIS-DAP firmware, visit [developer.mbed.org/handbook/Windows-serial-configuration](https://developer.mbed.org/handbook/Windows-serial-configuration) and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
- 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](http://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](http://www.nxp.com/opensda).
- P&E Micro: Downloading P&E Micro OpenSDA firmware images requires registration with P&E Micro ([www.pemicro.com](http://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-MCXA174	CMSIS-DAP	MCU-Link
FRDM-MCXA266	CMSIS-DAP	MCU-Link
FRDM-MCXA344	CMSIS-DAP	MCU-Link
FRDM-MCXA346	CMSIS-DAP	MCU-Link
FRDM-MCXA366	CMSIS-DAP	MCU-Link
FRDM-MCXA577	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-MCXE31B	CMSIS-DAP	MCU-Link
FRDM-MCXN236	CMSIS-DAP	MCU-Link

continues on next page

Table 1 – continued from previous page

Hardware platform	Default debugger firmware	On-board debugger probe
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
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
}
```

## Repository-Layout SDK Package

**Development Tools Installation** This guide explains how to install the essential tools for development with the MCUXpresso SDK.

**Quick Start: Automated Installation (Recommended)** The **MCUXpresso Installer** is the fastest way to get started. It automatically installs all the basic tools you need.

1. **Download the MCUXpresso Installer** from: [Dependency-Installation](#)
2. **Run the installer** and select “**MCUXpresso SDK Developer**” from the menu
3. **Click Install** and let it handle everything automatically

**Manual Installation** If you prefer to install tools manually or need specific versions, follow these steps:

## Essential Tools

**Git - Version Control** **What it does:** Manages code versions and downloads SDK repositories from GitHub.

### Installation:

- Visit [git-scm.com](https://git-scm.com)
- Download for your operating system
- Run installer with default settings
- **Important:** Make sure “Add Git to PATH” is selected during installation

### Setup:

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

**Python - Scripting Environment** **What it does:** Runs build scripts and SDK tools.

**Installation:**

- Install Python **3.10 or newer** from [python.org](https://python.org)
- **Important:** Check “Add Python to PATH” during installation

**West - SDK Management Tool** **What it does:** Manages SDK repositories and provides build commands. The west tool is developed by the Zephyr project for managing multiple repositories.

**Installation:**

```
pip install -U west
```

**Minimum version:** 1.2.0 or newer

## Build System Tools

**CMake - Build Configuration** **What it does:** Configures how your projects are built.

**Recommended version:** 3.30.0 or newer

**Installation:**

- **Windows:** Download .msi installer from [cmake.org/download](https://cmake.org/download)
- **Linux:** Use package manager or download from [cmake.org](https://cmake.org)
- **macOS:** Use Homebrew (`brew install cmake`) or download from [cmake.org](https://cmake.org)

**Ninja - Fast Build System** **What it does:** Compiles your code quickly.

**Minimum version:** 1.12.1 or newer

**Installation:**

- **Windows:** Usually included, or download from [ninja-build.org](https://ninja-build.org)
- **Linux:** `sudo apt install ninja-build` or download binary
- **macOS:** `brew install ninja` or download binary

**Ruby - IDE Project Generation (Optional)** **What it does:** Generates project files for IDEs like IAR and Keil.

**When needed:** Only if you want to use traditional IDEs instead of VS Code.

**Installation:** Follow the Ruby environment setup guide

**Compiler Toolchains** Choose and install the compiler toolchain you want to use:

Toolchain	Best For	Download Link	Environment Variable
<b>ARM GCC</b> (Recommended)	Most users, free	<a href="#">ARM GNU Toolchain</a>	ARMGCC_DIR
<b>IAR EWARM</b>	Professional development	<a href="#">IAR Systems</a>	IAR_DIR
<b>Keil MDK ARM Compiler</b>	ARM ecosystem Advanced optimization	<a href="#">ARM Developer</a> <a href="#">ARM Developer</a>	MDK_DIR ARMCLANG_DIR

**Setting Up Environment Variables** After toolchain installation, set an environment variable so the build system locates it:

**Windows:**

```
# Example for ARM GCC installed in C:\armgcc
setx ARMGCC_DIR "C:\armgcc"
```

**Linux/macOS:**

```
# Add to ~/.bashrc or ~/.zshrc
export ARMGCC_DIR="/usr" # or your installation path
```

**Verify Your Installation** After installation, verify everything works by opening a terminal/command prompt and running these commands:

```
# Check each tool - you should see version numbers
git --version
python --version
west --version
cmake --version
ninja --version
arm-none-eabi-gcc --version # (if using ARM GCC)
```

**Troubleshooting Installation Issues** “Command not found” errors:

- The tool isn’t in your system PATH
- **Solution:** Add the installation directory to your PATH environment variable

**Python/pip issues:**

- Try using python3 and pip3 instead of python and pip
- On Windows, run the Command Prompt as an Administrator

**Slow downloads:**

- Add timeout option: pip install -U west --default-timeout=1000
- Use alternative mirror: pip install -U west -i https://pypi.tuna.tsinghua.edu.cn/simple

**Building Your First Project** This guide explains how to build and run your first SDK example project using the west build system. This applies to both GitHub Repository SDK and Repository-Layout SDK Package.

**Prerequisites**

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development board connected via USB
- Build tools installed per [Installation Guide](#)

**Understanding Board Support** Use the west extension to discover available examples for your board:

```
west list _project -p examples/demo_apps/hello_world
```

This shows all supported build configurations. You can filter by toolchain:

```
west list _project -p examples/demo_apps/hello_world -t armgcc
```

## Basic Build Process

**Simple Build** Build the hello\_world example with default settings:

```
west build -b your_board examples/demo_apps/hello_world
```

The default toolchain is armgcc, and the build system will select the first debug target as default if no config is specified.

## Specifying Configuration

```
# Release build
west build -b your_board examples/demo_apps/hello_world --config release
```

```
# Debug build (default)
west build -b your_board examples/demo_apps/hello_world --config debug
```

## Alternative Toolchains

```
# IAR toolchain
west build -b your_board examples/demo_apps/hello_world --toolchain iar
```

```
# Other toolchains as supported by the example
```

**Multicore Applications** For multicore devices, specify the core ID:

```
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config_
↪ flexspi_nor_debug
```

For multicore projects using sysbuild:

```
west build -b evkbmimxrt1170 --sysbuild ./examples/multicore_examples/hello_world/primary -Dcore_
↪ id=cm7 --config flexspi_nor_debug --toolchain=armgcc -p always
```

**Flash an Application** Flash the built application to your board:

```
west flash -r linkserver
```

**Debug** Start a debug session:

```
west debug -r linkserver
```

## Common Build Options

**Clean Build** Force a complete rebuild:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

**Dry Run** See the commands that get executed without running them:

```
west build -b your_board examples/demo_apps/hello_world --dry-run
```

**Device Variants** For boards supporting multiple device variants:

```
west build -b your_board examples/demo_apps/hello_world --device DEVICE_PART_NUMBER --config ↵  
↵release
```

## Project Configuration

**CMake Configuration Only** Run configuration without building:

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

**Interactive Configuration** Launch the configuration GUI:

```
west build -t guiconfig
```

## Troubleshooting

**Build Failures** Use pristine builds to resolve dependency issues:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

**Getting Help** View the help information for west build:

```
west build -h
```

**Check Supported Configurations** To see available configuration options and board targets for an example, refer to the below command:

```
west list_project -p examples/demo_apps/hello_world
```

## Next Steps

- Explore other examples in the SDK
- Learn about [Command Line Development](#) for advanced options
- Try [VS Code Development](#) for integrated development
- Refer [Workspace Structure](#) to understand the SDK layout

**MCUXpresso for VS Code Development** This guide covers using MCUXpresso for VS Code extension to build, debug, and develop SDK applications with an integrated development environment.

## Prerequisites

- SDK workspace initialized (GitHub Repository SDK or Repository-Layout SDK Package)
- Development tools installed per [Installation Guide](#)
- Visual Studio Code installed
- MCUXpresso for VS Code extension installed

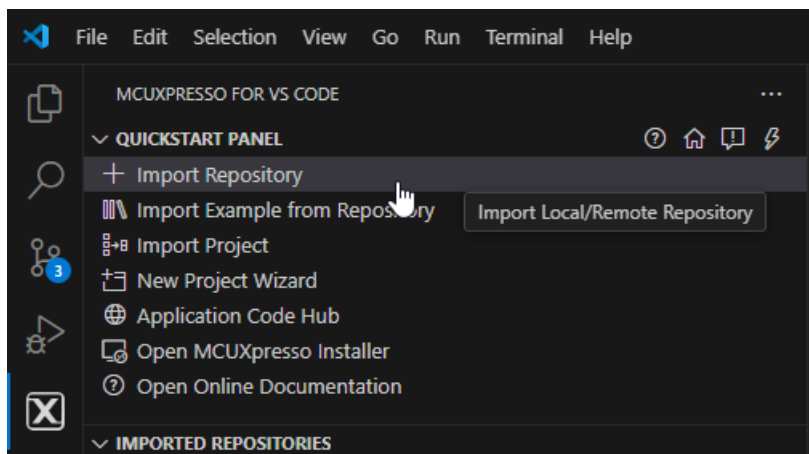
## Extension Installation

**Install MCUXpresso for VS Code** The MCUXpresso for VS Code extension provides integrated development capabilities for MCUXpresso SDK projects. Refer to the [MCUXpresso for VS Code Wiki](#) for detailed installation and setup instructions.

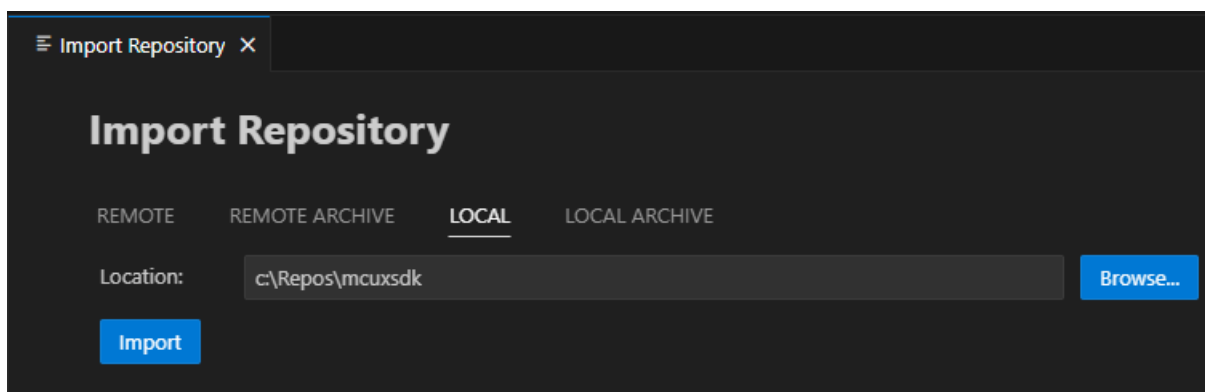
## SDK Import and Setup

**Import Methods** The SDK can be imported in several ways. The MCUXpresso for VS Code extension supports both GitHub Repository SDK and Repository-Layout SDK Package distributions.

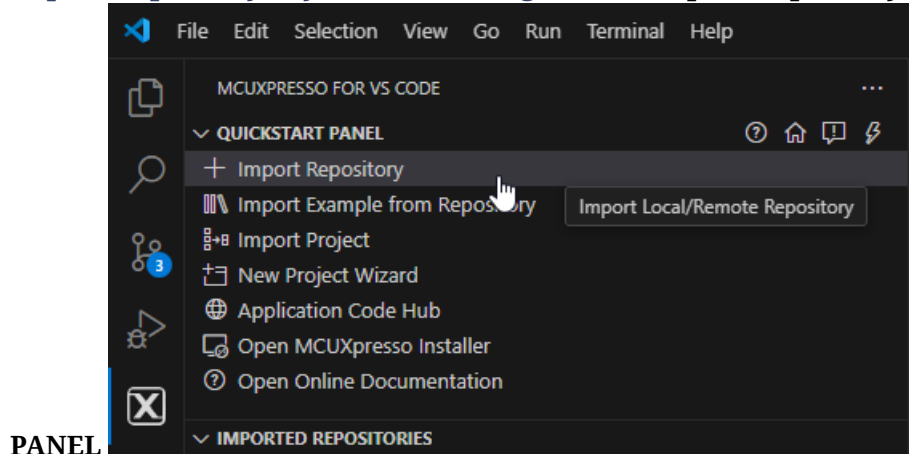
**Import GitHub Repository SDK** Click **Import Repository** from the **QUICKSTART PANEL**



**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 according to [setting up the repo](#). Select your location and click **Import**.

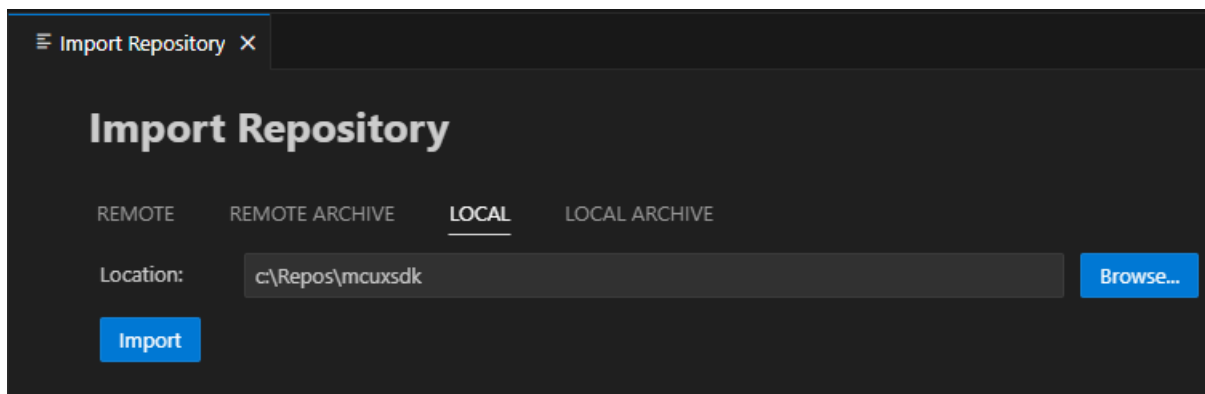


**Import Repository-Layout SDK Package** Click **Import Repository** from the **QUICKSTART**

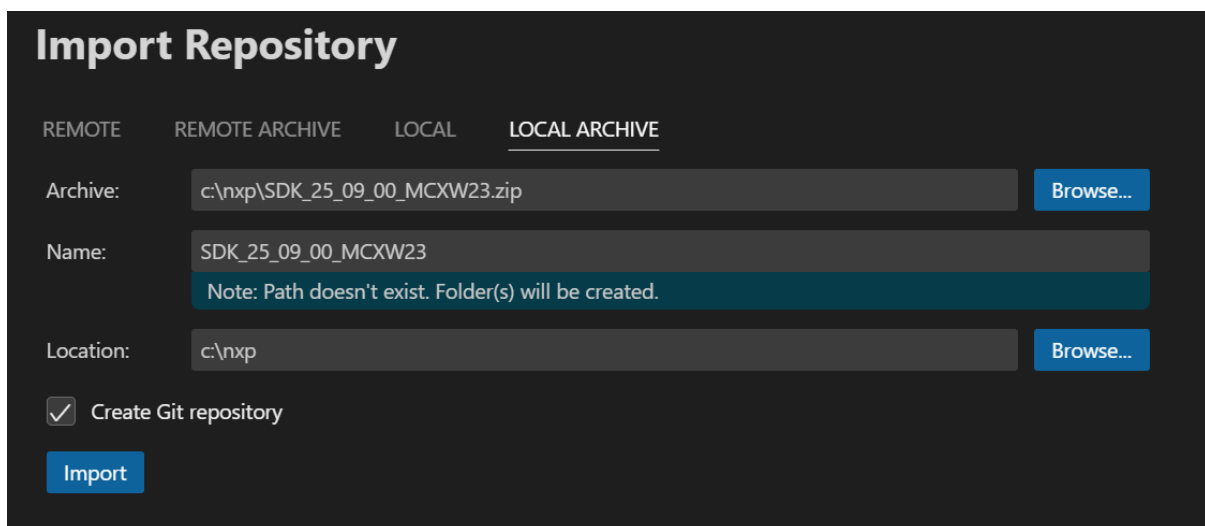


**PANEL**

Select **Local** if you've already unzipped the Repository-Layout SDK Package. Select your location and click **Import**.



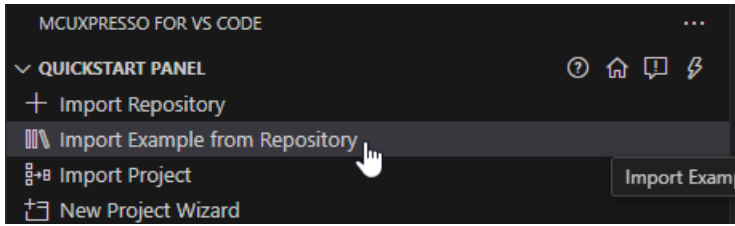
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



## Building Example Applications

### Import Example Project

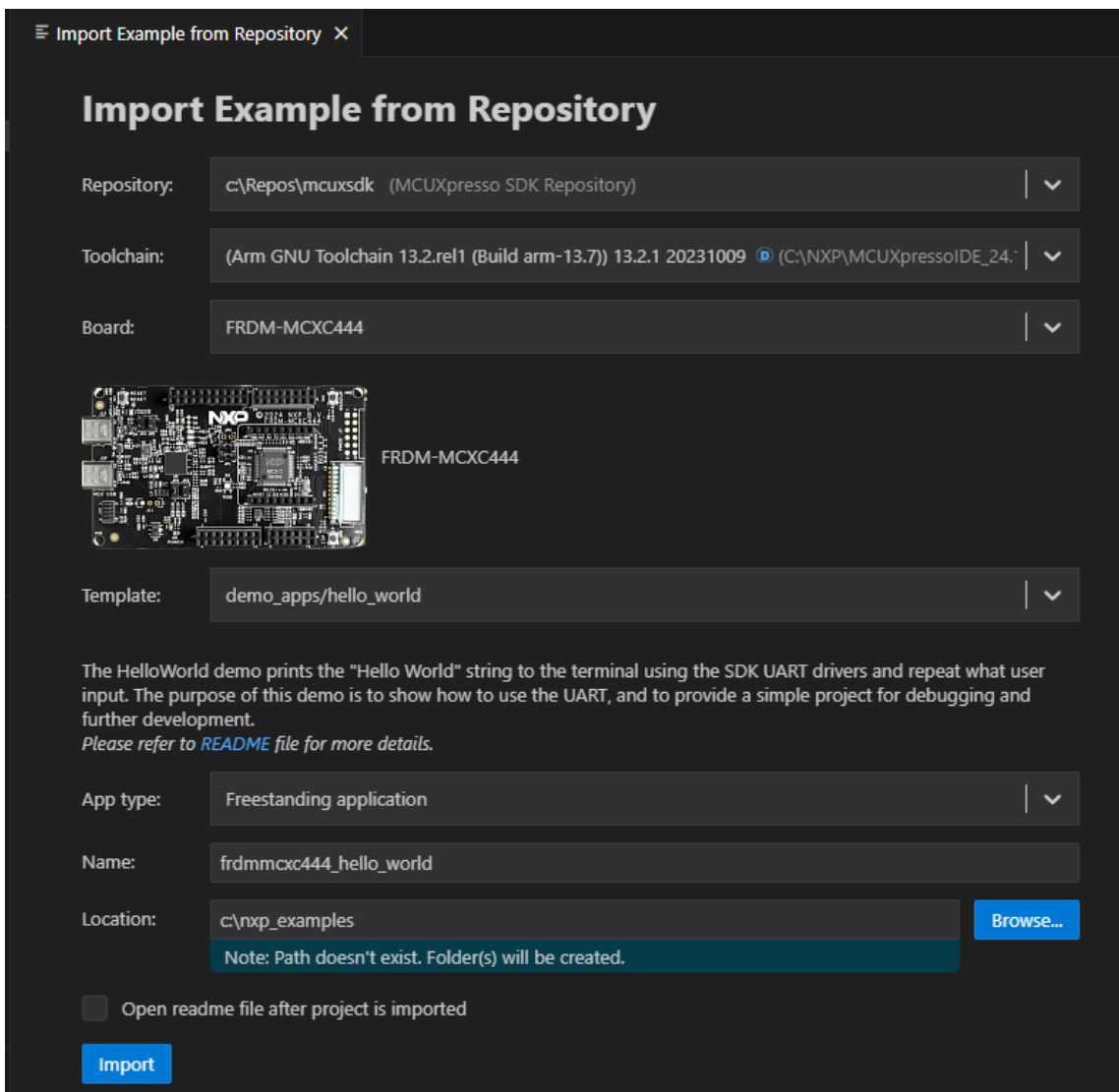
1. Click **Import Example from Repository** from the **QUICKSTART PANEL**



## 2. Configure project settings:

- **MCUXpresso SDK:** Select your imported SDK
- **Arm GNU Toolchain:** Choose toolchain
- **Board:** Select your target development board
- **Template:** Choose example category
- **Application:** Select specific example (e.g., hello\_world)
- **App type:** Choose between Repository applications or Freestanding applications

## 3. Click **Import**



## Application Types Repository Applications:

- Located inside the MCUXpresso SDK
- Integrated with SDK workspace

#### Freestanding Applications:

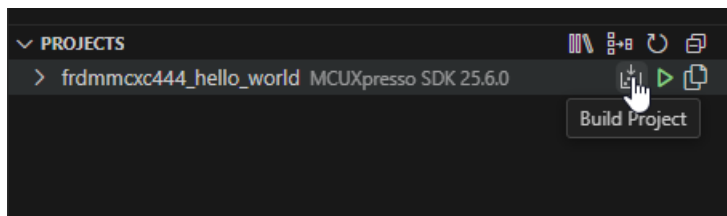
- Imported to user-defined location
- Independent of SDK location

**Trust Confirmation** VS Code will prompt you to confirm if the imported files are trusted. Click **Yes** to proceed.

## Building Projects

### Build Process

1. Navigate to **PROJECTS** view
2. Find your project
3. Click the **Build Project** icon

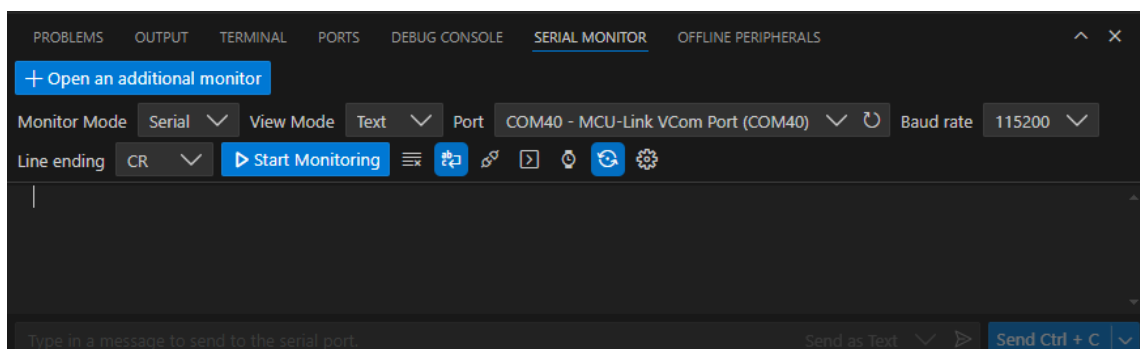


The integrated terminal will display build output at the bottom of the VS Code window.

## Running and Debugging

### Serial Monitor Setup

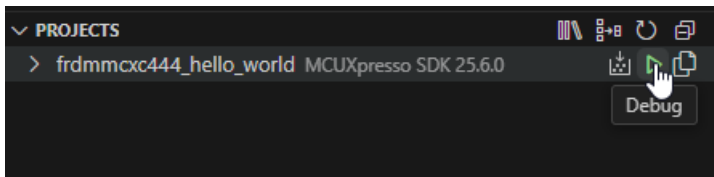
1. Open **Serial Monitor** from VS Code's integrated terminal



2. Configure serial settings:
  - **VCom Port:** Select port for your device
  - **Baud Rate:** Set to 115200

## Debug Session

1. Navigate to **PROJECTS** view
2. Click the play button to initiate a debug session



The debug session will begin with debug controls initially at the top of the interface.

**Debug Controls** Use the debug controls to manage execution:

- **Continue:** Resume code execution
- **Step controls:** Navigate through code

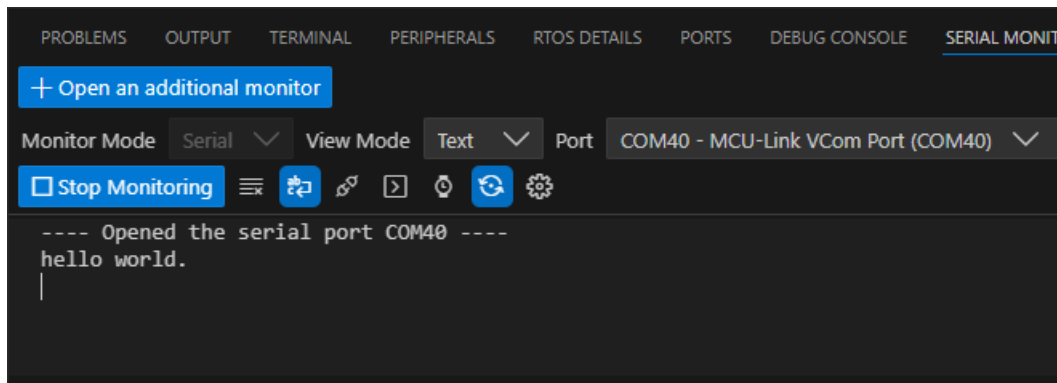
```

C hello_world.c X
frdmmxc444_hello_world > examples > demo_apps > hello_world > C hello_w
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

```

- **Stop:** Terminate debug session

**Monitor Output** Observe application output in the **Serial Monitor** to verify correct operation.



**Debug Probe Support** For comprehensive information on debug probe support and configuration, refer to the [MCUXpresso for VS Code Wiki DebugK section](#).

## Project Configuration

**Workspace Management** The extension integrates with the MCUXpresso SDK workspace structure, providing access to:

- Example applications
- Board configurations
- Middleware components
- Build system integration

**Multi-Project Support** The PROJECTS view allows management of multiple imported projects within the same workspace.

## Troubleshooting

### Import Issues **SDK not detected:**

- Verify SDK workspace is properly initialized
- Ensure all required repositories are updated
- Check SDK manifest files are present

### Project import failures:

- Confirm board support exists for selected example
- Verify toolchain installation
- Check example compatibility with selected board

### Build Problems **Build failures:**

- Check integrated terminal for error messages
- Verify all dependencies are installed
- Ensure toolchain is properly configured

**Debug Issues** **Debug session fails:**

- Verify board connection via USB
- Check debug probe drivers are installed
- Confirm build completed successfully

**Serial monitor problems:**

- Verify correct VCom port selection
- Check baud rate configuration (115200)
- Ensure board drivers are installed

**Integration with Command Line** MCUXpresso for VS Code integrates with the underlying west build system, allowing seamless integration with command line workflows described in [Command Line Development](#).

**Advanced Features**

**Project Types** The extension supports both repository-based and freestanding project types, providing flexibility in project organization and SDK integration.

**Build System Integration** The extension leverages the MCUXpresso SDK build system, providing access to all build configurations and options available through command line tools.

**Next Steps**

- Explore additional examples in the SDK
- Review [Command Line Development](#) for advanced build options
- Refer [MCUXpresso for VS Code Wiki](#) for detailed documentation
- Learn about [SDK Architecture](#) for better understanding of the development environment

**Command Line Development** This guide covers developing with the MCUXpresso SDK using command line tools and the west build system. This workflow applies to both GitHub Repository SDK and Repository-Layout SDK Package distributions.

**Prerequisites**

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- Development tools installed per [Installation Guide](#)
- Target board connected via USB

**Understanding Board Support** Use the west extension to discover available examples for your board:

```
west list _project -p examples/demo_apps/hello_world
```

This shows all supported build configurations. You can filter by toolchain:

```
west list _project -p examples/demo_apps/hello_world -t armgcc
```

## Basic Build Commands

**Standard Build Process** Build with default settings (armgcc toolchain, first debug config):

```
west build -b your_board examples/demo_apps/hello_world
```

## Specifying Build Configuration

# Release build

```
west build -b your_board examples/demo_apps/hello_world --config release
```

# Debug build with specific toolchain

```
west build -b your_board examples/demo_apps/hello_world --toolchain iar --config debug
```

**Multicore Applications** For multicore devices, specify the core ID:

```
west build -b evkbnimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config ↵  
↵ flexspi_nor_debug
```

For multicore projects using sysbuild:

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

**Shield Support** For boards with shields:

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

## Advanced Build Options

**Clean Builds** Force a complete rebuild:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

**Dry Run** See what commands would be executed:

```
west build -b your_board examples/demo_apps/hello_world --dry-run
```

**Device Variants** For boards supporting multiple device variants:

```
west build -b your_board examples/demo_apps/hello_world --device MK22F12810 --config release
```

## Project Configuration

**CMake Configuration Only** Run configuration without building:

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

**Interactive Configuration** Launch the configuration GUI:

```
west build -t guiconfig
```

## Flashing and Debugging

**Flash Application** Flash the built application to your board:

```
west flash -r linkserver
```

**Debug Session** Start a debugging session:

```
west debug -r linkserver
```

**IDE Project Generation** Generate IDE project files for traditional IDEs:

```
# Generate IAR project
west build -b evkbmimxrt1170 examples/demo_apps/hello_world --toolchain iar -Dcore_id=cm7 --config_
↪ flexspi_nor_debug -p always -t guiproject
```

IDE project files are generated in `mcuxsdk/build/<toolchain>` folder.

**Note:** Ruby installation is required for IDE project generation. See [Installation Guide](#) for setup instructions.

## Troubleshooting

**Build Failures** Use pristine builds to resolve dependency issues:

```
west build -b your_board examples/demo_apps/hello_world -p always
```

**Toolchain Issues** Verify environment variables are set correctly:

```
# Check ARM GCC
echo $ARMGCC_DIR
arm-none-eabi-gcc --version
```

```
# Check IAR (if using)
echo $IAR_DIR
```

**Getting Help** Display help information:

```
west build -h
west flash -h
west debug -h
```

**Check Supported Configurations** If unsure about supported options for an example:

```
west list _project -p examples/demo_apps/hello_world
```

## Best Practices

### Project Organization

- Keep custom projects outside the SDK tree
- Use version control for your application code
- Document any SDK modifications

### Build Efficiency

- Use `-p` always for clean builds when troubleshooting
- Leverage `--dry-run` to understand build processes
- Use specific configs and toolchains to reduce build time

### Development Workflow

1. Start with existing examples closest to your requirements
2. Copy and modify rather than building from scratch
3. Test with `hello_world` before moving to complex examples
4. Use configuration tools for pin muxing and clock setup

### Next Steps

- Explore [VS Code Development](#) for integrated development experience
- Review [Workspace Structure](#) to understand SDK organization
- Refer build system documentation for advanced configurations

**Workspace Structure** After you initialize your SDK workspace, it creates a specific directory structure that organizes all SDK components. This structure is identical for both GitHub Repository SDK and Repository-Layout SDK Package.

### Top-Level Organization

```
your-sdk-workspace/  
  manifests/      # West manifest repository  
  mcuxsdk/       # Main SDK content
```

The `mcuxsdk/` directory serves as your primary working directory and contains all the SDK components.

**SDK Component Layout** Based on the actual SDK structure, the main directories include:

Directory	Contents	Purpose
arch/	Architecture-specific files	ARM CMSIS, build configurations
cmake/	Build system modules	CMake configuration and build rules
compo	Software components	Reusable software libraries and utilities
devices/	Device support packages	MCU-specific headers, startup code, linker scripts
drivers/	Peripheral drivers	Hardware abstraction layer for MCU peripherals
examp	Sample applications	Demonstration code and reference implementations
middle	Optional software stacks	Networking, graphics, security, and other libraries
rtos/	Operating system support	FreeRTOS integration
scripts	Build and utility scripts	West extensions and development tools
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.	

**Example Organization** Examples follow a two-tier structure separating common code from board-specific implementations:

### Common Example Files

```
examples/demo_apps/hello_world/
  CMakeLists.txt      # Build configuration
  example.yml         # Example metadata
  hello_world.c       # Application source code
  Kconfig             # Configuration options
  readme.md           # General documentation
```

### Board-Specific Files

```
examples/_boards/your_board/demo_apps/hello_world/
  app.h               # Board specific application header
  example_board_readme.md # Board specific documentation
  hardware_init.c     # Board specific hardware initialization
  pin_mux.c           # Pin multiplexing configuration
  pin_mux.h           # Pin multiplexing header definitions
  hello_world.bin     # Pre-built binary for quick testing
  hello_world.mex     # MCUXpresso Config Tools project file
  prj.conf            # Board specific Kconfig configuration
  reconfig.cmake     # Board specific cmake configuration overrides
```

**Device Support Structure** Device support is organized hierarchically by MCU family:

```
devices/  
  MCX/           # MCU portfolio  
    MCXW/        # MCU family  
      MCXW235/   # Specific device  
        MCXW235.h # Device register definitions  
  drivers/       # Device-specific drivers  
  gcc/           # GNU toolchain files  
  iar/           # IAR toolchain files  
  mcuxpresso/   # MCUXpresso IDE files  
  startup_MCXW235.c # Startup and vector table  
  system_MCXW235.c # System initialization
```

**Middleware Organization** Middleware components are categorized by functionality and maintained in separate repositories. Based on the manifest files, common middleware categories include:

- **Connectivity:** USB, TCP/IP, industrial protocols
- **Security:** Cryptographic libraries, secure boot
- **Wireless:** Bluetooth, IEEE 802.15.4, Wi-Fi
- **Graphics:** Display drivers, UI frameworks
- **Audio:** Processing libraries, voice recognition
- **Machine Learning:** Inference engines, neural networks
- **Safety:** IEC60730B safety libraries
- **Motor Control:** Motor control and real-time control libraries

**Documentation Structure** SDK documentation is distributed across multiple locations:

- docs/ - Core SDK documentation and build infrastructure
- Component repositories - API documentation and integration guides
- Board directories - Hardware-specific setup instructions

For complete documentation, refer to the [online documentation](#).

**Understanding Example Structure** Each example has **two README files**:

**1. General README:** examples/demo\_apps/hello\_world/readme.md

- What the example does
- General functionality description
- Common usage information

**2. Board-Specific README:** examples/\_boards/{board\_name}/demo\_apps/hello\_world/example\_board\_readme.md

- Board-specific setup instructions
- Hardware connections required
- Board-specific behavior notes

**Tip:** Always check both readme files - start with the general one, then read the board-specific one for detailed setup.

## 1.3 Getting Started with MCUXpresso SDK GitHub

### 1.3.1 Getting Started with MCUXpresso SDK Repository

Welcome to the **GitHub Repository SDK Guide**. This documentation provides instructions for setting up and working with the MCUXpresso SDK distributed in a **multi-repository model**. The SDK is distributed across multiple GitHub repositories and managed using the **Zephyr West** tool, enabling modular development and streamlined workflows.

#### Overview

The GitHub Repository SDK approach offers:

- **Modular Structure:** Multiple repositories for flexibility and scalability.
- **Zephyr West Integration:** Simplified repository management and synchronization.
- **Cross-Platform Support:** Designed for MCUXpresso SDK development environments.

#### Benefits of the Multi-Repository Approach

- **Scalability:** Easily add or update components without impacting the entire SDK.
- **Collaboration:** Enables distributed development across teams and repositories.
- **Version Control:** Independent versioning for components ensures better stability.
- **Automation:** Zephyr West simplifies dependency handling and repository synchronization.

#### Setup and Configuration

Follow these steps to prepare your development environment:

**GitHub Repository Setup** This guide explains how to initialize your MCUXpresso SDK workspace from GitHub repositories using the west tool. The GitHub Repository SDK uses multiple repositories hosted on GitHub to provide modular, flexible development.

**Prerequisites** Verify the requirements:

#### System Requirements:

- Python 3.8 or later
- Git 2.25 or later
- CMake 3.20 or later
- Build tools for your target platform

#### Verification Commands:

```
python --version # Should show 3.8+
git --version # Should show 2.25+
cmake --version # Should show 3.20+
west --version # Should show west tool installation
```

**Workspace Initialization** The GitHub Repository SDK uses the Zephyr west tool to manage multiple repositories containing different SDK components.

**Step 1: Initialize Workspace** Create and initialize your SDK workspace from GitHub:

**Get the latest SDK from main branch:**

```
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests.git mcuxpresso-sdk
```

**Get SDK at specific revision:**

```
west init -m https://github.com/nxp-mcuxpresso/mcuxsdk-manifests.git mcuxpresso-sdk --mr {revision}
```

*Note: Replace {revision} with the desired release tag, such as v25.09.00*

**Step 2: Choose Your Repository Update Strategy** Navigate to the SDK workspace:

```
cd mcuxpresso-sdk
```

The west tool manages multiple GitHub repositories containing different SDK components. You have two options for downloading:

**Option A: Download All Repositories (Complete SDK)** Download all SDK repositories for comprehensive development:

```
west update
```

This command downloads all the repositories defined in the manifest from GitHub. Initial download takes several minutes and requires ~7 GB of disk space.

**Best for:**

- Exploring the complete SDK
- Multi-board development projects
- Comprehensive middleware evaluation

**Option B: Targeted Repository Download (Recommended)** Download only repositories needed for your specific board or device to save time and disk space:

```
# For specific board development
west update_board --set board your_board_name

# For specific device family development
west update_board --set device your_device_name

# List available repositories before downloading
west update_board --set board your_board_name --list-repo
```

**Best for:**

- Single board development

- Faster setup and reduced disk usage
- Focused development workflows

**Examples:**

```
# Update only repositories for FRDM-MCXW23 board
west update_board --set board frdm-mcxw23

# Update only repositories for MCXW23 device family
west update_board --set device mcxw23
```

**Step 3: Verify Installation** Confirm successful setup:

```
# Verify workspace structure
ls -la
# Should show: manifests/ and mcuxsdk/ directories

# Test build system
west list_project -p examples/demo_apps/hello_world
# Should display available build configurations
```

**Advanced Repository Management** The `west update_board` command provides advanced repository management capabilities for optimized workspace setup with GitHub repositories.

**Board-Specific Setup** Update only repositories required for a specific board:

```
# Update only repositories for specific board, e.g., frdm-mcxw23
west update_board --set board frdm-mcxw23

# List available repositories for the board before updating
west update_board --set board frdm-mcxw23 --list-repo
```

**Device-Specific Setup** Update only repositories required for a specific device family:

```
# Update only repositories for specific device, e.g., MCXW235
west update_board --set device mcxw23

# List available repositories for the device family
west update_board --set device mcxw23 --list-repo
```

**Custom Configuration** For advanced users who want to create custom repository combinations:

```
# Use custom configuration file
west update_board --set custom path/to/custom-config.yml

# Generate custom configuration template
cp manifests/boards/custom.yml.template my-custom-config.yml
```

**Benefits of Targeted Setup** **Reduced Download Size**

- Download only components needed for your target board or device
- Significantly faster initial setup for focused development

- Typical reduction from 7 GB to 2GB

### Optimized Workspace

- Cleaner workspace with relevant components only
- Reduced disk space usage
- Faster repository operations

### Flexible Development

- Switch between different board configurations easily
- Maintain separate workspaces for different projects
- Include optional components as needed

**Repository Information** Before setting up your workspace, you can explore what repositories are available:

```
# Display repository information in console
west update_board --set board frdmxcw23 --list-repo

# Export repository information to YAML file for reference
west update_board --set board frdmxcw23 --list-repo -o board-repos.yml
```

This command lists all the available repositories with descriptions and outlines the included components in the workspace.

**Package Generation (Optional)** The `update_board` command can also generate ZIP packages for offline distribution:

```
# Generate board-specific SDK package
west update_board --set board frdmxcw23 -o frdmxcw23-sdk.zip
```

**Note:** Package generation is primarily intended for creating custom SDK distributions. For regular development, use the workspace update commands without the `-o` option.

## Workspace Management

**Updating Your Workspace** Keep your SDK current with latest updates from GitHub:

### For Complete SDK Workspace:

```
# Update manifest repository
cd manifests
git pull

# Update all component repositories
cd ..
west update
```

### For Targeted Workspace:

```
# Update manifest repository
cd manifests
git pull

# Update board-specific repositories
cd ..
west update_board --set board your_board_name
```

**Workspace Status** Check workspace synchronization status:

```
# Show status of all repositories
west status
```

```
# Show detailed information about repositories
west list
```

**Troubleshooting Network Issues:**

- Use `west update --keep-descendants` for partial failures
- Configure Git credentials for private repositories
- Check firewall settings for Git protocol access

**Permission Issues:**

- Ensure write permissions in workspace directory
- Run commands without `sudo`/administrator privileges
- Verify Git SSH key configuration for authenticated access

**Disk Space:**

- Full SDK workspace requires approximately 7-8 GB
- Targeted workspace typically requires 1-2 GB
- Use board-specific setup to reduce workspace size

**Repository Management Issues:**

- Verify board/device names match available configurations
- Check that custom YAML files follow the correct template format
- Use `--list-repo` to verify available repositories before setup

**Next Steps** With your workspace initialized:

1. Review [Workspace Structure](#) to understand the layout
2. Build your first project with [First Build Guide](#)
3. Explore [Development Workflows MCUXpresso VSCode](#) or [Development Workflows Command Line](#) for the details on project setup and execution

For advanced repository management, see the [west tool documentation](#).

**Explore SDK Structure and Content**

Learn about the organization of the SDK and its components:

**SDK Architecture Overview** The MCUXpresso SDK uses a modular architecture where software components are distributed across multiple repositories hosted on GitHub and managed through the west tool. This approach provides flexibility, maintainability, and enables selective component inclusion.

**Repository Organization** Based on the manifest structure, the SDK consists of four main repository categories:

**Manifest Repository** The manifest repo (mcuxsdk-manifests) contains the west.yml manifest file that tracks all other repositories in the SDK.

**Base Repositories** Recorded in submanifests/base.yml and loaded in the root west.yml manifest file. These are the foundational repositories that build the SDK:

- **Devices:** MCU-specific support packages
- **Examples:** Demonstration applications and code samples
- **Boards:** Board support packages

**Middleware Repositories** Recorded in the submanifests/middleware subdirectory, categorized according to functionality:

- **Connectivity:** Networking stacks, USB, and communication protocols
- **Security:** Cryptographic libraries and secure boot components
- **Wireless:** Bluetooth, IEEE 802.15.4, and other wireless protocols
- **Graphics:** Display drivers and UI frameworks
- **Audio:** Audio processing and voice recognition libraries
- **Machine Learning:** AI inference engines and neural network libraries
- **Safety:** IEC60730B safety libraries
- **Motor Control:** Motor control and real-time control libraries

**Internal Repositories** Recorded in submanifests/internal.yml and grouped into the “bifrost” group. These are only visible to NXP internal developers and hosted on NXP internal git servers.

**Repository Hosting** Public repositories are hosted on GitHub under these organizations:

- [nxp-mcuxpresso](#)
- [NXP](#)
- [nxp-zephyr](#)

Internal repositories are hosted on NXP’s private Git infrastructure.

**Benefits of This Architecture** **Selective Integration:** Projects include only required components, reducing memory footprint and build complexity.

**Independent Versioning:** Each component maintains its own release cycle and version control.

**Community Collaboration:** Public repositories accept community contributions through standard Git workflows.

**Scalable Maintenance:** Component owners can update their repositories without affecting the entire SDK.

**Workspace Management** The west tool manages repository synchronization, version tracking, and workspace updates. All repositories are checked out under the mcuxsdk/ directory with their designated paths defined in the manifest files.

## Development Workflows

Get started with building and running projects:

**Using MCUXpresso Config Tools** MCUXpresso Config tools provide a user-friendly way to configure hardware initialization of your projects. This guide explains the basic workflow with the MCUXpresso SDK west build system and the Config Tools.

### Prerequisites

- GitHub Repository SDK workspace initialized OR Repository-Layout SDK Package extracted
- MCUXpresso Config Tools standalone installed (version 25.09 or above)
- MCUXpresso SDK Project that can be successfully built

**Board Files** MCUXpresso Config Tools generate source files for the board. These files include `pin_mux.c/h` and `clock_config.c/h`. The files contain initialization code functions that reflect the hardware configuration in the Config Tools. Within the SDK codebase, these files are specific for the board and either shared by multiple example projects or specific for one example. Open or import the configuration from the SDK project in the Config Tools and customize the settings to match the custom board or specific project use case and regenerate the code. See *User Guide for MCUXpresso Config Tools (Desktop)* (document [GSMCUXCTUG](#)) for details.

**Note:** When opening the configuration for SDK example projects, the board files may be shared across multiple examples. To ensure a separate copy of the board configuration files exists, create a freestanding project with copied board files.

**Visual Studio Code** To open the configuration in Visual Studio Code, use the context menu for the project to access Config Tools. See [MCUXpresso Extension Documentation](#) for details. Otherwise, use the manual workflow described in detail in the following section.

**Manual Workflow** Use the following steps:

1. Before using Config Tools, run the west command to get the project information for Config Tools from the SDK project files, for example:

```
west cfg_project_info -b lpcxpresso55s69 ...mcuxsdk/examples/demo_apps/hello_world/ -Dcore_
->id=cm33_core0
```

This results in the creation of the project information json file that is searched by the config tools when the configuration is created. The parameters of the command should match the build parameters that will be used for the project.

2. Launch the MCUXpresso Config Tools and in the **Start development** wizard, select **Create a new configuration based on the existing IDE/Toolchain project**. Select the created “cfg\_tools” subfolder as a project folder (for example: `...mcuxsdk/examples/demo_apps/hello_world/cfg_tools/`).

**Updating the SDK West project** **Note:** Updating project is supported with Config Tools V25.12 or newer only.

Changes in the Config tools generated source code modules may require adjustments to the toolchain project to ensure a successful build. These changes may mean, for example, adding the newly generated files, adding include paths, required drivers, or other SDK components.

This section describes how to manually resolve the changes needed in the project within the toolchain projects based on the SDK project managed by the West tool.

After the configuration in the Config Tools is finished, write updated files to the disk using the 'Update Code' command. The written files include a json file with the required changes for the toolchain project.

To resolve the changes in the project in the terminal, launch the west command that updates the project. For example:

```
west cfg_resolve -b lpcxpresso55s69 ...mcuxsdk/examples/demo_apps/hello_world/ -Dcore_id=cm33_core0
```

This command updates the appropriate cmake and kconfig files to address the changes. After this, the application can be built.

**Note:** The `cfg_resolve` command supports additional arguments. Launch the `west cfg_resolve -h` command to get the list and description.

## 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 9.70.2
- Keil MDK, version is 5.42a
- MCUXpresso for VS Code v25.12
- 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
<b>LPCXpresso802</b>	<b>LPC802M001JDH16,</b> <b>LPC802M001JDH20,</b> LPC802M001JHI33, LPC802M011JDH20, LPC802UK

## 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.

## Middleware

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

**FreeMASTER** FreeMASTER communication driver for 32-bit platforms.

## 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.

## 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
- 

**LPC\_ACOMP**

**[2.1.0]**

- Bug Fixes
  - Fixed one wrong enum value for the hysteresis.
  - Fixed the violations of MISRA C-2012 rules:
    - \* Rule 10.1, 17.7.

**[2.0.2]**

- Bug Fixes
  - Fixed the out-of-bounds error of Coverity caused by missing an assert sentence to avoid the return value of ACOMP\_GetInstance() exceeding the array bounds.

**[2.0.1]**

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

**[2.0.0]**

- Initial version.
-

## LPC\_ADC

### [2.6.0]

- New Features
  - Added new feature macro to distinguish whether the GPADC\_CTRL0\_GPADC\_TSAMP control bit is on the device.
  - Added new variable extendSampleTimeNumber to indicate the ADC extend sample time.
- Bugfix
  - Fixed the bug that incorrectly sets the PASS\_ENABLE bit based on the sample time setting.

### [2.5.3]

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

### [2.5.2]

- Improvements
  - Integrated different sequence's sample time numbers into one variable.
- Bug Fixes
  - Fixed violation of MISRA C-2012 rule 20.9 .

### [2.5.1]

- Bug Fixes
  - Fixed ADC conversion sequence priority misconfiguration issue in the ADC\_SetConvSeqAHighPriority() and ADC\_SetConvSeqBHighPriority() APIs.
- Improvements
  - Supported configuration ADC conversion sequence sampling time.

### [2.5.0]

- Improvements
  - Add missing parameter tag of ADC\_DoOffsetCalibration().
- Bug Fixes
  - Removed a duplicated API with typo in name: ADC\_EnableShresholdCompareInterrupt().

### [2.4.1]

- Bug Fixes
  - Enabled self-calibration after clock divider be changed to make sure the frequency update be taken.

**[2.4.0]**

- New Features
  - Added new API `ADC_DoOffsetCalibration()` which supports a specific operation frequency.
- Other Changes
  - Marked the `ADC_DoSelfCalibration(ADC_Type *base)` as deprecated.
- Bug Fixes
  - Fixed the violations of MISRA C-2012 rules:
    - \* Rule 10.1 10.3 10.4 10.7 10.8 17.7.

**[2.3.2]**

- Improvements
  - Added delay after enabling using the ADC `GPADC_CTRL0 LDO_POWER_EN` bit for JN5189/QN9090.
- New Features
  - Added support for platforms which have only one ADC sequence control/result register.

**[2.3.1]**

- Bug Fixes
  - Avoided writing ADC `STARTUP` register in `ADC_Init()`.
  - Fixed Coverity zero divider error in `ADC_DoSelfCalibration()`.

**[2.3.0]**

- Improvements
  - Updated “`ADC_Init()`”/“`ADC_GetChannelConversionResult()`” API and “`adc_resolution_t`” structure to match QN9090.
  - Added “`ADC_EnableTemperatureSensor`” API.

**[2.2.1]**

- Improvements
  - Added a brief delay in `uSec` after ADC calibration start.

**[2.2.0]**

- Improvements
  - Updated “`ADC_DoSelfCalibration`” API and “`adc_config_t`” structure to match LPC845.

**[2.1.0]**

- Improvements
  - Renamed “`ADC_EnableShresholdCompareInterrupt`” to “`ADC_EnableThresholdCompareInterrupt`”.

[2.0.0]

- Initial version.
- 

CLOCK

[2.3.4]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule 11.3.

[2.3.3]

- Improvements
  - Added lost comments for some enumerations.

[2.3.2]

- Improvements
  - Used “offsetof” macro to get the offset of the structure element from the beginning of the structure.

[2.3.1]

- Bug Fixes
  - Fixed MISRA C-2012 rule 10.1, rule 10.3, rule 15.5 and so on.

[2.3.0]

- New feature:
  - Moved SDK\_DelayAtLeastUs function from clock driver to common driver.

[2.2.0]

- Replace the delay function

[2.1.0]

- New feature
  - Adding new API CLOCK\_DelayAtLeastUs() to implement a delay function which allows users set delay in unit of microsecond.

[2.0.3]

- add api to get uart clock frequency.
- add api to set fractional multiplier value.

[2.0.2]

- some minor fixes.

[2.0.0]

- initial version.
- 

COMMON

[2.6.3]

- New Features
  - Added bit mask inversion macros to avoid type promotion.
  - Added register operation macros.
- Improvements
  - Make function `MSDK_EnableCpuCycleCounter` compatible with CMSIS-5 and CMSIS-6.
- Bug Fixes
  - Fixed build issue of CMSIS PACK BSP example caused by CMSIS 6.1 issue.

[2.6.2]

- Bug Fixes
  - Fixed violations of MISRA C-2012 rule for implicit conversions in boolean contexts

[2.6.1]

- Improvements
  - Support Cortex M23.

[2.6.0]

- Bug Fixes
  - Fix CERT-C violations.

[2.5.0]

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

[2.4.3]

- Improvements
  - Enable irqs that mount under `irqsteer` interrupt extender.

[2.4.2]

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

#### [2.4.1]

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

#### [2.4.0]

- New Features
  - Added EnableIRQWithPriority, IRQ\_SetPriority, and IRQ\_ClearPendingIRQ for ARM.
  - Added MSDK\_EnableCpuCycleCounter, MSDK\_GetCpuCycleCount for ARM.

#### [2.3.3]

- New Features
  - Added NETC into status group.

#### [2.3.2]

- Improvements
  - Make driver aarch64 compatible

#### [2.3.1]

- Bug Fixes
  - Fixed MAKE\_VERSION overflow on 16-bit platforms.

#### [2.3.0]

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

#### [2.2.10]

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

#### [2.2.9]

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

#### [2.2.8]

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

#### [2.2.7]

- Other Change
  - Added MECC status group definition.

#### [2.2.6]

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

#### [2.2.5]

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

#### [2.2.4]

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

#### [2.2.3]

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

#### [2.2.2]

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

#### [2.2.1]

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

#### [2.2.0]

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

#### [2.1.4]

- New Features
  - Added OTFAD into status group.

### [2.1.3]

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

### [2.1.2]

- Improvements
  - Add SUPPRESS\_FALL\_THROUGH\_WARNING() macro for the usage of suppressing fallthrough warning.

### [2.1.1]

- Bug Fixes
  - Deleted and optimized repeated macro.

### [2.1.0]

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

### [2.0.2]

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

### [2.0.1]

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

### [2.0.0]

- Initial version.
- 

## CRC

### [2.1.1]

- Fix MISRA issue.

**[2.1.0]**

- Add CRC\_WriteSeed function.

**[2.0.2]**

- Fix MISRA issue.

**[2.0.1]**

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

**[2.0.0]**

- Initial version.
- 

**CTIMER**

**[2.3.4]**

- Bug Fixes
  - Fixed ERRATA ERR053024 CTIMER will enter interrupt twice when function clock much slower than bus clock.

**[2.3.3]**

- Bug Fixes
  - Fix CERT INT30-C INT31-C issue.
  - Make API CTIMER\_SetupPwm and CTIMER\_UpdatePwmDutycycle return fail if pulse width register overflow.

**[2.3.2]**

- Bug Fixes
  - Clear unexpected DMA request generated by RESET\_PeripheralReset in API CTIMER\_Init to avoid trigger DMA by mistake.

**[2.3.1]**

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

**[2.3.0]**

- Improvements
  - Added the CTIMER\_SetPrescale(), CTIMER\_GetCaptureValue(), CTIMER\_EnableResetMatchChannel(), CTIMER\_EnableStopMatchChannel(), CTIMER\_EnableRisingEdgeCapture(), CTIMER\_EnableFallingEdgeCapture(), CTIMER\_SetShadowValue(), APIs Interface to reduce code complexity.

### [2.2.2]

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

### [2.2.1]

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

### [2.2.0]

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

### [2.1.0]

- Improvements
  - Added the CTIMER\_GetOutputMatchStatus() API Interface.
  - Added feature macro for FSL\_FEATURE\_CTIMER\_HAS\_NO\_CCR\_CAP2 and FSL\_FEATURE\_CTIMER\_HAS\_NO\_IR\_CR2INT.

### [2.0.3]

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

### [2.0.2]

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

### [2.0.1]

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

[2.0.0]

- Initial version.
- 

**GPIO**

[2.1.7]

- Improvements
  - Enhanced GPIO\_PinInit to enable clock internally.

[2.1.6]

- Bug Fixes
  - Clear bit before set it within GPIO\_SetPinInterruptConfig() API.

[2.1.5]

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

[2.1.4]

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

[2.1.3]

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

[2.1.2]

- Improvements
  - Removed deprecated APIs.

[2.1.1]

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

#### [2.1.0]

- New Features
  - Added GPIO initialize API.

#### [2.0.0]

- Initial version.
- 

## I2C

#### [2.2.1]

- Bug Fixes
  - Fixed coverity issues.

#### [2.2.0]

- Removed lpc\_i2c\_dma driver.

#### [2.1.0]

- Bug Fixes
  - Fixed MISRA 8.6 violations.

#### [2.0.4]

- Bug Fixes
  - Fixed wrong assignment for datasize in I2C\_InitTransferStateMachineDMA.
  - Fixed wrong working flow in I2C\_RunTransferStateMachineDMA to ensure master can work in no start flag and no stop flag mode.
  - Fixed wrong working flow in I2C\_RunTransferStateMachine and added kReceiveDataBeginState in \_i2c\_transfer\_states to ensure master can work in no start flag and no stop flag mode.
  - Fixed wrong handle state in I2C\_MasterTransferDMAHandleIRQ. After all the data has been transferred or nak is returned, handle state should be changed to idle.
  - Eliminated IAR Pa082 warning in I2C\_SlaveTransferHandleIRQ by assigning volatile variable to local variable and using local variable instead.
  - Fixed MISRA issues.
    - \* Fixed rules 4.7, 10.1, 10.3, 10.4, 11.1, 11.8, 14.4, 17.7.
- Improvements
  - Rounded up the calculated divider value in I2C\_MasterSetBaudRate.
  - Updated the I2C\_WAIT\_TIMEOUT macro to unified name I2C\_RETRY\_TIMES.

#### [2.0.3]

- Bug Fixes
  - Fixed Coverity issue of unchecked return value in I2C\_RTOS\_Transfer.

### [2.0.2]

- New Features
  - Added macro gate “FSL\_SDK\_ENABLE\_I2C\_DRIVER\_TRANSACTIONAL\_APIS” to enable/disable the transactional APIs which will help reduce the code size when no non-blocking transfer is used. Default configuration is enabled.
  - Added a control macro to enable/disable the RESET and CLOCK code in current driver.

### [2.0.1]

- Improvements
  - Added I2C\_WATI\_TIMEOUT macro to allow the user to specify the timeout times for waiting flags in functional API and blocking transfer API.

### [2.0.0]

- Initial version.
- 

## IAP

### [2.0.7]

- Bug Fixes
  - Fixed IAP\_ReinvokeISP bug that can't support UART ISP auto baud detection.

### [2.0.6]

- Bug Fixes
  - Fixed IAP\_ReinvokeISP wrong parameter setting.

### [2.0.5]

- New Feature
  - Added support config flash memory access time.

### [2.0.4]

- Bug Fixes
  - Fixed the violations of MISRA 2012 rules 9.1

### [2.0.3]

- New Features
  - Added support for LPC 845's FAIM operation.
  - Added support for LPC 80x's fixed reference clock for flash controller.
  - Added support for LPC 5411x's Read UID command useless situation.
- Improvements
  - Improved the document and code structure.

- Bug Fixes
  - Fixed the violations of MISRA 2012 rules:
    - \* Rule 10.1 10.3 10.4 17.7

#### [2.0.2]

- New Features
  - Added an API to read generated signature.
- Bug Fixes
  - Fixed the incorrect board support of IAP\_ExtendedFlashSignatureRead().

#### [2.0.1]

- New Features
  - Added an API to read factory settings for some calibration registers.
- Improvements
  - Updated the size of result array in part APIs.

#### [2.0.0]

- Initial version.
- 

### IOCON

#### [2.0.2]

- Bug Fixes
  - Fixed MISRA-C 2012 violations.

#### [2.0.1]

- Bug Fixes
  - Fixed out-of-range issue of the IOCON mode function when enabling DAC.

#### [2.0.0]

- Initial version.
- 

### MRT

#### [2.0.5]

- Bug Fixes
  - Fixed CERT INT31-C violations.

#### [2.0.4]

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

#### [2.0.3]

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

#### [2.0.2]

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

#### [2.0.1]

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

#### [2.0.0]

- Initial version.
- 

## PINT

#### [2.3.0]

- Improvements
  - Add API PINT\_EnableInterruptByIndex and PINT\_DisableInterruptByIndex to provide more granular interrupt control.

#### [2.2.0]

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

#### [2.1.13]

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

**[2.1.12]**

- Bug Fixes
  - Fixed coverity issue.

**[2.1.11]**

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

**[2.1.10]**

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

**[2.1.9]**

- Bug Fixes
  - Fixed MISRA-2012 rule 8.4.

**[2.1.8]**

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

**[2.1.7]**

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

**[2.1.6]**

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

**[2.1.5]**

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

#### [2.1.4]

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

#### [2.1.3]

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

#### [2.1.2]

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

#### [2.1.1]

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

#### [2.1.0]

- Added PINT\_EnableCallbackByIndex/PINT\_DisableCallbackByIndex APIs to enable/disable callback by index.

#### [2.0.2]

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

#### [2.0.1]

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

[2.0.0]

- Initial version.
- 

**POWER**

[2.1.0]

- New features
  - Added BOD control APIs.

[2.0.4]

- Bug Fixes
  - Fixed the typo “Enbale”, correcting it as “Enable”.

[2.0.3]

- Bug Fixes
  - Fixed doxygen warnings(remove wrong character in annotation).

[2.0.2]

- New Features
  - Added the Enable/DisableDeepSleepIRQ() to enable/disable pin wake up.

[2.0.1]

- Improvements
  - Updated power drive to support PMU.

[2.0.0]

- initial version.
- 

**RESET**

[2.4.0]

- Improvements
  - Add RESET\_ReleasePeripheralReset API.

[2.0.1]

- Update component full\_name to “Reset Driver”.

[2.0.0]

- initial version.
- 

SPI

[2.0.8]

- Bug Fixes
  - Fixed coverity issue.

[2.0.7]

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

[2.0.6]

- Improvements
  - Changed SPI\_DUMMYDATA to 0x00.

[2.0.5]

- Bug Fixes
  - Fixed bug that the transfer configuration does not take effect after the first transfer.

[2.0.4]

- Bug Fixes
  - Fixed the issue that when transfer finish callback is invoked TX data is not sent to bus yet.

[2.0.3]

- Improvements
  - Added timeout mechanism when waiting certain states in transfer driver.
  - Fixed MISRA 10.4 issue.

[2.0.2]

- Bug Fixes
  - Fixed Coverity issue of incrementing null pointer in SPI\_MasterTransferNonBlocking.
  - Fixed MISRA issues.
    - \* Fixed rules 10.1, 10.3, 10.4, 10.6, 14.4.
- New Features
  - Added enumeration for dataWidth.

### [2.0.1]

- Bug Fixes
  - Added wait mechanism in SPI\_MasterTransferBlocking() API, which checks if master SPI becomes IDLE when the EOT bit is set before returning. This confirms that all data will be sent out by SPI master.
  - Fixed the bug that the EOT bit couldn't be set when only one frame was sent in polling mode and interrupt transfer mode.
- New Features
  - Added macro gate “FSL\_SDK\_ENABLE\_SPI\_DRIVER\_TRANSACTIONAL\_APIS” to enable/disable the transactional APIs, which helps reduce the code size when no non-blocking transfer is used. Enabled default configuration.
  - Added a control macro to enable/disable the RESET and CLOCK code in current driver.

### [2.0.0]

- Initial version.
- 

## SWM

### [2.1.2]

- Improvements
  - Reduce RAM footprint.

### [2.1.1]

- Bug Fixes
  - MISRA C-2012 issue fixed: rule 10.1 and 10.3.

### [2.1.0]

- New Features
  - Supported Flextimer function pin assign.

### [2.0.2]

- Bug Fixes
  - MISRA C-2012 issue fixed: rule 14.3.

### [2.0.1]

- Bug Fixes
  - MISRA C-2012 issue fixed: rule 10.1, 10.3, and 10.4.

#### [2.0.0]

- Initial version.
  - The API `SWM_SetFixedMovablePinSelect()` is targeted at the device that has `PINASSIGN-FIXED0` register, such as `LPC804`.
- 

### SYSCON

#### [2.0.2]

- Bug Fixes
  - Fixed CERT-C violations.

#### [2.0.1]

- Bug Fixes
  - Fixed issue for MISRA-2012 check.
    - \* Fixed rule 10.4.

#### [2.0.0]

- Initial version.
- 

### USART

#### [2.5.2]

- Improvements
  - Fixed coverity issues.

#### [2.5.1]

- Improvements
  - Fixed doxygen warning in `USART_SetRxIdleTimeout`.

#### [2.5.0]

- New Features
  - Supported new feature of rx idle timeout.

#### [2.4.0]

- Improvements
  - Used separate data for TX and RX in `usart_transfer_t`.
- Bug Fixes
  - Fixed bug that when ring buffer is used, if some data is received in ring buffer first before calling `USART_TransferReceiveNonBlocking`, the received data count returned by `USART_TransferGetReceiveCount` is wrong.

### [2.3.0]

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

### [2.2.0]

- Improvements
  - Added timeout mechanism when waiting for certain states in transfer driver.
  - Fixed MISRA 10.4 issues.

### [2.1.1]

- Bug Fixes
  - Fixed the bug that in `USART_SetBaudRate` `best_diff` rather than `diff` should be used to compare with calculated baudrate.
  - Eliminated IAR pa082 warnings from `USART_TransferGetRxRingBufferLength` and `USART_TransferHandleIRQ`.
  - Fixed MISRA issues.
- Improvements
  - Rounded up the calculated `sbr` value in `USART_SetBaudRate` to achieve more accurate baudrate setting.
  - Modified `USART_ReadBlocking` so that if more than one receiver errors occur, all status flags will be cleared and the most severe error status will be returned.

### [2.1.0]

- New Features
  - Added new APIs to allow users to configure the USART continuous SCLK feature in synchronous mode transfer.

### [2.0.1]

- Bug Fixes
  - Fixed the repeated reading issue of the STAT register while dealing with the IRQ routine.
- New Features
  - Added macro gate “`FSL_SDK_ENABLE_USART_DRIVER_TRANSACTIONAL_APIS`” to enable/disable the transactional APIs, which helps reduce the code size when no non-blocking transfer is used. Enabled default configuration.
  - Added a control macro to enable/disable the RESET and CLOCK code in current driver.
  - Added macro switch gate “`FSL_SDK_USART_DRIVER_ENABLE_BAUDRATE_AUTO_GENERATE`” to enable/disable the baud rate to generate automatically. Disabling this feature will help reduce the code size to a certain degree. Default configuration enables auto generating of baud rate.
  - Added the check of baud rate while initializing the USART. If the baud rate calculated is not precise, the software assertion will be triggered.

- Added a new API to allow users to enable the CTS, which determines whether CTS is used for flow control.

#### [2.0.0]

- Initial version.
- 

### WKT

#### [2.0.2]

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

#### [2.0.1]

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

#### [2.0.0]

- Initial version.
- 

### WWDT

#### [2.1.10]

- Bug Fixes
  - Check WWDT\_RSTS instead of FSL\_FEATURE\_WWDT\_HAS\_NO\_RESET to determine whether the peripheral can be reset.

#### [2.1.9]

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

#### [2.1.8]

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

#### [2.1.7]

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

#### [2.1.5]

- Bug Fixes
  - deprecated a unusable API in WWWDT driver.
    - \* WWDT\_Disable

#### [2.1.4]

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

#### [2.1.3]

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

#### [2.1.2]

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

#### [2.1.1]

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

#### [2.1.0]

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

[2.0.0]

- Initial version.
- 

## 1.6 Driver API Reference Manual

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

[LPC802](#)

## 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 FreeMASTER

[freemaster](#)



# Chapter 2

## LPC802

### 2.1 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\_Ram0

Clock gate name: Ram0.

enumerator kCLOCK\_Flash

Clock gate name: Flash.

enumerator kCLOCK\_I2c0

Clock gate name: I2c0.

enumerator kCLOCK\_Gpio0

Clock gate name: Gpio0.

enumerator kCLOCK\_Swm

Clock gate name: Swm.

enumerator kCLOCK\_Wkt

Clock gate name: Wkt.

enumerator kCLOCK\_Mrt

Clock gate name: Mrt.

enumerator kCLOCK\_Spi0

Clock gate name: Spi0.

enumerator kCLOCK\_Crc

Clock gate name: Crc.

enumerator kCLOCK\_Uart0

Clock gate name: Uart0.

enumerator kCLOCK\_Uart1  
 Clock gate name: Uart1.

enumerator kCLOCK\_Wwdt  
 Clock gate name: Wwdt.

enumerator kCLOCK\_Iocon  
 Clock gate name: Iocon.

enumerator kCLOCK\_Acmp  
 Clock gate name: Acmp.

enumerator kCLOCK\_Adc  
 Clock gate name: Adc.

enumerator kCLOCK\_Ctimer0  
 Clock gate name: Ctimer0.

enumerator kCLOCK\_GpioInt  
 Clock gate name: GpioInt.

enum \_clock\_name

Clock name used to get clock frequency.

*Values:*

enumerator kCLOCK\_CoreSysClk  
 Cpu/AHB/AHB matrix/Memories,etc

enumerator kCLOCK\_MainClk  
 Main clock

enumerator kCLOCK\_Fro  
 FRO18/24/30

enumerator kCLOCK\_FroDiv  
 FRO div clock

enumerator kCLOCK\_ExtClk  
 External Clock

enumerator kCLOCK\_LPOsc  
 Low power Oscillator

enumerator kCLOCK\_Frg  
 fractional rate0

enum \_clock\_select

Clock Mux Switches CLK\_MUX\_DEFINE(reg, mux) reg is used to define the mux register mux is used to define the mux value.

*Values:*

enumerator kADC\_Clk\_From\_Fro  
 Mux ADC\_Clk from Fro.

enumerator kADC\_Clk\_From\_ClkIn  
 Mux ADC\_Clk from ClkIn.

enumerator kUART0\_Clk\_From\_Fro  
 Mux UART0\_Clk from Fro.

enumerator kUART0\_Clk\_From\_MainClk  
 Mux UART0\_Clk from MainClk.

enumerator kUART0\_Clk\_From\_Frg0Clk  
Mux UART0\_Clk from Frg0Clk.

enumerator kUART0\_Clk\_From\_Fro\_Div  
Mux UART0\_Clk from Fro\_Div.

enumerator kUART1\_Clk\_From\_Fro  
Mux UART1\_Clk from Fro.

enumerator kUART1\_Clk\_From\_MainClk  
Mux UART1\_Clk from MainClk.

enumerator kUART1\_Clk\_From\_Frg0Clk  
Mux UART1\_Clk from Frg0Clk.

enumerator kUART1\_Clk\_From\_Fro\_Div  
Mux UART1\_Clk from Fro\_Div.

enumerator kI2C\_Clk\_From\_Fro  
Mux I2C\_Clk from Fro.

enumerator kI2C\_Clk\_From\_MainClk  
Mux I2C\_Clk from MainClk.

enumerator kI2C\_Clk\_From\_Frg0Clk  
Mux I2C\_Clk from Frg0Clk.

enumerator kI2C\_Clk\_From\_Fro\_Div  
Mux I2C\_Clk from Fro\_Div.

enumerator kSPI\_Clk\_From\_Fro  
Mux SPI\_Clk from Fro.

enumerator kSPI\_Clk\_From\_MainClk  
Mux SPI\_Clk from MainClk.

enumerator kSPI\_Clk\_From\_Frg0Clk  
Mux SPI\_Clk from Frg0Clk.

enumerator kSPI\_Clk\_From\_Fro\_Div  
Mux SPI\_Clk from Fro\_Div.

enumerator kFRG0\_Clk\_From\_Fro  
Mux FRG0\_Clk from Fro.

enumerator kFRG0\_Clk\_From\_MainClk  
Mux FRG0\_Clk from MainClk.

enumerator kCLKOUT\_From\_Fro  
Mux CLKOUT from Fro.

enumerator kCLKOUT\_From\_MainClk  
Mux CLKOUT from MainClk.

enumerator kCLKOUT\_From\_ExtClk  
Mux CLKOUT from ExtClk.

enumerator kCLKOUT\_From\_WdtOsc  
Mux clock out from WdtOsc.

enum \_clock\_divider  
Clock divider.

*Values:*

enumerator kCLOCK\_DivAdcClk  
Adc Clock Divider.

enumerator kCLOCK\_DivClkOut  
Clock out divider.

enum \_clock\_fro\_src  
fro output frequency source definition

*Values:*

enumerator kCLOCK\_FroSrcLpwrBootValue  
fro source from the fro oscillator divided by low power boot value

enum \_clock\_fro\_osc\_freq  
fro oscillator output frequency value definition

*Values:*

enumerator kCLOCK\_FroOscOut18M  
FRO oscillator output 18M

enumerator kCLOCK\_FroOscOut24M  
FRO oscillator output 24M

enumerator kCLOCK\_FroOscOut30M  
FRO oscillator output 30M

enum \_clock\_main\_clk\_src  
Main clock source definition.

*Values:*

enumerator kCLOCK\_MainClkSrcFro  
main clock source from FRO

enumerator kCLOCK\_MainClkSrcExtClk  
main clock source from Ext clock

enumerator kCLOCK\_MainClkSrcLPOsc  
main clock source from watchdog oscillator

enumerator kCLOCK\_MainClkSrcFroDiv  
main clock source from FRO Div

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\_select clock\_select\_t  
Clock Mux Switches CLK\_MUX\_DEFINE(reg, mux) reg is used to define the mux register mux is used to define the mux value.

typedef enum \_clock\_divider clock\_divider\_t  
Clock divider.

typedef enum \_clock\_fro\_src clock\_fro\_src\_t  
fro output frequency source definition

typedef enum \_clock\_fro\_osc\_freq clock\_fro\_osc\_freq\_t  
fro oscillator output frequency value definition

```
typedef enum _clock_main_clk_src clock_main_clk_src_t
```

Main clock source definition.

```
volatile uint32_t g_Wdt_Osc_Freq
```

watchdog oscillator clock frequency.

This variable is used to store the watchdog oscillator frequency which is set by `CLOCK_InitWdtOsc`, and it is returned by `CLOCK_GetWdtOscFreq`.

```
volatile uint32_t g_Ext_Clk_Freq
```

external clock frequency.

This variable is used to store the external clock frequency which is include external oscillator clock and external clk in clock frequency value, it is set by `CLOCK_InitExtClkin` when CLK IN is used as external clock or by `CLOCK_InitSysOsc` when external oscillator is used as external clock ,and it is returned by `CLOCK_GetExtClkFreq`.

```
FSL_CLOCK_DRIVER_VERSION
```

CLOCK driver version 2.3.4.

```
SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY
```

```
CLOCK_FRO_SETTING_API_ROM_ADDRESS
```

FRO clock setting API address in ROM.

```
CLOCK_FAIM_BASE
```

FAIM base address.

```
ADC_CLOCKS
```

Clock ip name array for ADC.

```
ACMP_CLOCKS
```

Clock ip name array for ACMP.

```
DAC_CLOCKS
```

Clock ip name array for DAC.

```
SWM_CLOCKS
```

Clock ip name array for SWM.

```
ROM_CLOCKS
```

Clock ip name array for ROM.

```
SRAM_CLOCKS
```

Clock ip name array for SRAM.

```
IOCON_CLOCKS
```

Clock ip name array for IOCON.

```
GPIO_CLOCKS
```

Clock ip name array for GPIO.

```
GPIO_INT_CLOCKS
```

Clock ip name array for GPIO\_INT.

```
DMA_CLOCKS
```

Clock ip name array for DMA.

```
CRC_CLOCKS
```

Clock ip name array for CRC.

```
WWDT_CLOCKS
```

Clock ip name array for WWDT.

SCT\_CLOCKS

Clock ip name array for SCT0.

I2C\_CLOCKS

Clock ip name array for I2C.

USART\_CLOCKS

Clock ip name array for I2C.

SPI\_CLOCKS

Clock ip name array for SPI.

CAPT\_CLOCKS

Clock ip name array for CAPT.

CTIMER\_CLOCKS

Clock ip name array for CTIMER.

MTB\_CLOCKS

Clock ip name array for MTB.

MRT\_CLOCKS

Clock ip name array for MRT.

WKT\_CLOCKS

Clock ip name array for WKT.

CLK\_GATE\_DEFINE(reg, bit)

Internal used Clock definition only.

CLK\_GATE\_GET\_REG(x)

CLK\_GATE\_GET\_BITS\_SHIFT(x)

CLK\_MUX\_DEFINE(reg, mux)

CLK\_MUX\_GET\_REG(x)

CLK\_MUX\_GET\_MUX(x)

CLK\_MAIN\_CLK\_MUX\_DEFINE(preMux, mux)

CLK\_MAIN\_CLK\_MUX\_GET\_PRE\_MUX(x)

CLK\_MAIN\_CLK\_MUX\_GET\_MUX(x)

CLK\_DIV\_DEFINE(reg)

CLK\_DIV\_GET\_REG(x)

CLK\_WDT\_OSC\_DEFINE(freq, regValue)

CLK\_WDT\_OSC\_GET\_FREQ(x)

CLK\_WDT\_OSC\_GET\_REG(x)

CLK\_FRG\_DIV\_REG\_MAP(base)

CLK\_FRG\_MUL\_REG\_MAP(base)

CLK\_FRG\_SEL\_REG\_MAP(base)

SYS\_AHB\_CLK\_CTRL0

static inline void CLOCK\_EnableClock(*clock\_ip\_name\_t* clk)

static inline void CLOCK\_DisableClock(*clock\_ip\_name\_t* clk)

static inline void CLOCK\_Select(*clock\_select\_t* sel)

static inline void CLOCK\_SetClkDivider(*clock\_divider\_t* name, uint32\_t value)

static inline uint32\_t CLOCK\_GetClkDivider(*clock\_divider\_t* name)

static inline void CLOCK\_SetCoreSysClkDiv(uint32\_t value)

void CLOCK\_SetMainClkSrc(*clock\_main\_clk\_src\_t* src)

Set main clock reference source.

#### Parameters

- src – Reference clock\_main\_clk\_src\_t to set the main clock source.

static inline void CLOCK\_SetFRGClkMul(uint32\_t \*base, uint32\_t mul)

uint32\_t CLOCK\_GetFRGClkFreq(void)

Return Frequency of FRG0 Clock.

#### Returns

Frequency of FRG0 Clock.

uint32\_t CLOCK\_GetMainClkFreq(void)

Return Frequency of Main Clock.

#### Returns

Frequency of Main Clock.

uint32\_t CLOCK\_GetFroFreq(void)

Return Frequency of FRO.

#### Returns

Frequency of FRO.

static inline uint32\_t CLOCK\_GetCoreSysClkFreq(void)

Return Frequency of core.

#### Returns

Frequency of core.

uint32\_t CLOCK\_GetClockOutClkFreq(void)

Return Frequency of ClockOut.

#### Returns

Frequency of ClockOut

uint32\_t CLOCK\_GetUart0ClkFreq(void)

Get UART0 frequency.

#### Return values

UART0 – frequency value.

uint32\_t CLOCK\_GetUart1ClkFreq(void)

Get UART1 frequency.

#### Return values

UART1 – frequency value.

uint32\_t CLOCK\_GetFreq(*clock\_name\_t* clockName)

Return Frequency of selected clock.

**Returns**

Frequency of selected clock

static inline uint32\_t CLOCK\_GetLPOscFreq(void)

Get watch dog OSC frequency.

**Return values**

watch – dog OSC frequency value.

static inline uint32\_t CLOCK\_GetExtClkFreq(void)

Get external clock frequency.

**Return values**

external – clock frequency value.

bool CLOCK\_SetFRGClkFreq(uint32\_t freq)

Set FRG0 output frequency.

**Parameters**

- target (freq,) – output frequency,  $\text{freq} < \text{input}$  and  $(\text{input} / \text{freq}) < 2$  should be satisfy.

**Return values**

true – - successfully, false - input argument is invalid.

void CLOCK\_InitExtClkin(uint32\_t clkInFreq)

Init external CLK IN, select the CLKIN as the external clock source.

**Parameters**

- clkInFreq – external clock in frequency.

static inline void CLOCK\_DeinitLPOsc(void)

Deinit watch dog OSC.

**Parameters**

- config – oscillator configuration.

void CLOCK\_SetFroOscFreq(*clock\_fro\_osc\_freq\_t* freq)

Set FRO oscillator output frequency. Initialize the FRO clock to given frequency (18, 24 or 30 MHz).

**Parameters**

- please (freq,) – reference *clock\_fro\_osc\_freq\_t* definition, frequency must be one of 18000, 24000 or 30000 KHz.

## 2.2 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.

## 2.3 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 2.3.4

```
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

## 2.4 I2C: Inter-Integrated Circuit Driver

### 2.5 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\_Addr\_Nak

NAK received during the address probe.

enumerator kStatus\_I2C\_Timeout

Timeout polling status flags.

I2C\_RETRY\_TIMES

Retry times for waiting flag.

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

## 2.6 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.

```
static inline 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\\_master\\_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_MasterClearStatusFlags(I2C_Type *base, uint32_t statusMask)
```

Clears the I2C master status flag state.

The following status register flags can be cleared:

- kI2C\_MasterArbitrationLostFlag
- kI2C\_MasterStartStopErrorFlag

Attempts to clear other flags has no effect.

#### See also:

[\\_i2c\\_master\\_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_master_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 master interrupt requests.

**Parameters**

- `base` – The I2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i2c_master_flags` 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 master interrupt requests.

**Parameters**

- `base` – The I2C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_i2c_master_flags` 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 master interrupt requests.

**Parameters**

- `base` – The I2C peripheral base address.

**Returns**

A bitmask composed of `_i2c_master_flags` 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.

```
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.

`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 – Abort failure due to flags polling timeout.

```
void I2C_MasterTransferHandleIRQ(I2C_Type *base, void *i2cHandle)
```

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.

- `i2cHandle` – Pointer to the I2C master driver handle `i2c_master_handle_t`.

enum `_i2c_master_flags`  
I2C master peripheral 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.

enumerator `kI2C_MasterArbitrationLostFlag`

The arbitration of the bus was lost. There was collision on the bus

enumerator `kI2C_MasterStartStopErrorFlag`

There was an error during start or stop phase of the transaction.

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.

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.

uint16\_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.

*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.

## 2.7 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, void *i2cHandle)
```

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.
- `i2cHandle` – Pointer to `i2c_slave_handle_t` structure which stores the transfer state.

enum `_i2c_slave_flags`  
I2C slave peripheral flags.

---

**Note:** These enums are meant to be OR'd together to form a bit mask.

---

*Values:*

enumerator `kI2C_SlavePendingFlag`  
The I2C module is waiting for software interaction.

enumerator `kI2C_SlaveNotStretching`  
Indicates whether the slave is currently stretching clock (0 = yes, 1 = no).

enumerator `kI2C_SlaveSelected`  
Indicates whether the slave is selected by an address match.

enumerator `kI2C_SlaveDeselected`  
Indicates that slave was previously deselected (deselect event took place, w1c).

enum `_i2c_slave_address_register`  
I2C slave address register.

*Values:*

enumerator `kI2C_SlaveAddressRegister0`  
Slave Address 0 register.

enumerator `kI2C_SlaveAddressRegister1`  
Slave Address 1 register.

enumerator `kI2C_SlaveAddressRegister2`  
Slave Address 2 register.

enumerator `kI2C_SlaveAddressRegister3`  
Slave Address 3 register.

enum `_i2c_slave_address_qual_mode`  
I2C slave address match options.

*Values:*

enumerator `kI2C_QualModeMask`  
The `SLVQUAL0` field (`qualAddress`) is used as a logical mask for matching address0.

enumerator `kI2C_QualModeExtend`  
The `SLVQUAL0` (`qualAddress`) field is used to extend address 0 matching in a range of addresses.

enum `_i2c_slave_bus_speed`  
I2C slave bus speed options.

*Values:*

enumerator `kI2C_SlaveStandardMode`

enumerator `kI2C_SlaveFastMode`

enumerator `kI2C_SlaveFastModePlus`

enumerator `kI2C_SlaveHsMode`

`enum _i2c_slave_transfer_event`

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

---

**Note:** These enumerations are meant to be OR'd together to form a bit mask of events.

---

*Values:*

enumerator `kI2C_SlaveAddressMatchEvent`

Received the slave address after a start or repeated start.

enumerator `kI2C_SlaveTransmitEvent`

Callback is requested to provide data to transmit (slave-transmitter role).

enumerator `kI2C_SlaveReceiveEvent`

Callback is requested to provide a buffer in which to place received data (slave-receiver role).

enumerator `kI2C_SlaveCompletionEvent`

All data in the active transfer have been consumed.

enumerator `kI2C_SlaveDeselectedEvent`

The slave function has become deselected (SLVSEL flag changing from 1 to 0).

enumerator `kI2C_SlaveAllEvents`

Bit mask of all available events.

`enum _i2c_slave_fsm`

I2C slave software finite state machine states.

*Values:*

enumerator `kI2C_SlaveFsmAddressMatch`

enumerator `kI2C_SlaveFsmReceive`

enumerator `kI2C_SlaveFsmTransmit`

`typedef enum _i2c_slave_address_register i2c_slave_address_register_t`

I2C slave address register.

`typedef struct _i2c_slave_address i2c_slave_address_t`

Data structure with 7-bit Slave address and Slave address disable.

`typedef enum _i2c_slave_address_qual_mode i2c_slave_address_qual_mode_t`

I2C slave address match options.

`typedef enum _i2c_slave_bus_speed i2c_slave_bus_speed_t`

I2C slave bus speed options.

`typedef struct _i2c_slave_config i2c_slave_config_t`

Structure with settings to initialize the I2C slave module.

This structure holds configuration settings for the I2C slave peripheral. To initialize this structure to reasonable defaults, call the `I2C_SlaveGetDefaultConfig()` function and pass a pointer to your configuration structure instance.

The configuration structure can be made constant so it resides in flash.

```
typedef enum _i2c_slave_transfer_event i2c_slave_transfer_event_t
```

Set of events sent to the callback for non blocking slave transfers.

These event enumerations are used for two related purposes. First, a bit mask created by OR'ing together events is passed to `I2C_SlaveTransferNonBlocking()` in order to specify which events to enable. Then, when the slave callback is invoked, it is passed the current event through its *transfer* parameter.

---

**Note:** These enumerations are meant to be OR'd together to form a bit mask of events.

---

```
typedef struct _i2c_slave_handle i2c_slave_handle_t
```

I2C slave handle typedef.

```
typedef struct _i2c_slave_transfer i2c_slave_transfer_t
```

I2C slave transfer structure.

```
typedef void (*i2c_slave_transfer_callback_t)(I2C_Type *base, volatile i2c_slave_transfer_t *transfer, void *userData)
```

Slave event callback function pointer type.

This callback is used only for the slave non-blocking transfer API. To install a callback, use the `I2C_SlaveSetCallback()` function after you have created a handle.

**Param base**

Base address for the I2C instance on which the event occurred.

**Param transfer**

Pointer to transfer descriptor containing values passed to and/or from the callback.

**Param userData**

Arbitrary pointer-sized value passed from the application.

```
typedef enum _i2c_slave_fsm i2c_slave_fsm_t
```

I2C slave software finite state machine states.

```
typedef void (*i2c_isr_t)(I2C_Type *base, void *i2cHandle)
```

Typedef for 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 &lt;fsl\_i2c.h&gt; 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.

## 2.8 IAP: In Application Programming Driver

status\_t IAP\_ReadPartID(uint32\_t \*partID)

Read part identification number.

This function is used to read the part identification number.

### Parameters

- partID – Address to store the part identification number.

### Return values

kStatus\_IAP\_Success – Api has been executed successfully.

status\_t IAP\_ReadBootCodeVersion(uint32\_t \*bootCodeVersion)

Read boot code version number.

This function is used to read the boot code version number.

note Boot code version is two 32-bit words. Word 0 is the major version, word 1 is the minor version.

### Parameters

- bootCodeVersion – Address to store the boot code version.

### Return values

kStatus\_IAP\_Success – Api has been executed successfully.

```
void IAP_ReinvokeISP(uint8_t ispType, uint32_t *status)
```

Reinvoke ISP.

This function is used to invoke the boot loader in ISP mode. It maps boot vectors and configures the peripherals for ISP.

note The error response will be returned when IAP is disabled or an invalid ISP type selection appears. The call won't return unless an error occurs, so there can be no status code.

#### Parameters

- `ispType` – ISP type selection.
- `status` – store the possible status.

#### Return values

`kStatus_IAP_ReinvokeISPConfig` – reinvoke configuration error.

```
status_t IAP_ReadUniqueID(uint32_t *uniqueID)
```

Read unique identification.

This function is used to read the unique id.

#### Parameters

- `uniqueID` – store the uniqueID.

#### Return values

`kStatus_IAP_Success` – Api has been executed successfully.

```
status_t IAP_PrepareSectorForWrite(uint32_t startSector, uint32_t endSector)
```

Prepare sector for write operation.

This function prepares sector(s) for write/erase operation. This function must be called before calling the `IAP_CopyRamToFlash()` or `IAP_EraseSector()` or `IAP_ErasePage()` function. The end sector number must be greater than or equal to the start sector number.

#### Parameters

- `startSector` – Start sector number.
- `endSector` – End sector number.

#### Return values

- `kStatus_IAP_Success` – Api has been executed successfully.
- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_InvalidSector` – Sector number is invalid or end sector number is greater than start sector number.
- `kStatus_IAP_Busy` – Flash programming hardware interface is busy.

```
status_t IAP_CopyRamToFlash(uint32_t dstAddr, uint32_t *srcAddr, uint32_t numBytes,
                           uint32_t systemCoreClock)
```

Copy RAM to flash.

This function programs the flash memory. Corresponding sectors must be prepared via `IAP_PrepareSectorForWrite` before calling this function.

#### Parameters

- `dstAddr` – Destination flash address where data bytes are to be written, the address should be multiples of `FSL_FEATURE_SYSCON_FLASH_PAGE_SIZE_BYTES` boundary.

- `srcAddr` – Source ram address from where data bytes are to be read.
- `numOfBytes` – Number of bytes to be written, it should be multiples of `FSL_FEATURE_SYSCON_FLASH_PAGE_SIZE_BYTES`, and ranges from `FSL_FEATURE_SYSCON_FLASH_PAGE_SIZE_BYTES` to `FSL_FEATURE_SYSCON_FLASH_SECTOR_SIZE_BYTES`.
- `systemCoreClock` – `SystemCoreClock` in Hz. It is converted to KHz before calling the rom IAP function. When the flash controller has a fixed reference clock, this parameter is bypassed.

#### Return values

- `kStatus_IAP_Success` – Api has been executed successfully.
- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_SrcAddrError` – Source address is not on word boundary.
- `kStatus_IAP_DstAddrError` – Destination address is not on a correct boundary.
- `kStatus_IAP_SrcAddrNotMapped` – Source address is not mapped in the memory map.
- `kStatus_IAP_DstAddrNotMapped` – Destination address is not mapped in the memory map.
- `kStatus_IAP_CountError` – Byte count is not multiple of 4 or is not a permitted value.
- `kStatus_IAP_NotPrepared` – Command to prepare sector for write operation has not been executed.
- `kStatus_IAP_Busy` – Flash programming hardware interface is busy.

`status_t IAP_EraseSector(uint32_t startSector, uint32_t endSector, uint32_t systemCoreClock)`

Erase sector.

This function erases sector(s). The end sector number must be greater than or equal to the start sector number.

#### Parameters

- `startSector` – Start sector number.
- `endSector` – End sector number.
- `systemCoreClock` – `SystemCoreClock` in Hz. It is converted to KHz before calling the rom IAP function. When the flash controller has a fixed reference clock, this parameter is bypassed.

#### Return values

- `kStatus_IAP_Success` – Api has been executed successfully.
- `kStatus_IAP_NoPower` – Flash memory block is powered down.
- `kStatus_IAP_NoClock` – Flash memory block or controller is not clocked.
- `kStatus_IAP_InvalidSector` – Sector number is invalid or end sector number is greater than start sector number.
- `kStatus_IAP_NotPrepared` – Command to prepare sector for write operation has not been executed.
- `kStatus_IAP_Busy` – Flash programming hardware interface is busy.

*status\_t* IAP\_ErasePage(uint32\_t startPage, uint32\_t endPage, uint32\_t systemCoreClock)

Erase page.

This function erases page(s). The end page number must be greater than or equal to the start page number.

#### Parameters

- startPage – Start page number.
- endPage – End page number.
- systemCoreClock – SystemCoreClock in Hz. It is converted to KHz before calling the rom IAP function. When the flash controller has a fixed reference clock, this parameter is bypassed.

#### Return values

- kStatus\_IAP\_Success – Api has been executed successfully.
- kStatus\_IAP\_NoPower – Flash memory block is powered down.
- kStatus\_IAP\_NoClock – Flash memory block or controller is not clocked.
- kStatus\_IAP\_InvalidSector – Page number is invalid or end page number is greater than start page number.
- kStatus\_IAP\_NotPrepared – Command to prepare sector for write operation has not been executed.
- kStatus\_IAP\_Busy – Flash programming hardware interface is busy.

*status\_t* IAP\_BlankCheckSector(uint32\_t startSector, uint32\_t endSector)

Blank check sector(s)

Blank check single or multiples sectors of flash memory. The end sector number must be greater than or equal to the start sector number. It can be used to verify the sector erasure after IAP\_EraseSector call.

#### Parameters

- startSector – Start sector number.
- endSector – End sector number.

#### Return values

- kStatus\_IAP\_Success – One or more sectors are in erased state.
- kStatus\_IAP\_NoPower – Flash memory block is powered down.
- kStatus\_IAP\_NoClock – Flash memory block or controller is not clocked.
- kStatus\_IAP\_SectorNotblank – One or more sectors are not blank.

*status\_t* IAP\_Compare(uint32\_t dstAddr, uint32\_t \*srcAddr, uint32\_t numOfBytes)

Compare memory contents of flash with ram.

This function compares the contents of flash and ram. It can be used to verify the flash memory contents after IAP\_CopyRamToFlash call.

#### Parameters

- dstAddr – Destination flash address.
- srcAddr – Source ram address.
- numOfBytes – Number of bytes to be compared.

#### Return values

- kStatus\_IAP\_Success – Contents of flash and ram match.

- kStatus\_IAP\_NoPower – Flash memory block is powered down.
- kStatus\_IAP\_NoClock – Flash memory block or controller is not clocked.
- kStatus\_IAP\_AddrError – Address is not on word boundary.
- kStatus\_IAP\_AddrNotMapped – Address is not mapped in the memory map.
- kStatus\_IAP\_CountError – Byte count is not multiple of 4 or is not a permitted value.
- kStatus\_IAP\_CompareError – Destination and source memory contents do not match.

FSL\_IAP\_DRIVER\_VERSION

iap status codes.

*Values:*

enumerator kStatus\_IAP\_Success

Api is executed successfully

enumerator kStatus\_IAP\_InvalidCommand

Invalid command

enumerator kStatus\_IAP\_SrcAddrError

Source address is not on word boundary

enumerator kStatus\_IAP\_DstAddrError

Destination address is not on a correct boundary

enumerator kStatus\_IAP\_SrcAddrNotMapped

Source address is not mapped in the memory map

enumerator kStatus\_IAP\_DstAddrNotMapped

Destination address is not mapped in the memory map

enumerator kStatus\_IAP\_CountError

Byte count is not multiple of 4 or is not a permitted value

enumerator kStatus\_IAP\_InvalidSector

Sector/page number is invalid or end sector/page number is greater than start sector/page number

enumerator kStatus\_IAP\_SectorNotblank

One or more sectors are not blank

enumerator kStatus\_IAP\_NotPrepared

Command to prepare sector for write operation has not been executed

enumerator kStatus\_IAP\_CompareError

Destination and source memory contents do not match

enumerator kStatus\_IAP\_Busy

Flash programming hardware interface is busy

enumerator kStatus\_IAP\_ParamError

Insufficient number of parameters or invalid parameter

enumerator kStatus\_IAP\_AddrError

Address is not on word boundary

enumerator kStatus\_IAP\_AddrNotMapped  
Address is not mapped in the memory map

enumerator kStatus\_IAP\_NoPower  
Flash memory block is powered down

enumerator kStatus\_IAP\_NoClock  
Flash memory block or controller is not clocked

enumerator kStatus\_IAP\_ReinvokeISPConfig  
Reinvoke configuration error

enum \_iap\_commands

iap command codes.

*Values:*

enumerator kIapCmd\_IAP\_ReadFactorySettings  
Read the factory settings

enumerator kIapCmd\_IAP\_PrepareSectorforWrite  
Prepare Sector for write

enumerator kIapCmd\_IAP\_CopyRamToFlash  
Copy RAM to flash

enumerator kIapCmd\_IAP\_EraseSector  
Erase Sector

enumerator kIapCmd\_IAP\_BlankCheckSector  
Blank check sector

enumerator kIapCmd\_IAP\_ReadPartId  
Read part id

enumerator kIapCmd\_IAP\_Read\_BootromVersion  
Read bootrom version

enumerator kIapCmd\_IAP\_Compare  
Compare

enumerator kIapCmd\_IAP\_ReinvokeISP  
Reinvoke ISP

enumerator kIapCmd\_IAP\_ReadUid  
Read Uid

enumerator kIapCmd\_IAP\_ErasePage  
Erase Page

enumerator kIapCmd\_IAP\_ReadSignature  
Read Signature

enumerator kIapCmd\_IAP\_ExtendedReadSignature  
Extended Read Signature

enumerator kIapCmd\_IAP\_ReadEEPROMPage  
Read EEPROM page

enumerator kIapCmd\_IAP\_WriteEEPROMPage  
Write EEPROM page

enum `_flash_access_time`

Flash memory access time.

*Values:*

enumerator `kFlash_IAP_OneSystemClockTime`

enumerator `kFlash_IAP_TwoSystemClockTime`

1 system clock flash access time

enumerator `kFlash_IAP_ThreeSystemClockTime`

2 system clock flash access time

## 2.9 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.

MCUX\_MASK\_INVERT\_8(mask)

8-bit mask inversion.

MCUX\_MASK\_INVERT\_16(mask)

16-bit mask inversion.

MCUX\_MASK\_INVERT\_32(mask)

32-bit mask inversion for completeness.

MCUX\_REG\_WRITE8(reg, value)

8-bit register write macro

MCUX\_REG\_WRITE16(reg, value)

16-bit register write macro

MCUX\_REG\_WRITE32(reg, value)

32-bit register write macro

MCUX\_REG\_READ8(reg)

8-bit register read macro

MCUX\_REG\_READ16(reg)

16-bit register read macro

MCUX\_REG\_READ32(reg)

32-bit register read macro

MCUX\_REG\_BIT\_SET8(reg, mask)

8-bit register bit set macro

MCUX\_REG\_BIT\_SET16(reg, mask)

16-bit register bit set macro

MCUX\_REG\_BIT\_SET32(reg, mask)

32-bit register bit set macro

MCUX\_REG\_BIT\_CLEAR8(reg, mask)

8-bit register bit clear macro

MCUX\_REG\_BIT\_CLEAR16(reg, mask)

16-bit register bit clear macro

MCUX\_REG\_BIT\_CLEAR32(reg, mask)

32-bit register bit clear macro

MCUX\_REG\_BIT\_GET8(reg, mask)

8-bit register bit get macro

MCUX\_REG\_BIT\_GET16(reg, mask)

16-bit register bit get macro

MCUX\_REG\_BIT\_GET32(reg, mask)

32-bit register bit get macro

MCUX\_REG\_MODIFY8(reg, mask, value)

32-bit register read-modify-write macro

MCUX\_REG\_MODIFY16(reg, mask, value)

16-bit register read-modify-write macro

MCUX\_REG\_MODIFY32(reg, mask, value)

32-bit register read-modify-write macro

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\_ALIGN(var, alignbytes)

Macro to define a variable with alignbytes alignment

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 (rounded up)

SDK\_SIZEALIGN\_UP(var, alignbytes)

Macro to change a value to a given size aligned value (rounded up), the wrapper of SDK\_SIZEALIGN

SDK\_SIZEALIGN\_DOWN(var, alignbytes)

Macro to change a value to a given size aligned value (rounded down)

SDK\_IS\_ALIGNED(var, alignbytes)

Macro to check if a value is aligned to a given size

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*.

AT\_CACHE\_LINE\_SECTION(*var*)

Define a variable *var*, which is cache line size aligned and be placed in CacheLineData section.

AT\_CACHE\_LINE\_SECTION\_INIT(*var*)

Define a variable *var* with initial value, which is cache line size aligned and be placed in CacheLineData.init section.

AT\_QUICKACCESS\_SECTION\_CODE(*func*)

Place function in a section which can be accessed quickly by core.

AT\_QUICKACCESS\_SECTION\_DATA(*var*)

Place data in a section which can be accessed quickly by core.

AT\_QUICKACCESS\_SECTION\_DATA\_ALIGN(*var*, *alignbytes*)

Place data in a section which can be accessed quickly by core, and the variable address is set to align with *alignbytes*.

MCUX\_RAMFUNC

Function attribute to place function in RAM. For example, to place function *my\_func* in ram, use like:

```
MCUX_RAMFUNC my_func
```

RAMFUNCTION\_SECTION\_CODE(*func*)

Place function in ram.

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\_MIPI\_DSI  
Group number for MIPI DSI status codes

enumerator kStatusGroup\_SDRAMC  
Group number for SDRAMC status codes.

enumerator kStatusGroup\_POWER  
Group number for POWER status codes.

enumerator kStatusGroup\_ENET  
Group number for ENET status codes.

enumerator kStatusGroup\_PHY  
Group number for PHY status codes.

enumerator kStatusGroup\_TRGMUX  
Group number for TRGMUX status codes.

enumerator kStatusGroup\_SMARTCARD  
Group number for SMARTCARD status codes.

enumerator kStatusGroup\_LMEM  
Group number for LMEM status codes.

enumerator kStatusGroup\_QSPI  
Group number for QSPI status codes.

enumerator kStatusGroup\_DMA  
Group number for DMA status codes.

enumerator kStatusGroup\_EDMA  
Group number for EDMA status codes.

enumerator kStatusGroup\_DMAMGR  
Group number for DMAMGR status codes.

enumerator kStatusGroup\_FLEXCAN  
Group number for FlexCAN status codes.

enumerator kStatusGroup\_LTC  
Group number for LTC status codes.

enumerator kStatusGroup\_FLEXIO\_CAMERA  
Group number for FLEXIO CAMERA status codes.

enumerator kStatusGroup\_LPC\_SPI  
Group number for LPC\_SPI status codes.

enumerator kStatusGroup\_LPC\_USART  
Group number for LPC\_USART status codes.

enumerator kStatusGroup\_DMIC  
Group number for DMIC status codes.

enumerator kStatusGroup\_SDIF  
Group number for SDIF status codes.

- enumerator kStatusGroup\_SPIFI  
Group number for SPIFI status codes.
- enumerator kStatusGroup\_OTP  
Group number for OTP status codes.
- enumerator kStatusGroup\_MCAN  
Group number for MCAN status codes.
- enumerator kStatusGroup\_CAAM  
Group number for CAAM status codes.
- enumerator kStatusGroup\_ECSPi  
Group number for ECSPi status codes.
- enumerator kStatusGroup\_USDHC  
Group number for USDHC status codes.
- enumerator kStatusGroup\_LPC\_I2C  
Group number for LPC\_I2C status codes.
- enumerator kStatusGroup\_DCP  
Group number for DCP status codes.
- enumerator kStatusGroup\_MSCAN  
Group number for MSCAN status codes.
- enumerator kStatusGroup\_ESAI  
Group number for ESAI status codes.
- enumerator kStatusGroup\_FLEXSPI  
Group number for FLEXSPI status codes.
- enumerator kStatusGroup\_MMDC  
Group number for MMDC status codes.
- enumerator kStatusGroup\_PDM  
Group number for MIC status codes.
- enumerator kStatusGroup\_SDMA  
Group number for SDMA status codes.
- enumerator kStatusGroup\_ICS  
Group number for ICS status codes.
- enumerator kStatusGroup\_SPDIF  
Group number for SPDIF status codes.
- enumerator kStatusGroup\_LPC\_MINISPI  
Group number for LPC\_MINISPI status codes.
- enumerator kStatusGroup\_HASHCRYPT  
Group number for Hashcrypt status codes
- enumerator kStatusGroup\_LPC\_SPI\_SSP  
Group number for LPC\_SPI\_SSP status codes.
- enumerator kStatusGroup\_I3C  
Group number for I3C status codes
- enumerator kStatusGroup\_LPC\_I2C\_1  
Group number for LPC\_I2C\_1 status codes.

enumerator `kStatusGroup_NOTIFIER`  
Group number for NOTIFIER status codes.

enumerator `kStatusGroup_DebugConsole`  
Group number for debug console status codes.

enumerator `kStatusGroup_SEMC`  
Group number for SEMC status codes.

enumerator `kStatusGroup_ApplicationRangeStart`  
Starting number for application groups.

enumerator `kStatusGroup_IAP`  
Group number for IAP status codes

enumerator `kStatusGroup_SFA`  
Group number for SFA status codes

enumerator `kStatusGroup_SPC`  
Group number for SPC status codes.

enumerator `kStatusGroup_PUF`  
Group number for PUF status codes.

enumerator `kStatusGroup_TOUCH_PANEL`  
Group number for touch panel status codes

enumerator `kStatusGroup_VBAT`  
Group number for VBAT status codes

enumerator `kStatusGroup_XSPI`  
Group number for XSPI status codes

enumerator `kStatusGroup_PNGDEC`  
Group number for PNGDEC status codes

enumerator `kStatusGroup_JPEGDEC`  
Group number for JPEGDEC status codes

enumerator `kStatusGroup_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.

enumerator kStatusGroup\_LPC\_QSPI  
Group number for LPC QSPI 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().

void EnableDeepSleepIRQ(IRQn\_Type interrupt)

Enable specific interrupt for wake-up from deep-sleep mode.

Enable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

---

**Note:** This function also enables the interrupt in the NVIC (EnableIRQ() is called internally).

---

### Parameters

- interrupt – The IRQ number.

void DisableDeepSleepIRQ(IRQn\_Type interrupt)

Disable specific interrupt for wake-up from deep-sleep mode.

Disable the interrupt for wake-up from deep sleep mode. Some interrupts are typically used in sleep mode only and will not occur during deep-sleep mode because relevant clocks are stopped. However, it is possible to enable those clocks (significantly increasing power consumption in the reduced power mode), making these wake-ups possible.

---

**Note:** This function also disables the interrupt in the NVIC (DisableIRQ() is called internally).

---

### Parameters

- interrupt – The IRQ number.

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\_HAS\_DWT\_CYCCNT

The chip supports DWT CYCCNT or not.

MSDK\_INVALID\_IRQ\_HANDLER

Invalid IRQ handler address.

## 2.10 LPC\_ACOMP: Analog comparator Driver

void ACOMP\_Init(ACOMP\_Type \*base, const *acomp\_config\_t* \*config)

Initialize the ACOMP module.

### Parameters

- base – ACOMP peripheral base address.
- config – Pointer to “acomp\_config\_t” structure.

void ACOMP\_Deinit(ACOMP\_Type \*base)

De-initialize the ACOMP module.

### Parameters

- base – ACOMP peripheral base address.

void ACOMP\_GetDefaultConfig(*acomp\_config\_t* \*config)

Gets an available pre-defined settings for the ACOMP’s configuration.

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

```
config->enableSyncToBusClk = false;
config->hysteresisSelection = kACOMP_hysteresisNoneSelection;
```

In default configuration, the ACOMP’s output would be used directly and switch as the voltages cross.

### Parameters

- config – Pointer to the configuration structure.

void ACOMP\_EnableInterrupts(ACOMP\_Type \*base, *acomp\_interrupt\_enable\_t* enable)

Enable ACOMP interrupts.

### Parameters

- base – ACOMP peripheral base address.
- enable – Enable/Disable interrupt feature.

```
static inline bool ACOMP_GetInterruptsStatusFlags(ACOMP_Type *base)
```

Get interrupts status flags.

**Parameters**

- base – ACOMP peripheral base address.

**Returns**

Reflect the state ACOMP edge-detect status, true or false.

```
static inline void ACOMP_ClearInterruptsStatusFlags(ACOMP_Type *base)
```

Clear the ACOMP interrupts status flags.

**Parameters**

- base – ACOMP peripheral base address.

```
static inline bool ACOMP_GetOutputStatusFlags(ACOMP_Type *base)
```

Get ACOMP output status flags.

**Parameters**

- base – ACOMP peripheral base address.

**Returns**

Reflect the state of the comparator output, true or false.

```
static inline void ACOMP_SetInputChannel(ACOMP_Type *base, uint32_t positiveInputChannel,
                                         uint32_t negativeInputChannel)
```

Set the ACOMP positive and negative input channel.

**Parameters**

- base – ACOMP peripheral base address.
- positiveInputChannel – The index of positive input channel.
- negativeInputChannel – The index of negative input channel.

```
void ACOMP_SetLadderConfig(ACOMP_Type *base, const acomp_ladder_config_t *config)
```

Set the voltage ladder configuration.

**Parameters**

- base – ACOMP peripheral base address.
- config – The structure for voltage ladder. If the config is NULL, voltage ladder would be disabled, otherwise the voltage ladder would be configured and enabled.

```
FSL_ACOMP_DRIVER_VERSION
```

ACOMP driver version 2.1.0.

```
enum _acomp_ladder_reference_voltage
```

The ACOMP ladder reference voltage.

*Values:*

```
enumerator kACOMP_LadderRefVoltagePinVDD
```

Supply from pin VDD.

```
enumerator kACOMP_LadderRefVoltagePinVDDCMP
```

Supply from pin VDDCMP.

```
enum _acomp_interrupt_enable
```

The ACOMP interrupts enable.

*Values:*

enumerator kACOMP\_InterruptsFallingEdgeEnable

Enable the falling edge interrupts.

enumerator kACOMP\_InterruptsRisingEdgeEnable

Enable the rising edge interrupts.

enumerator kACOMP\_InterruptsBothEdgesEnable

Enable the both edges interrupts.

enumerator kACOMP\_InterruptsDisable

Disable the interrupts.

enum \_acomp\_hysteresis\_selection

The ACOMP hysteresis selection.

*Values:*

enumerator kACOMP\_HysteresisNoneSelection

None (the output will switch as the voltages cross).

enumerator kACOMP\_Hysteresis5MVSelection

5mV.

enumerator kACOMP\_Hysteresis10MVSelection

10mV.

enumerator kACOMP\_Hysteresis20MVSelection

20mV.

typedef enum *\_acomp\_ladder\_reference\_voltage* acomp\_ladder\_reference\_voltage\_t

The ACOMP ladder reference voltage.

typedef enum *\_acomp\_interrupt\_enable* acomp\_interrupt\_enable\_t

The ACOMP interrupts enable.

typedef enum *\_acomp\_hysteresis\_selection* acomp\_hysteresis\_selection\_t

The ACOMP hysteresis selection.

typedef struct *\_acomp\_config* acomp\_config\_t

The structure for ACOMP basic configuration.

typedef struct *\_acomp\_ladder\_config* acomp\_ladder\_config\_t

The structure for ACOMP voltage ladder.

struct \_acomp\_config

*#include <fsl\_acomp.h>* The structure for ACOMP basic configuration.

### Public Members

bool enableSyncToBusClk

If true, Comparator output is synchronized to the bus clock for output to other modules.  
If false, Comparator output is used directly.

*acomp\_hysteresis\_selection\_t* hysteresisSelection

Controls the hysteresis of the comparator.

struct \_acomp\_ladder\_config

*#include <fsl\_acomp.h>* The structure for ACOMP voltage ladder.

**Public Members**

uint8\_t ladderValue

Voltage ladder value. 00000 = Vss, 00001 = 1\*Vref/31, ..., 11111 = Vref.

*acomp\_ladder\_reference\_voltage\_t* referenceVoltage

Selects the reference voltage(Vref) for the voltage ladder.

**2.11 ADC: 12-bit SAR Analog-to-Digital Converter Driver**

void ADC\_Init(ADC\_Type \*base, const *adc\_config\_t* \*config)

Initialize the ADC module.

**Parameters**

- base – ADC peripheral base address.
- config – Pointer to configuration structure, see to *adc\_config\_t*.

void ADC\_Deinit(ADC\_Type \*base)

Deinitialize the ADC module.

**Parameters**

- base – ADC peripheral base address.

void ADC\_GetDefaultConfig(*adc\_config\_t* \*config)

Gets an available pre-defined settings for initial configuration.

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

```
config->clockMode = kADC_ClockSynchronousMode;
config->clockDividerNumber = 0U;
config->resolution = kADC_Resolution12bit;
config->enableBypassCalibration = false;
config->sampleTimeNumber = 0U;
config->extendSampleTimeNumber = kADC_ExtendSampleTimeNotUsed;
```

**Parameters**

- config – Pointer to configuration structure.

static inline void ADC\_EnableConvSeqA(ADC\_Type \*base, bool enable)

Enable the conversion sequence A.

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. when the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

**Parameters**

- base – ADC peripheral base address.
- enable – Switcher to enable the feature or not.

void ADC\_SetConvSeqAConfig(ADC\_Type \*base, const *adc\_conv\_seq\_config\_t* \*config)

Configure the conversion sequence A.

**Parameters**

- base – ADC peripheral base address.
- config – Pointer to configuration structure, see to *adc\_conv\_seq\_config\_t*.

```
static inline void ADC_DoSoftwareTriggerConvSeqA(ADC_Type *base)
```

Do trigger the sequence's conversion by software.

**Parameters**

- base – ADC peripheral base address.

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

Enable the burst conversion of sequence A.

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before conversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

**Parameters**

- base – ADC peripheral base address.
- enable – Switcher to enable this feature.

```
static inline void ADC_SetConvSeqAHighPriority(ADC_Type *base)
```

Set the high priority for conversion sequence A.

**Parameters**

- base – ADC peripheral base address.

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

Enable the conversion sequence B.

In order to avoid spuriously triggering the sequence, the trigger to conversion sequence should be ready before the sequence is ready. When the sequence is disabled, the trigger would be ignored. Also, it is suggested to disable the sequence during changing the sequence's setting.

**Parameters**

- base – ADC peripheral base address.
- enable – Switcher to enable the feature or not.

```
void ADC_SetConvSeqBConfig(ADC_Type *base, const adc_conv_seq_config_t *config)
```

Configure the conversion sequence B.

**Parameters**

- base – ADC peripheral base address.
- config – Pointer to configuration structure, see to *adc\_conv\_seq\_config\_t*.

```
static inline void ADC_DoSoftwareTriggerConvSeqB(ADC_Type *base)
```

Do trigger the sequence's conversion by software.

**Parameters**

- base – ADC peripheral base address.

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

Enable the burst conversion of sequence B.

Enable the burst mode would cause the conversion sequence to be continuously cycled through. Other triggers would be ignored while this mode is enabled. Repeated conversions could be halted by disabling this mode. And the sequence currently in process will be completed before conversions are terminated. Note that a new sequence could begin just before the burst mode is disabled.

**Parameters**

- base – ADC peripheral base address.
- enable – Switcher to enable this feature.

```
static inline void ADC_SetConvSeqBHighPriority(ADC_Type *base)
```

Set the high priority for conversion sequence B.

#### Parameters

- base – ADC peripheral base address.

```
bool ADC_GetConvSeqAGlobalConversionResult(ADC_Type *base, adc_result_info_t *info)
```

Get the global ADC conversion information of sequence A.

#### Parameters

- base – ADC peripheral base address.
- info – Pointer to information structure, see to `adc_result_info_t`;

#### Return values

- true – The conversion result is ready.
- false – The conversion result is not ready yet.

```
bool ADC_GetConvSeqBGlobalConversionResult(ADC_Type *base, adc_result_info_t *info)
```

Get the global ADC conversion information of sequence B.

#### Parameters

- base – ADC peripheral base address.
- info – Pointer to information structure, see to `adc_result_info_t`;

#### Return values

- true – The conversion result is ready.
- false – The conversion result is not ready yet.

```
bool ADC_GetChannelConversionResult(ADC_Type *base, uint32_t channel, adc_result_info_t *info)
```

Get the channel's ADC conversion completed under each conversion sequence.

#### Parameters

- base – ADC peripheral base address.
- channel – The indicated channel number.
- info – Pointer to information structure, see to `adc_result_info_t`;

#### Return values

- true – The conversion result is ready.
- false – The conversion result is not ready yet.

```
static inline void ADC_SetThresholdPair0(ADC_Type *base, uint32_t lowValue, uint32_t highValue)
```

Set the threshold pair 0 with low and high value.

#### Parameters

- base – ADC peripheral base address.
- lowValue – LOW threshold value.
- highValue – HIGH threshold value.

```
static inline void ADC_SetThresholdPair1(ADC_Type *base, uint32_t lowValue, uint32_t highValue)
```

Set the threshold pair 1 with low and high value.

**Parameters**

- base – ADC peripheral base address.
- lowValue – LOW threshold value. The available value is with 12-bit.
- highValue – HIGH threshold value. The available value is with 12-bit.

```
static inline void ADC_SetChannelWithThresholdPair0(ADC_Type *base, uint32_t channelMask)
```

Set given channels to apply the threshold pare 0.

**Parameters**

- base – ADC peripheral base address.
- channelMask – Indicated channels' mask.

```
static inline void ADC_SetChannelWithThresholdPair1(ADC_Type *base, uint32_t channelMask)
```

Set given channels to apply the threshold pare 1.

**Parameters**

- base – ADC peripheral base address.
- channelMask – Indicated channels' mask.

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

Enable interrupts for conversion sequences.

**Parameters**

- base – ADC peripheral base address.
- mask – Mask of interrupt mask value for global block except each channel, see to `_adc_interrupt_enable`.

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

Disable interrupts for conversion sequence.

**Parameters**

- base – ADC peripheral base address.
- mask – Mask of interrupt mask value for global block except each channel, see to `_adc_interrupt_enable`.

```
static inline void ADC_EnableThresholdCompareInterrupt(ADC_Type *base, uint32_t channel, adc_threshold_interrupt_mode_t mode)
```

Enable the interrupt of threshold compare event for each channel.

**Parameters**

- base – ADC peripheral base address.
- channel – Channel number.
- mode – Interrupt mode for threshold compare event, see to `adc_threshold_interrupt_mode_t`.

```
static inline uint32_t ADC_GetStatusFlags(ADC_Type *base)
```

Get status flags of ADC module.

**Parameters**

- base – ADC peripheral base address.

**Returns**

Mask of status flags of module, see to `_adc_status_flags`.

```
static inline void ADC_ClearStatusFlags(ADC_Type *base, uint32_t mask)
```

Clear status flags of ADC module.

**Parameters**

- `base` – ADC peripheral base address.
- `mask` – Mask of status flags of module, see to `_adc_status_flags`.

```
FSL_ADC_DRIVER_VERSION
```

ADC driver version 2.6.0.

```
enum _adc_status_flags
```

Flags.

*Values:*

```
enumerator kADC_ThresholdCompareFlagOnChn0
```

Threshold comparison event on Channel 0.

```
enumerator kADC_ThresholdCompareFlagOnChn1
```

Threshold comparison event on Channel 1.

```
enumerator kADC_ThresholdCompareFlagOnChn2
```

Threshold comparison event on Channel 2.

```
enumerator kADC_ThresholdCompareFlagOnChn3
```

Threshold comparison event on Channel 3.

```
enumerator kADC_ThresholdCompareFlagOnChn4
```

Threshold comparison event on Channel 4.

```
enumerator kADC_ThresholdCompareFlagOnChn5
```

Threshold comparison event on Channel 5.

```
enumerator kADC_ThresholdCompareFlagOnChn6
```

Threshold comparison event on Channel 6.

```
enumerator kADC_ThresholdCompareFlagOnChn7
```

Threshold comparison event on Channel 7.

```
enumerator kADC_ThresholdCompareFlagOnChn8
```

Threshold comparison event on Channel 8.

```
enumerator kADC_ThresholdCompareFlagOnChn9
```

Threshold comparison event on Channel 9.

```
enumerator kADC_ThresholdCompareFlagOnChn10
```

Threshold comparison event on Channel 10.

```
enumerator kADC_ThresholdCompareFlagOnChn11
```

Threshold comparison event on Channel 11.

```
enumerator kADC_OverrunFlagForChn0
```

Mirror the `OVERRUN` status flag from the result register for ADC channel 0.

```
enumerator kADC_OverrunFlagForChn1
```

Mirror the `OVERRUN` status flag from the result register for ADC channel 1.

```
enumerator kADC_OverrunFlagForChn2
```

Mirror the `OVERRUN` status flag from the result register for ADC channel 2.

enumerator kADC\_OverrunFlagForChn3

Mirror the OVERRUN status flag from the result register for ADC channel 3.

enumerator kADC\_OverrunFlagForChn4

Mirror the OVERRUN status flag from the result register for ADC channel 4.

enumerator kADC\_OverrunFlagForChn5

Mirror the OVERRUN status flag from the result register for ADC channel 5.

enumerator kADC\_OverrunFlagForChn6

Mirror the OVERRUN status flag from the result register for ADC channel 6.

enumerator kADC\_OverrunFlagForChn7

Mirror the OVERRUN status flag from the result register for ADC channel 7.

enumerator kADC\_OverrunFlagForChn8

Mirror the OVERRUN status flag from the result register for ADC channel 8.

enumerator kADC\_OverrunFlagForChn9

Mirror the OVERRUN status flag from the result register for ADC channel 9.

enumerator kADC\_OverrunFlagForChn10

Mirror the OVERRUN status flag from the result register for ADC channel 10.

enumerator kADC\_OverrunFlagForChn11

Mirror the OVERRUN status flag from the result register for ADC channel 11.

enumerator kADC\_GlobalOverrunFlagForSeqA

Mirror the global OVERRUN status flag for conversion sequence A.

enumerator kADC\_GlobalOverrunFlagForSeqB

Mirror the global OVERRUN status flag for conversion sequence B.

enumerator kADC\_ConvSeqAInterruptFlag

Sequence A interrupt/DMA trigger.

enumerator kADC\_ConvSeqBInterruptFlag

Sequence B interrupt/DMA trigger.

enumerator kADC\_ThresholdCompareInterruptFlag

Threshold comparison interrupt flag.

enumerator kADC\_OverrunInterruptFlag

Overrun interrupt flag.

enum \_adc\_interrupt\_enable

Interrupts.

---

**Note:** Not all the interrupt options are listed here

---

*Values:*

enumerator kADC\_ConvSeqAInterruptEnable

Enable interrupt upon completion of each individual conversion in sequence A, or entire sequence.

enumerator kADC\_ConvSeqBInterruptEnable

Enable interrupt upon completion of each individual conversion in sequence B, or entire sequence.

enumerator kADC\_OverrunInterruptEnable

Enable the detection of an overrun condition on any of the channel data registers will cause an overrun interrupt/DMA trigger.

enum \_adc\_clock\_mode

Define selection of clock mode.

*Values:*

enumerator kADC\_ClockSynchronousMode

The ADC clock would be derived from the system clock based on “clockDividerNumber”.

enumerator kADC\_ClockAsynchronousMode

The ADC clock would be based on the SYSCON block’s divider.

enum \_adc\_resolution

Define selection of resolution.

*Values:*

enumerator kADC\_Resolution6bit

6-bit resolution.

enumerator kADC\_Resolution8bit

8-bit resolution.

enumerator kADC\_Resolution10bit

10-bit resolution.

enumerator kADC\_Resolution12bit

12-bit resolution.

enum \_adc\_voltage\_range

Define range of the analog supply voltage VDDA.

*Values:*

enumerator kADC\_HighVoltageRange

enumerator kADC\_LowVoltageRange

enum \_adc\_trigger\_polarity

Define selection of polarity of selected input trigger for conversion sequence.

*Values:*

enumerator kADC\_TriggerPolarityNegativeEdge

A negative edge launches the conversion sequence on the trigger(s).

enumerator kADC\_TriggerPolarityPositiveEdge

A positive edge launches the conversion sequence on the trigger(s).

enum \_adc\_priority

Define selection of conversion sequence’s priority.

*Values:*

enumerator kADC\_PriorityLow

This sequence would be preempted when another sequence is started.

enumerator kADC\_PriorityHigh

This sequence would preempt other sequence even when it is started.

enum `_adc_seq_interrupt_mode`

Define selection of conversion sequence's interrupt.

*Values:*

enumerator `kADC_InterruptForEachConversion`

The sequence interrupt/DMA trigger will be set at the end of each individual ADC conversion inside this conversion sequence.

enumerator `kADC_InterruptForEachSequence`

The sequence interrupt/DMA trigger will be set when the entire set of this sequence conversions completes.

enum `_adc_threshold_compare_status`

Define status of threshold compare result.

*Values:*

enumerator `kADC_ThresholdCompareInRange`

LOW threshold  $\leq$  conversion value  $\leq$  HIGH threshold.

enumerator `kADC_ThresholdCompareBelowRange`

conversion value  $<$  LOW threshold.

enumerator `kADC_ThresholdCompareAboveRange`

conversion value  $>$  HIGH threshold.

enum `_adc_threshold_crossing_status`

Define status of threshold crossing detection result.

*Values:*

enumerator `kADC_ThresholdCrossingNoDetected`

No threshold Crossing detected.

enumerator `kADC_ThresholdCrossingDownward`

Downward Threshold Crossing detected.

enumerator `kADC_ThresholdCrossingUpward`

Upward Threshold Crossing Detected.

enum `_adc_threshold_interrupt_mode`

Define interrupt mode for threshold compare event.

*Values:*

enumerator `kADC_ThresholdInterruptDisabled`

Threshold comparison interrupt is disabled.

enumerator `kADC_ThresholdInterruptOnOutside`

Threshold comparison interrupt is enabled on outside threshold.

enumerator `kADC_ThresholdInterruptOnCrossing`

Threshold comparison interrupt is enabled on crossing threshold.

enum `_adc_inforesultshift`

Define the info result mode of different resolution.

*Values:*

enumerator `kADC_Resolution12bitInfoResultShift`

Info result shift of Resolution12bit.

enumerator `kADC_Resolution10bitInfoResultShift`

Info result shift of Resolution10bit.

enumerator `kADC_Resolution8bitInfoResultShift`  
Info result shift of Resolution8bit.

enumerator `kADC_Resolution6bitInfoResultShift`  
Info result shift of Resolution6bit.

enum `_adc_tempsensor_common_mode`  
Define common modes for Temperature sensor.

*Values:*

enumerator `kADC_HighNegativeOffsetAdded`  
Temperature sensor common mode: high negative offset added.

enumerator `kADC_IntermediateNegativeOffsetAdded`  
Temperature sensor common mode: intermediate negative offset added.

enumerator `kADC_NoOffsetAdded`  
Temperature sensor common mode: no offset added.

enumerator `kADC_LowPositiveOffsetAdded`  
Temperature sensor common mode: low positive offset added.

enum `_adc_second_control`  
Define source impedance modes for GPADC control.

*Values:*

enumerator `kADC_Impedance621Ohm`  
Extend ADC sampling time according to source impedance 1: 0.621 kOhm.

enumerator `kADC_Impedance55kOhm`  
Extend ADC sampling time according to source impedance 20 (default): 55 kOhm.

enumerator `kADC_Impedance87kOhm`  
Extend ADC sampling time according to source impedance 31: 87 kOhm.

enumerator `kADC_NormalFunctionalMode`  
TEST mode: Normal functional mode.

enumerator `kADC_MultiplexeTestMode`  
TEST mode: Multiplexer test mode.

enumerator `kADC_ADCInUnityGainMode`  
TEST mode: ADC in unity gain mode.

typedef enum `_adc_clock_mode` `adc_clock_mode_t`  
Define selection of clock mode.

typedef enum `_adc_resolution` `adc_resolution_t`  
Define selection of resolution.

typedef enum `_adc_voltage_range` `adc_vdda_range_t`  
Define range of the analog supply voltage VDDA.

typedef enum `_adc_trigger_polarity` `adc_trigger_polarity_t`  
Define selection of polarity of selected input trigger for conversion sequence.

typedef enum `_adc_priority` `adc_priority_t`  
Define selection of conversion sequence's priority.

typedef enum `_adc_seq_interrupt_mode` `adc_seq_interrupt_mode_t`  
Define selection of conversion sequence's interrupt.

typedef enum *\_adc\_threshold\_compare\_status* adc\_threshold\_compare\_status\_t

Define status of threshold compare result.

typedef enum *\_adc\_threshold\_crossing\_status* adc\_threshold\_crossing\_status\_t

Define status of threshold crossing detection result.

typedef enum *\_adc\_threshold\_interrupt\_mode* adc\_threshold\_interrupt\_mode\_t

Define interrupt mode for threshold compare event.

typedef enum *\_adc\_inforeresultshift* adc\_inforeresult\_t

Define the info result mode of different resolution.

typedef enum *\_adc\_tempsensor\_common\_mode* adc\_tempsensor\_common\_mode\_t

Define common modes for Temperature sensor.

typedef enum *\_adc\_second\_control* adc\_second\_control\_t

Define source impedance modes for GPADC control.

typedef struct *\_adc\_config* adc\_config\_t

Define structure for configuring the block.

typedef struct *\_adc\_conv\_seq\_config* adc\_conv\_seq\_config\_t

Define structure for configuring conversion sequence.

typedef struct *\_adc\_result\_info* adc\_result\_info\_t

Define structure of keeping conversion result information.

struct *\_adc\_config*

*#include <fsl\_adc.h>* Define structure for configuring the block.

## Public Members

*adc\_clock\_mode\_t* clockMode

Select the clock mode for ADC converter.

uint32\_t clockDividerNumber

This field is only available when using *kADC\_ClockSynchronousMode* for “clockMode” field. The divider would be plused by 1 based on the value in this field. The available range is in 8 bits.

*adc\_resolution\_t* resolution

Select the conversion bits.

bool enableBypassCalibration

By default, a calibration cycle must be performed each time the chip is powered-up. Re-calibration may be warranted periodically - especially if operating conditions have changed. To enable this option would avoid the need to calibrate if offset error is not a concern in the application.

uint32\_t sampleTimeNumber

By default, with value as “0U”, the sample period would be 2.5 ADC clocks. Then, to plus the “sampleTimeNumber” value here. The available value range is in 3 bits.

bool enableLowPowerMode

If disable low-power mode, ADC remains activated even when no conversions are requested. If enable low-power mode, The ADC is automatically powered-down when no conversions are taking place.

*adc\_vdda\_range\_t* voltageRange

Configure the ADC for the appropriate operating range of the analog supply voltage VDDA. Failure to set the area correctly causes the ADC to return incorrect conversion results.

struct *\_adc\_conv\_seq\_config*

*#include <fsl\_adc.h>* Define structure for configuring conversion sequence.

### Public Members

uint32\_t channelMask

Selects which one or more of the ADC channels will be sampled and converted when this sequence is launched. The masked channels would be involved in current conversion sequence, beginning with the lowest-order. The available range is in 12-bit.

uint32\_t triggerMask

Selects which one or more of the available hardware trigger sources will cause this conversion sequence to be initiated. The available range is 6-bit.

*adc\_trigger\_polarity\_t* triggerPolarity

Select the trigger to launch conversion sequence.

bool enableSyncBypass

To enable this feature allows the hardware trigger input to bypass synchronization flip-flop stages and therefore shorten the time between the trigger input signal and the start of a conversion.

bool enableSingleStep

When enabling this feature, a trigger will launch a single conversion on the next channel in the sequence instead of the default response of launching an entire sequence of conversions.

*adc\_seq\_interrupt\_mode\_t* interruptMode

Select the interrupt/DMA trigger mode.

struct *\_adc\_result\_info*

*#include <fsl\_adc.h>* Define structure of keeping conversion result information.

### Public Members

uint32\_t result

Keep the conversion data value.

*adc\_threshold\_compare\_status\_t* thresholdCompareStatus

Keep the threshold compare status.

*adc\_threshold\_crossing\_status\_t* thresholdCorssingStatus

Keep the threshold crossing status.

uint32\_t channelNumber

Keep the channel number for this conversion.

bool overrunFlag

Keep the status whether the conversion is overrun or not.

## 2.12 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

## 2.13 IOCON: I/O pin configuration

LPC\_IOCON\_DRIVER\_VERSION

IOCON driver version 2.0.2.

```
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 ionumber,  
uint32_t modefunc)
```

IOCON function and mode selection definitions.

Sets I/O Control pin mux

---

**Note:** See the User Manual for specific modes and functions supported by the various pins.

---

### Parameters

- base – : The base of IOCON peripheral on the chip
- ionumber – : GPIO number 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
```

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

## 2.14 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

```
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

## 2.15 PINT: Pin Interrupt and Pattern Match Driver

```
FSL_PINT_DRIVER_VERSION
```

```
enum _pint_pin_enable
```

PINT Pin Interrupt enable type.

*Values:*

```
enumerator kPINT_PinIntEnableNone
```

Do not generate Pin Interrupt

```
enumerator kPINT_PinIntEnableRiseEdge
```

Generate Pin Interrupt on rising edge

```
enumerator kPINT_PinIntEnableFallEdge
```

Generate Pin Interrupt on falling edge

```
enumerator kPINT_PinIntEnableBothEdges
```

Generate Pin Interrupt on both edges

```
enumerator kPINT_PinIntEnableLowLevel
```

Generate Pin Interrupt on low level

```
enumerator kPINT_PinIntEnableHighLevel
```

Generate Pin Interrupt on high level

```
enum _pint_int
```

PINT Pin Interrupt type.

*Values:*

```
enumerator kPINT_PinInt0
```

Pin Interrupt 0

```
enum _pint_pmatch_input_src
```

PINT Pattern Match bit slice input source type.

*Values:*

```
enumerator kPINT_PatternMatchInp0Src
```

Input source 0

```
enumerator kPINT_PatternMatchInp1Src
```

Input source 1

enumerator kPINT\_PatternMatchInp2Src  
Input source 2

enumerator kPINT\_PatternMatchInp3Src  
Input source 3

enumerator kPINT\_PatternMatchInp4Src  
Input source 4

enumerator kPINT\_PatternMatchInp5Src  
Input source 5

enumerator kPINT\_PatternMatchInp6Src  
Input source 6

enumerator kPINT\_PatternMatchInp7Src  
Input source 7

enumerator kPINT\_SecPatternMatchInp0Src  
Input source 0

enumerator kPINT\_SecPatternMatchInp1Src  
Input source 1

enum \_pint\_pmatch\_bslice  
PINT Pattern Match bit slice type.

*Values:*

enumerator kPINT\_PatternMatchBSlice0  
Bit slice 0

enum \_pint\_pmatch\_bslice\_cfg  
PINT Pattern Match configuration type.

*Values:*

enumerator kPINT\_PatternMatchAlways  
Always Contributes to product term match

enumerator kPINT\_PatternMatchStickyRise  
Sticky Rising edge

enumerator kPINT\_PatternMatchStickyFall  
Sticky Falling edge

enumerator kPINT\_PatternMatchStickyBothEdges  
Sticky Rising or Falling edge

enumerator kPINT\_PatternMatchHigh  
High level

enumerator kPINT\_PatternMatchLow  
Low level

enumerator kPINT\_PatternMatchNever  
Never contributes to product term match

enumerator kPINT\_PatternMatchBothEdges  
Either rising or falling edge

typedef enum \_pint\_pin\_enable pint\_pin\_enable\_t  
PINT Pin Interrupt enable type.

```
typedef enum _pint_int pint_pin_int_t
```

PINT Pin Interrupt type.

```
typedef enum _pint_pmatch_input_src pint_pmatch_input_src_t
```

PINT Pattern Match bit slice input source type.

```
typedef enum _pint_pmatch_bslice pint_pmatch_bslice_t
```

PINT Pattern Match bit slice type.

```
typedef enum _pint_pmatch_bslice_cfg pint_pmatch_bslice_cfg_t
```

PINT Pattern Match configuration type.

```
typedef struct _pint_status pint_status_t
```

PINT event status.

```
typedef void (*pint_cb_t)(pint_pin_int_t pintr, pint_status_t *status)
```

PINT Callback function.

```
typedef struct _pint_pmatch_cfg pint_pmatch_cfg_t
```

```
void PINT_Init(PINT_Type *base)
```

Initialize PINT peripheral.

This function initializes the PINT peripheral and enables the clock.

#### Parameters

- *base* – Base address of the PINT peripheral.

#### Return values

None. –

```
void PINT_SetCallback(PINT_Type *base, pint_cb_t callback)
```

Set PINT callback.

This function set the callback for PINT interrupt handler.

#### Parameters

- *base* – Base address of the PINT peripheral.
- *callback* – Callback.

#### Return values

None. –

```
void PINT_PinInterruptConfig(PINT_Type *base, pint_pin_int_t intr, pint_pin_enable_t enable)
```

Configure PINT peripheral pin interrupt.

This function configures a given pin interrupt.

#### Parameters

- *base* – Base address of the PINT peripheral.
- *intr* – Pin interrupt.
- *enable* – Selects detection logic.

#### Return values

None. –

```
void PINT_PinInterruptGetConfig(PINT_Type *base, pint_pin_int_t pintr, pint_pin_enable_t *enable)
```

Get PINT peripheral pin interrupt configuration.

This function returns the configuration of a given pin interrupt.

#### Parameters

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.
- enable – Pointer to store the detection logic.

**Return values**

None. –

```
void PINT_PinInterruptClrStatus(PINT_Type *base, pint_pin_int_t pintr)
```

Clear Selected pin interrupt status only when the pin was triggered by edge-sensitive.

This function clears the selected pin interrupt status.

**Parameters**

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

**Return values**

None. –

```
static inline uint32_t PINT_PinInterruptGetStatus(PINT_Type *base, pint_pin_int_t 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, pint_pin_int_t pintr)
```

Clear Selected pin interrupt fall flag.

This function clears the selected pin interrupt fall flag.

**Parameters**

- base – Base address of the PINT peripheral.
- pintr – Pin interrupt.

**Return values**

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlag(PINT_Type *base, pintr)
```

Get selected pin interrupt fall flag.

This function returns the selected pin interrupt fall flag.

**Parameters**

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

**Return values***flag* – = 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrFallFlagAll(PINT_Type *base)
```

Clear all pin interrupt fall flags.

This function clears the fall flag for all pin interrupts.

**Parameters**

- *base* – Base address of the PINT peripheral.

**Return values**

None. –

```
static inline uint32_t PINT_PinInterruptGetFallFlagAll(PINT_Type *base)
```

Get all pin interrupt fall flags.

This function returns the fall flag of all pin interrupts.

**Parameters**

- *base* – Base address of the PINT peripheral.

**Return values***flags* – Each bit position indicates the falling edge detection of the corresponding pin interrupt. 0 Falling edge has not been detected. = 1 Falling edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlag(PINT_Type *base, pintr)
```

Clear Selected pin interrupt rise flag.

This function clears the selected pin interrupt rise flag.

**Parameters**

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

**Return values**

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlag(PINT_Type *base, pintr)
```

Get selected pin interrupt rise flag.

This function returns the selected pin interrupt rise flag.

**Parameters**

- *base* – Base address of the PINT peripheral.
- *pintr* – Pin interrupt.

**Return values***flag* – = 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
static inline void PINT_PinInterruptClrRiseFlagAll(PINT_Type *base)
```

Clear all pin interrupt rise flags.

This function clears the rise flag for all pin interrupts.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

```
static inline uint32_t PINT_PinInterruptGetRiseFlagAll(PINT_Type *base)
```

Get all pin interrupt rise flags.

This function returns the rise flag of all pin interrupts.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

flags – Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
void PINT_PatternMatchConfig(PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)
```

Configure PINT pattern match.

This function configures a given pattern match bit slice.

**Parameters**

- base – Base address of the PINT peripheral.
- bslice – Pattern match bit slice number.
- cfg – Pointer to bit slice configuration.

**Return values**

None. –

```
void PINT_PatternMatchGetConfig(PINT_Type *base, pint_pmatch_bslice_t bslice, pint_pmatch_cfg_t *cfg)
```

Get PINT pattern match configuration.

This function returns the configuration of a given pattern match bit slice.

**Parameters**

- base – Base address of the PINT peripheral.
- bslice – Pattern match bit slice number.
- cfg – Pointer to bit slice configuration.

**Return values**

None. –

```
static inline uint32_t PINT_PatternMatchGetStatus(PINT_Type *base, pint_pmatch_bslice_t bslice)
```

Get pattern match bit slice status.

This function returns the status of selected bit slice.

**Parameters**

- base – Base address of the PINT peripheral.
- bslice – Pattern match bit slice number.

**Return values**

status – = 0 Match has not been detected. = 1 Match has been detected.

```
static inline uint32_t PINT_PatternMatchGetStatusAll(PINT_Type *base)
```

Get status of all pattern match bit slices.

This function returns the status of all bit slices.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

status – Each bit position indicates the match status of corresponding bit slice.  
= 0 Match has not been detected. = 1 Match has been detected.

```
uint32_t PINT_PatternMatchResetDetectLogic(PINT_Type *base)
```

Reset pattern match detection logic.

This function resets the pattern match detection logic if any of the product term is matching.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

pmstatus – Each bit position indicates the match status of corresponding bit slice.  
= 0 Match was detected. = 1 Match was not detected.

```
static inline void PINT_PatternMatchEnable(PINT_Type *base)
```

Enable pattern match function.

This function enables the pattern match function.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

```
static inline void PINT_PatternMatchDisable(PINT_Type *base)
```

Disable pattern match function.

This function disables the pattern match function.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

```
static inline void PINT_PatternMatchEnableRXEV(PINT_Type *base)
```

Enable RXEV output.

This function enables the pattern match RXEV output.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

```
static inline void PINT_PatternMatchDisableRXEV(PINT_Type *base)
```

Disable RXEV output.

This function disables the pattern match RXEV output.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

void PINT\_EnableCallback(PINT\_Type \*base)

Enable callback.

This function enables the interrupt for the selected PINT peripheral. Although the pin(s) are monitored as soon as they are enabled, the callback function is not enabled until this function is called.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

void PINT\_DisableCallback(PINT\_Type \*base)

Disable callback.

This function disables the interrupt for the selected PINT peripheral. Although the pins are still being monitored but the callback function is not called.

**Parameters**

- base – Base address of the peripheral.

**Return values**

None. –

void PINT\_Deinit(PINT\_Type \*base)

Deinitialize PINT peripheral.

This function disables the PINT clock.

**Parameters**

- base – Base address of the PINT peripheral.

**Return values**

None. –

void PINT\_EnableCallbackByIndex(PINT\_Type \*base, *pint\_pin\_int\_t* pintIdx)

enable callback by pin index.

This function enables callback by pin index instead of enabling all pins.

**Parameters**

- base – Base address of the peripheral.
- pintIdx – pin index.

**Return values**

None. –

void PINT\_EnableInterruptByIndex(PINT\_Type \*base, *pint\_pin\_int\_t* pintIdx)

enable interrupt in NVIC by pin index.

This function enables the interrupt in the NVIC. The difference with PINT\_EnableCallbackByIndex() is that PINT\_EnableCallbackByIndex() not only enables the interrupt in the NVIC but also clears pending interrupts. Use this function together with PINT\_DisableInterruptByIndex() to temporarily disable/enable the pin interrupt. Use PINT\_EnableCallbackByIndex() to enable the interrupt after installing the callback.

**Parameters**

- base – Base address of the peripheral.

- pinIdx – pin index.

**Return values**

None. –

void PINT\_DisableInterruptByIndex(PINT\_Type \*base, *pint\_pin\_int\_t* pinIdx)

disable interrupt in NVIC by pin index.

This function disables the interrupt in the NVIC. The difference with PINT\_DisableCallbackByIndex() is that PINT\_DisableCallbackByIndex() not only disables the interrupt in the NVIC but also clears pending interrupts. Use this function together with PINT\_EnableInterruptByIndex() to temporarily disable/enable the pin interrupt. Use PINT\_DisableCallbackByIndex() to disable the interrupt in a de-init function.

**Parameters**

- base – Base address of the peripheral.
- pinIdx – pin index.

**Return values**

None. –

void PINT\_DisableCallbackByIndex(PINT\_Type \*base, *pint\_pin\_int\_t* pinIdx)

disable callback by pin index.

This function disables callback by pin index instead of disabling all pins.

**Parameters**

- base – Base address of the peripheral.
- pinIdx – 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>
```

## 2.16 Power Driver

```
enum pd_bits
    Values:
    enumerator kPDRUNCFG_PD_FRO_OUT
    enumerator kPDRUNCFG_PD_FRO
    enumerator kPDRUNCFG_PD_FLASH
    enumerator kPDRUNCFG_PD_BOD
    enumerator kPDRUNCFG_PD_ADC0
    enumerator kPDRUNCFG_PD_LPOSC
    enumerator kPDRUNCFG_PD_ACMP
    enumerator kPDRUNCFG_ForceUnsigned

enum __power_wakeup
    Deep sleep and power down mode wake up configurations.
    Values:
    enumerator kPDAWAKECFG_Wakeup_FRO_OUT
    enumerator kPDAWAKECFG_Wakeup_FRO
    enumerator kPDAWAKECFG_Wakeup_FLASH
    enumerator kPDAWAKECFG_Wakeup_BOD
    enumerator kPDAWAKECFG_Wakeup_ADC
    enumerator kPDAWAKECFG_Wakeup_LPOSC
    enumerator kPDAWAKECFG_Wakeup_ACMP

enum __power_dpd_wakeup_pin
    Deep power down mode wake up pins.
    Values:
    enumerator kPmu_Dpd_En_Pio0_15
    enumerator kPmu_Dpd_En_Pio0_9
    enumerator kPmu_Dpd_En_Pio0_8
    enumerator kPmu_Dpd_En_Pio0_17
    enumerator kPmu_Dpd_En_Pio0_13
    enumerator kPmu_Dpd_En_Pio0_4
```

enumerator kPmu\_Dpd\_En\_Pio0\_11

enumerator kPmu\_Dpd\_En\_Pio0\_10

enum \_\_power\_deep\_sleep\_active

Deep sleep/power down mode active part.

*Values:*

enumerator kPDSLEEPCFG\_DeepSleepBODActive

enumerator kPDSLEEPCFG\_DeepSleepLPOscActive

enum \_\_power\_gen\_reg

pmu general purpose register index

*Values:*

enumerator kPmu\_GenReg0

general purpose register0

enumerator kPmu\_GenReg1

general purpose register1

enumerator kPmu\_GenReg2

general purpose register2

enumerator kPmu\_GenReg3

general purpose register3

enumerator kPmu\_GenReg4

general purpose register4

enum \_\_power\_mode\_config

*Values:*

enumerator kPmu\_Sleep

enumerator kPmu\_Deep\_Sleep

enumerator kPmu\_PowerDown

enumerator kPmu\_Deep\_PowerDown

enum \_\_power\_bod\_reset\_level

BOD reset level, if VDD below reset level value, the reset will be asserted.

*Values:*

enumerator kBod\_ResetLevel0

BOD Reset Level0: 1.51V.

enum \_\_power\_bod\_interrupt\_level

BOD interrupt level, if VDD below interrupt level value, the BOD interrupt will be asserted.

*Values:*

enumerator kBod\_InterruptLevelReserved

BOD interrupt level reserved.

enumerator kBod\_InterruptLevel1

BOD interrupt level1: 2.24V.

enumerator kBod\_InterruptLevel2

BOD interrupt level2: 2.52V.

enumerator kBod\_InterruptLevel3

BOD interrupt level3: 2.81V.

typedef enum *pd\_bits* pd\_bit\_t

typedef enum *\_power\_gen\_reg* power\_gen\_reg\_t

pmu general purpose register index

typedef enum *\_power\_mode\_config* power\_mode\_cfg\_t

typedef enum *\_power\_bod\_reset\_level* power\_bod\_reset\_level\_t

BOD reset level, if VDD below reset level value, the reset will be asserted.

typedef enum *\_power\_bod\_interrupt\_level* power\_bod\_interrupt\_level\_t

BOD interrupt level, if VDD below interrupt level value, the BOD interrupt will be asserted.

FSL\_POWER\_DRIVER\_VERSION

power driver version 2.1.0.

PMUC\_PCON\_RESERVED\_MASK

PMU PCON reserved mask, used to clear reserved field which should not write 1.

POWER\_EnbaleLPO

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\_EnableLPO(bool enable)

API to enable LPO.

#### Parameters

- enable – true to enable LPO, false to disable LPO.

static inline void POWER\_WakeUpConfig(uint32\_t mask, bool powerDown)

API to config wakeup configurations for deep sleep mode and power down mode.

#### Parameters

- mask – wake up configurations for deep sleep mode and power down mode, reference *\_power\_wakeup*.
- powerDown – true is power down the mask part, false is powered part.

static inline void POWER\_DeepSleepConfig(uint32\_t mask, bool powerDown)

API to config active part for deep sleep mode and power down mode.

#### Parameters

- mask – active part configurations for deep sleep mode and power down mode, reference `_power_deep_sleep_active`.
- powerDown – true is power down the mask part, false is powered part.

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\_EnterSleep(void)

API to enter sleep power mode.

**Returns**

none

void POWER\_EnterDeepSleep(uint32\_t activePart)

API to enter deep sleep power mode.

**Parameters**

- activePart – should be a single or combine value of `_power_deep_sleep_active`.

**Returns**

none

void POWER\_EnterPowerDown(uint32\_t activePart)

API to enter power down mode.

**Parameters**

- activePart – should be a single or combine value of `_power_deep_sleep_active`.

**Returns**

none

void POWER\_EnterDeepPowerDownMode(void)

API to enter deep power down mode.

**Returns**

none

static inline uint32\_t POWER\_GetSleepModeFlag(void)

API to get sleep mode flag.

**Returns**

sleep mode flag: 0 is active mode, 1 is sleep mode entered.

static inline void POWER\_ClrSleepModeFlag(void)

API to clear sleep mode flag.

static inline uint32\_t POWER\_GetDeepPowerDownModeFlag(void)

API to get deep power down mode flag.

**Returns**

sleep mode flag: 0 not deep power down, 1 is deep power down mode entered.

static inline void POWER\_ClrDeepPowerDownModeFlag(void)

API to clear deep power down mode flag.

static inline void POWER\_ClrWakeupPinFlag(void)

API to clear wake up pin status flag.

static inline void POWER\_DeepPowerDownWakeupSourceSelect(uint32\_t wakeup\_pin)

static inline void POWER\_EnableNonDpd(bool enable)

API to enable non deep power down mode.

#### Parameters

- enable – true is enable non deep power down, otherwise disable.

static inline void POWER\_SetRetainData(power\_gen\_reg\_t index, uint32\_t data)

API to restore data to general purpose register which can be retained during deep power down mode.

#### Parameters

- index – general purpose data register index.
- data – data to restore.

static inline uint32\_t POWER\_GetRetainData(power\_gen\_reg\_t index)

API to get data from general purpose register which is retained during deep power down mode.

#### Parameters

- index – general purpose data register index.

#### Returns

data stored in the general purpose register.

static inline void POWER\_SetBodLevel(power\_bod\_reset\_level\_t resetLevel,  
power\_bod\_interrupt\_level\_t interruptLevel, bool  
enable)

Set Bod interrupt level and reset level.

#### Parameters

- resetLevel – BOD reset threshold level, please refer to power\_bod\_reset\_level\_t.
- interruptLevel – BOD interrupt threshold level, please refer to power\_bod\_interrupt\_level\_t.
- enable – Used to enable/disable the BOD interrupt and BOD reset.

## 2.17 Reset Driver

enum \_SYSCON\_RSTh

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control bits in PRESETCTRL/ASYNCPRESETCTRL registers

Values:

enumerator kFLASH\_RST\_N\_SHIFT\_RSTh

Flash controller reset control

```

enumerator kI2C0_RST_N_SHIFT_RSTn
    I2C0 reset control
enumerator kGPIO0_RST_N_SHIFT_RSTn
    GPIO0 reset control
enumerator kSWM_RST_N_SHIFT_RSTn
    SWM reset control
enumerator kWKT_RST_N_SHIFT_RSTn
    Self-wake-up timer(WKT) reset control
enumerator kMRT_RST_N_SHIFT_RSTn
    Multi-rate timer(MRT) reset control
enumerator kSPI0_RST_N_SHIFT_RSTn
    SPI0 reset control.
enumerator kCRC_RST_SHIFT_RSTn
    CRC reset control
enumerator kUART0_RST_N_SHIFT_RSTn
    UART0 reset control
enumerator kUART1_RST_N_SHIFT_RSTn
    UART1 reset control
enumerator kIOCON_RST_N_SHIFT_RSTn
    IOCON reset control
enumerator kACMP_RST_N_SHIFT_RSTn
    Analog comparator reset control
enumerator kADC_RST_N_SHIFT_RSTn
    ADC reset control
enumerator kCTIMER0_RST_N_SHIFT_RSTn
    CTIMER0 reset control
enumerator kGPIOINT_RST_N_SHIFT_RSTn
    GPIOINT reset control
enumerator kFRG0_RST_N_SHIFT_RSTn
    Fractional baud rate generator 0 reset control
enumerator kOTHER_RST_N_SHIFT_RSTn
    Some functions don't need reset
typedef enum _SYSCON_RSTn SYSCON_RSTn_t
    Enumeration for peripheral reset control bits.
    Defines the enumeration for peripheral reset control bits in PRESETCTRL/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

FLASH\_RSTS\_N

Array initializers with peripheral reset bits

I2C\_RSTS\_N

GPIO\_RSTS\_N

SWM\_RSTS\_N

WKT\_RSTS\_N

MRT\_RSTS\_N

SPI\_RSTS\_N

CRC\_RSTS\_N

UART\_RSTS\_N

IOCON\_RSTS\_N

ACMP\_RSTS\_N

ADC\_RSTS\_N

CTIMER\_RSTS\_N

GPIOINT\_RSTS\_N

FRG\_RSTS\_N

WWDT\_RSTS\_N

## 2.18 SPI: Serial Peripheral Interface Driver

### 2.19 SPI Driver

`void SPI_MasterGetDefaultConfig(spi_master_config_t *config)`

Sets the SPI master configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in `SPI_MasterInit()`. User may use the initialized structure unchanged in `SPI_MasterInit()`, or modify some fields of the structure before calling `SPI_MasterInit()`. After calling this API, the master is ready to transfer. Example:

```
spi_master_config_t config;
SPI_MasterGetDefaultConfig(&config);
```

#### Parameters

- `config` – pointer to master config structure

`status_t SPI_MasterInit(SPI_Type *base, const spi_master_config_t *config, uint32_t srcClock_Hz)`

Initializes the SPI with master configuration.

The configuration structure can be filled by user from scratch, or be set with default values by `SPI_MasterGetDefaultConfig()`. After calling this API, the slave is ready to transfer. Example

```
spi_master_config_t config = {
    .baudRate_Bps = 500000,
    ...
};
SPI_MasterInit(SPI0, &config);
```

#### Parameters

- `base` – SPI base pointer
- `config` – pointer to master configuration structure
- `srcClock_Hz` – Source clock frequency.

`void SPI_SlaveGetDefaultConfig(spi_slave_config_t *config)`

Sets the SPI slave configuration structure to default values.

The purpose of this API is to get the configuration structure initialized for use in `SPI_SlaveInit()`. Modify some fields of the structure before calling `SPI_SlaveInit()`. Example:

```
spi_slave_config_t config;
SPI_SlaveGetDefaultConfig(&config);
```

#### Parameters

- `config` – pointer to slave configuration structure

`status_t SPI_SlaveInit(SPI_Type *base, const spi_slave_config_t *config)`

Initializes the SPI with slave configuration.

The configuration structure can be filled by user from scratch or be set with default values by `SPI_SlaveGetDefaultConfig()`. After calling this API, the slave is ready to transfer. Example

```
spi_slave_config_t config = {  
    .polarity = kSPI_ClockPolarityActiveHigh;  
    .phase = kSPI_ClockPhaseFirstEdge;  
    .direction = kSPI_MsbFirst;  
    ...  
};  
SPI_SlaveInit(SPI0, &config);
```

### Parameters

- base – SPI base pointer
- config – pointer to slave configuration structure

void SPI\_Deinit(SPI\_Type \*base)

De-initializes the SPI.

Calling this API resets the SPI module, gates the SPI clock. Disable the fifo if enabled. The SPI module can't work unless calling the SPI\_MasterInit/SPI\_SlaveInit to initialize module.

### Parameters

- base – SPI base pointer

static inline void SPI\_Enable(SPI\_Type \*base, bool enable)

Enable or disable the SPI Master or Slave.

### Parameters

- base – SPI base pointer
- enable – or disable ( true = enable, false = disable)

static inline uint32\_t SPI\_GetStatusFlags(SPI\_Type \*base)

Gets the status flag.

### Parameters

- base – SPI base pointer

### Returns

SPI Status, use status flag to AND \_spi\_status\_flags could get the related status.

static inline void SPI\_ClearStatusFlags(SPI\_Type \*base, uint32\_t mask)

Clear the status flag.

### Parameters

- base – SPI base pointer
- mask – SPI Status, use status flag to AND \_spi\_status\_flags could get the related status.

static inline void SPI\_EnableInterrupts(SPI\_Type \*base, uint32\_t irq)

Enables the interrupt for the SPI.

### Parameters

- base – SPI base pointer
- irq – SPI interrupt source. The parameter can be any combination of the following values:
  - kSPI\_RxReadyInterruptEnable
  - kSPI\_TxReadyInterruptEnable

```
static inline void SPI_DisableInterrupts(SPI_Type *base, uint32_t irqs)
```

Disables the interrupt for the SPI.

#### Parameters

- base – SPI base pointer
- irqs – SPI interrupt source. The parameter can be any combination of the following values:
  - kSPI\_RxReadyInterruptEnable
  - kSPI\_TxReadyInterruptEnable

```
static inline bool SPI_IsMaster(SPI_Type *base)
```

Returns whether the SPI module is in master mode.

#### Parameters

- base – SPI peripheral address.

#### Returns

Returns true if the module is in master mode or false if the module is in slave mode.

```
status_t SPI_MasterSetBaudRate(SPI_Type *base, uint32_t baudrate_Bps, uint32_t srcClock_Hz)
```

Sets the baud rate for SPI transfer. This is only used in master.

#### Parameters

- base – SPI base pointer
- baudrate\_Bps – baud rate needed in Hz.
- srcClock\_Hz – SPI source clock frequency in Hz.

```
static inline void SPI_WriteData(SPI_Type *base, uint16_t data)
```

Writes a data into the SPI data register directly.

#### Parameters

- base – SPI base pointer
- data – needs to be write.

```
static inline void SPI_WriteConfigFlags(SPI_Type *base, uint32_t configFlags)
```

Writes a data into the SPI TXCTL register directly.

#### Parameters

- base – SPI base pointer
- configFlags – control command needs to be written.

```
void SPI_WriteDataWithConfigFlags(SPI_Type *base, uint16_t data, uint32_t configFlags)
```

Writes a data control info and data into the SPI TX register directly.

#### Parameters

- base – SPI base pointer
- data – value needs to be written.
- configFlags – control command needs to be written.

```
static inline uint32_t SPI_ReadData(SPI_Type *base)
```

Gets a data from the SPI data register.

#### Parameters

- base – SPI base pointer

**Returns**

Data in the register.

void SPI\_SetTransferDelay(SPI\_Type \*base, const *spi\_delay\_config\_t* \*config)

Set delay time for transfer. the delay uint is SPI clock time, maximum value is 0xF.

**Parameters**

- base – SPI base pointer
- config – configuration for delay option *spi\_delay\_config\_t*.

void SPI\_SetDummyData(SPI\_Type \*base, uint16\_t dummyData)

Set up the dummy data. This API can change the default data to be transferred when users set the tx buffer to NULL.

**Parameters**

- base – SPI peripheral address.
- dummyData – Data to be transferred when tx buffer is NULL.

*status\_t* SPI\_MasterTransferBlocking(SPI\_Type \*base, *spi\_transfer\_t* \*xfer)

Transfers a block of data using a polling method.

**Parameters**

- base – SPI base pointer
- xfer – pointer to *spi\_xfer\_config\_t* structure

**Return values**

- kStatus\_Success – Successfully start a transfer.
- kStatus\_InvalidArgument – Input argument is invalid.
- kStatus\_SPI\_Timeout – The transfer timed out and was aborted.

*status\_t* SPI\_MasterTransferCreateHandle(SPI\_Type \*base, *spi\_master\_handle\_t* \*handle, *spi\_master\_callback\_t* callback, void \*userData)

Initializes the SPI master handle.

This function initializes the SPI master handle which can be used for other SPI master transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

**Parameters**

- base – SPI peripheral base address.
- handle – SPI handle pointer.
- callback – Callback function.
- userData – User data.

*status\_t* SPI\_MasterTransferNonBlocking(SPI\_Type \*base, *spi\_master\_handle\_t* \*handle, *spi\_transfer\_t* \*xfer)

Performs a non-blocking SPI interrupt transfer.

**Parameters**

- base – SPI peripheral base address.
- handle – pointer to *spi\_master\_handle\_t* structure which stores the transfer state
- xfer – pointer to *spi\_xfer\_config\_t* 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_MasterTransferGetCount(SPI_Type *base, spi_master_handle_t *handle, size_t *count)`

Gets the master transfer count.

This function gets the master transfer count.

#### Parameters

- `base` – SPI peripheral base address.
- `handle` – Pointer to the `spi_master_handle_t` structure which stores the transfer state.
- `count` – The number of bytes transferred by using the non-blocking transaction.

#### Returns

status of `status_t`.

`void SPI_MasterTransferAbort(SPI_Type *base, spi_master_handle_t *handle)`

SPI master aborts a transfer using an interrupt.

This function aborts a transfer using an interrupt.

#### Parameters

- `base` – SPI peripheral base address.
- `handle` – Pointer to the `spi_master_handle_t` structure which stores the transfer state.

`void SPI_MasterTransferHandleIRQ(SPI_Type *base, spi_master_handle_t *handle)`

Interrupts the handler for the SPI.

#### Parameters

- `base` – SPI peripheral base address.
- `handle` – pointer to `spi_master_handle_t` structure which stores the transfer state.

`status_t SPI_SlaveTransferCreateHandle(SPI_Type *base, spi_slave_handle_t *handle, spi_slave_callback_t callback, void *userData)`

Initializes the SPI slave handle.

This function initializes the SPI slave handle which can be used for other SPI slave transactional APIs. Usually, for a specified SPI instance, call this API once to get the initialized handle.

#### Parameters

- `base` – SPI peripheral base address.
- `handle` – SPI handle pointer.
- `callback` – Callback function.
- `userData` – User data.

`status_t SPI_SlaveTransferNonBlocking(SPI_Type *base, spi_slave_handle_t *handle, spi_transfer_t *xfer)`

Performs a non-blocking SPI slave interrupt transfer.

---

**Note:** The API returns immediately after the transfer initialization is finished.

---

### Parameters

- `base` – SPI peripheral base address.
- `handle` – pointer to `spi_master_handle_t` structure which stores the transfer state
- `xfer` – pointer to `spi_xfer_config_t` 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.

```
static inline status_t SPI_SlaveTransferGetCount(SPI_Type *base, spi_slave_handle_t *handle, size_t *count)
```

Gets the slave transfer count.

This function gets the slave transfer count.

### Parameters

- `base` – SPI peripheral base address.
- `handle` – Pointer to the `spi_master_handle_t` structure which stores the transfer state.
- `count` – The number of bytes transferred by using the non-blocking transaction.

### Returns

status of `status_t`.

```
static inline void SPI_SlaveTransferAbort(SPI_Type *base, spi_slave_handle_t *handle)
```

SPI slave aborts a transfer using an interrupt.

This function aborts a transfer using an interrupt.

### Parameters

- `base` – SPI peripheral base address.
- `handle` – Pointer to the `spi_slave_handle_t` structure which stores the transfer state.

```
void SPI_SlaveTransferHandleIRQ(SPI_Type *base, spi_slave_handle_t *handle)
```

Interrupts a handler for the SPI slave.

### Parameters

- `base` – SPI peripheral base address.
- `handle` – pointer to `spi_slave_handle_t` structure which stores the transfer state

```
FSL_SPI_DRIVER_VERSION
```

SPI driver version.

```
enum _spi_xfer_option
```

SPI transfer option.

*Values:*

enumerator kSPI\_EndOfFrame

Add delay at the end of each frame(the last clk edge).

enumerator kSPI\_EndOfTransfer

Re-assert the CS signal after transfer finishes to deselect slave.

enumerator kSPI\_ReceiveIgnore

Ignore the receive data.

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\_ssel

Slave select.

*Values:*

enumerator kSPI\_Ssel0Assert

Slave select 0

enumerator kSPI\_SselDeAssertAll

enum \_spi\_spol

ssel polarity

*Values:*

enumerator kSPI\_Spol0ActiveHigh

enumerator kSPI\_Spol1ActiveHigh

enumerator kSPI\_Spol2ActiveHigh

enumerator kSPI\_Spol3ActiveHigh

enumerator kSPI\_SpolActiveAllHigh

enumerator kSPI\_SpolActiveAllLow

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

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\_RxReadyInterruptEnable  
Rx ready interrupt

enumerator kSPI\_TxReadyInterruptEnable  
Tx ready interrupt

enumerator kSPI\_RxOverrunInterruptEnable  
Rx overrun interrupt

enumerator kSPI\_TxUnderrunInterruptEnable  
Tx underrun interrupt

enumerator kSPI\_SlaveSelectAssertInterruptEnable  
Slave select assert interrupt

enumerator kSPI\_SlaveSelectDeassertInterruptEnable  
Slave select deassert interrupt

enumerator kSPI\_AllInterruptEnable

enum \_spi\_status\_flags  
SPI status flags.

*Values:*

enumerator kSPI\_RxReadyFlag  
Receive ready flag.

enumerator kSPI\_TxReadyFlag  
Transmit ready flag.

enumerator kSPI\_RxOverrunFlag  
Receive overrun flag.

enumerator kSPI\_TxUnderrunFlag  
Transmit underrun flag.

enumerator kSPI\_SlaveSelectAssertFlag  
Slave select assert flag.

enumerator kSPI\_SlaveSelectDeassertFlag  
slave select deassert flag.

enumerator kSPI\_StallFlag  
Stall flag.

enumerator kSPI\_EndTransferFlag  
End transfer bit.

enumerator kSPI\_MasterIdleFlag  
Master in idle status flag.

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\_ssel* spi\_ssel\_t  
 Slave select.

typedef enum *\_spi\_spol* spi\_spol\_t  
 ssel polarity

typedef enum *\_spi\_data\_width* spi\_data\_width\_t  
 Transfer data width.

typedef struct *\_spi\_delay\_config* spi\_delay\_config\_t  
 SPI delay time configure structure.

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\_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.

volatile uint16\_t s\_dummyData[]

uint32\_t SPI\_GetInstance(SPI\_Type \*base)  
 Returns instance number for SPI peripheral base address.

SPI\_DUMMYDATA  
 SPI dummy transfer data, the data is sent while txBuff is NULL.

FSL\_SDK\_ENABLE\_SPI\_DRIVER\_TRANSACTIONAL\_APIS

SPI\_RETRY\_TIMES  
 Retry times for waiting flag.

struct *\_spi\_delay\_config*  
*#include <fsl\_spi.h>* SPI delay time configure structure.

### 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

`uint32_t` `baudRate_Bps`  
 Baud Rate for SPI in Hz

`spi_clock_polarity_t` `clockPolarity`  
 Clock polarity

`spi_clock_phase_t` `clockPhase`  
 Clock phase

`spi_shift_direction_t` `direction`  
 MSB or LSB

`uint8_t` `dataWidth`  
 Width of the data

`spi_ssel_t` `sselNumber`  
 Slave select number

`spi_spol_t` `sselPolarity`  
 Configure active CS polarity

`spi_delay_config_t` `delayConfig`  
 Configure for delay time.

`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` `clockPolarity`  
 Clock polarity

`spi_clock_phase_t` `clockPhase`  
 Clock phase

`spi_shift_direction_t` `direction`  
 MSB or LSB

uint8\_t dataWidth

Width of the data

*spi\_spol\_t* sselPolarity

Configure active CS polarity

struct *\_spi\_transfer*

*#include <fsl\_spi.h>* SPI transfer structure.

### Public Members

const uint8\_t \*txData

Send buffer

uint8\_t \*rxData

Receive buffer

size\_t dataSize

Transfer bytes

uint32\_t configFlags

Additional option to control transfer *\_spi\_xfer\_option*.

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]

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]

uint32\_t lastCommand

Last command for transfer.

## 2.20 SWM: Switch Matrix Module

enum `_swm_port_pin_type_t`

SWM port\_pin number.

*Values:*

enumerator `kSWM_PortPin_P0_0`  
port\_pin number `P0_0`.

enumerator `kSWM_PortPin_P0_1`  
port\_pin number `P0_1`.

enumerator `kSWM_PortPin_P0_2`  
port\_pin number `P0_2`.

enumerator `kSWM_PortPin_P0_3`  
port\_pin number `P0_3`.

enumerator `kSWM_PortPin_P0_4`  
port\_pin number `P0_4`.

enumerator `kSWM_PortPin_P0_5`  
port\_pin number `P0_5`.

enumerator `kSWM_PortPin_P0_6`  
port\_pin number `P0_6`.

enumerator `kSWM_PortPin_P0_7`  
port\_pin number `P0_7`.

enumerator `kSWM_PortPin_P0_8`  
port\_pin number `P0_8`.

enumerator `kSWM_PortPin_P0_9`  
port\_pin number `P0_9`.

enumerator `kSWM_PortPin_P0_10`  
port\_pin number `P0_10`.

enumerator `kSWM_PortPin_P0_11`  
port\_pin number `P0_11`.

enumerator `kSWM_PortPin_P0_12`  
port\_pin number `P0_12`.

enumerator `kSWM_PortPin_P0_13`  
port\_pin number `P0_13`.

enumerator `kSWM_PortPin_P0_14`  
port\_pin number `P0_14`.

enumerator `kSWM_PortPin_P0_15`  
port\_pin number `P0_15`.

enumerator `kSWM_PortPin_P0_16`  
port\_pin number `P0_16`.

enumerator `kSWM_PortPin_P0_17`  
port\_pin number `P0_17`.

enumerator kSWM\_PortPin\_Reset  
port\_pin reset number.

enum \_swm\_select\_movable\_t  
SWM movable selection.

*Values:*

enumerator kSWM\_USART0\_TXD  
Movable function as USART0\_TXD.

enumerator kSWM\_USART0\_RXD  
Movable function as USART0\_RXD.

enumerator kSWM\_USART0\_RTS  
Movable function as USART0\_RTS.

enumerator kSWM\_USART0\_CTS  
Movable function as USART0\_CTS.

enumerator kSWM\_USART0\_SCLK  
Movable function as USART0\_SCLK.

enumerator kSWM\_USART1\_TXD  
Movable function as USART1\_TXD.

enumerator kSWM\_USART1\_RXD  
Movable function as USART1\_RXD.

enumerator kSWM\_USART1\_SCLK  
Movable function as USART1\_SCLK.

enumerator kSWM\_SPI0\_SCK  
Movable function as SPI0\_SCK.

enumerator kSWM\_SPI0\_MOSI  
Movable function as SPI0\_MOSI.

enumerator kSWM\_SPI0\_MISO  
Movable function as SPI0\_MISO.

enumerator kSWM\_SPI0\_SSEL0  
Movable function as SPI0\_SSEL0.

enumerator kSWM\_SPI0\_SSEL1  
Movable function as SPI0\_SSEL1.

enumerator kSWM\_T0\_CAP\_CHN0  
Movable function as Timer Capture Channel 0.

enumerator kSWM\_T0\_CAP\_CHN1  
Movable function as Timer Capture Channel 1.

enumerator kSWM\_T0\_CAP\_CHN2  
Movable function as Timer Capture Channel 2.

enumerator kSWM\_T0\_MAT\_CHN0  
Movable function as Timer Match Channel 0.

enumerator kSWM\_T0\_MAT\_CHN1  
Movable function as Timer Match Channel 1.

enumerator kSWM\_T0\_MAT\_CHN2  
Movable function as Timer Match Channel 2.

enumerator kSWM\_T0\_MAT\_CHN3  
Movable function as Timer Match Channel 3.

enumerator kSWM\_I2C0\_SDA  
Movable function as I2C1\_SDA.

enumerator kSWM\_I2C0\_SCL  
Movable function as I2C1\_SCL.

enumerator kSWM\_ACMP\_OUT  
Movable function as ACMP\_OUT.

enumerator kSWM\_CLKOUT  
Movable function as CLKOUT.

enumerator kSWM\_GPIO\_INT\_BMAT  
Movable function as GPIO\_INT\_BMAT.

enumerator kSWM\_LVLSHFT\_IN0  
Movable function as LVLSHFT\_IN0.

enumerator kSWM\_LVLSHFT\_IN1  
Movable function as LVLSHFT\_IN1.

enumerator kSWM\_LVLSHFT\_OUT0  
Movable function as LVLSHFT\_OUT0.

enumerator kSWM\_LVLSHFT\_OUT1  
Movable function as LVLSHFT\_OUT1.

enumerator kSWM\_MOVABLE\_NUM\_FUNCS  
Movable function number.

enum \_swm\_select\_fixed\_pin\_t  
SWM fixed pin selection.

*Values:*

enumerator kSWM\_ACMP\_INPUT1  
Fixed-pin function as ACMP\_INPUT1.

enumerator kSWM\_ACMP\_INPUT2  
Fixed-pin function as ACMP\_INPUT2.

enumerator kSWM\_ACMP\_INPUT3  
Fixed-pin function as ACMP\_INPUT3.

enumerator kSWM\_ACMP\_INPUT4  
Fixed-pin function as ACMP\_INPUT4.

enumerator kSWM\_SWCLK  
Fixed-pin function as SWCLK.

enumerator kSWM\_SWDIO  
Fixed-pin function as SWDIO.

enumerator kSWM\_RESETN  
Fixed-pin function as RESETN.

enumerator `kSWM_CLKIN`  
Fixed-pin function as CLKIN.

enumerator `kSWM_WKCLKIN`  
Fixed-pin function as CLKIN.

enumerator `kSWM_VDDCMP`  
Fixed-pin function as VDDCMP.

enumerator `kSWM_ADC_CHN0`  
Fixed-pin function as ADC\_CHN0.

enumerator `kSWM_ADC_CHN1`  
Fixed-pin function as ADC\_CHN1.

enumerator `kSWM_ADC_CHN2`  
Fixed-pin function as ADC\_CHN2.

enumerator `kSWM_ADC_CHN3`  
Fixed-pin function as ADC\_CHN3.

enumerator `kSWM_ADC_CHN4`  
Fixed-pin function as ADC\_CHN4.

enumerator `kSWM_ADC_CHN5`  
Fixed-pin function as ADC\_CHN5.

enumerator `kSWM_ADC_CHN6`  
Fixed-pin function as ADC\_CHN6.

enumerator `kSWM_ADC_CHN7`  
Fixed-pin function as ADC\_CHN7.

enumerator `kSWM_ADC_CHN8`  
Fixed-pin function as ADC\_CHN8.

enumerator `kSWM_ADC_CHN9`  
Fixed-pin function as ADC\_CHN9.

enumerator `kSWM_ADC_CHN10`  
Fixed-pin function as ADC\_CHN10.

enumerator `kSWM_ADC_CHN11`  
Fixed-pin function as ADC\_CHN11.

enumerator `kSWM_FIXEDPIN_NUM_FUNCS`  
Fixed-pin function number.

typedef enum `_swm_port_pin_type_t` `swm_port_pin_type_t`  
SWM port\_pin number.

typedef enum `_swm_select_movable_t` `swm_select_movable_t`  
SWM movable selection.

typedef enum `_swm_select_fixed_pin_t` `swm_select_fixed_pin_t`  
SWM fixed pin selection.

FSL\_SWM\_DRIVER\_VERSION  
LPC SWM driver version.

```
void SWM_SetMovablePinSelect(SWM_Type *base, swm_select_movable_t func,
                             swm_port_pin_type_t swm_port_pin)
```

Assignment of digital peripheral functions to pins.

This function will selects a pin (designated by its GPIO port and bit numbers) to a function.

#### Parameters

- base – SWM peripheral base address.
- func – any function name that is movable.
- swm\_port\_pin – any pin which has a GPIO port number and bit number.

```
void SWM_SetFixedPinSelect(SWM_Type *base, swm_select_fixed_pin_t func, bool enable)
```

Enable the fixed-pin function.

This function will enables a fixed-pin function in PINENABLE0 or PINENABLE1.

#### Parameters

- base – SWM peripheral base address.
- func – any function name that is fixed pin.
- enable – enable or disable.

## 2.21 SYSCON: System Configuration

```
enum _syscon_connection_t
```

SYSCON connections type.

*Values:*

```
enumerator kSYSCON_GpioPort0Pin0ToPintsel
```

Pin Interrupt.

```
enumerator kSYSCON_GpioPort0Pin1ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin2ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin3ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin4ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin5ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin7ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin8ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin9ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin10ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin11ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin12ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin13ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin14ToPintsel
```

```
enumerator kSYSCON_GpioPort0Pin15ToPintsel
```

enumerator kSYSCON\_GpioPort0Pin16ToPintsel

enumerator kSYSCON\_GpioPort0Pin17ToPintsel

typedef enum *\_syscon\_connection\_t* syscon\_connection\_t  
SYSCON connections type.

PINTSEL\_ID

Periphinmux IDs.

SYSCON\_SHIFT

FSL\_SYSCON\_DRIVER\_VERSION

Group syscon driver version for SDK.

void SYSCON\_AttachSignal(SYSCON\_Type \*base, uint16\_t index, *syscon\_connection\_t* connection)

Attaches a signal.

This function gates the SYSCON clock.

#### Parameters

- base – Base address of the SYSCON peripheral.
- index – Destination peripheral to attach the signal to.
- connection – Selects connection.

#### Return values

None. –

## 2.22 USART: Universal Asynchronous Receiver/Transmitter Driver

### 2.23 USART Driver

uint32\_t USART\_GetInstance(USART\_Type \*base)

Returns instance number for USART peripheral base address.

*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 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 = 9600U`; `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, 2000000U);
```

**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.

`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 RX is ready:

```
if (kUSART_RxReady & 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 For example:

```
USART_ClearStatusFlags(USART1, kUSART_HardwareOverrunFlag)
```

**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 ready interrupt and RX ready interrupt:

```
USART_EnableInterrupts(USART1, kUSART_RxReadyInterruptEnable | kUSART_
↳TxReadyInterruptEnable);
```

**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 ready interrupt and RX ready interrupt:

```
USART_DisableInterrupts(USART1, kUSART_TxReadyInterruptEnable | kUSART_
↳RxReadyInterruptEnable);
```

**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_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 cuntion, 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_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_EnableTx(USART_Type *base, bool enable)
```

Enable the USART transmit.

This function will enable or disable the USART transmit.

**Parameters**

- base – USART peripheral base address.
- enable – true for enable and false for disable.

```
static inline void USART_EnableRx(USART_Type *base, bool enable)
```

Enable the USART receive.

This function will enable or disable the USART receive. Note: if the transmit is enabled, the receive will not be disabled.

**Parameters**

- base – USART peripheral base address.
- enable – true for enable and false for disable.

```
static inline void USART_WriteByte(USART_Type *base, uint8_t data)
```

Writes to the TXDAT register.

This function will writes data to the TXDAT automatly.The upper layer must ensure that TXDATA 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 RXDAT directly.

This function reads data from the RXDAT automatly. The upper layer must ensure that the RXDAT is not empty before calling this function.

**Parameters**

- base – USART peripheral base address.

**Returns**

The byte read from USART data register.

*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.

**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\_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.

**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 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.

---

**Note:** The kStatus\_USART\_TxIdle is passed to the upper layer when all data is written to the TX register. However it does not ensure that all data are sent out. Before disabling the TX, check the kUSART\_TransmissionCompleteFlag to ensure that the TX is finished.

---

#### 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 written to USART TX register.

This function gets the number of bytes that have been written to USART TX register 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 tx.

enumerator kStatus\_USART\_RxError  
Error happens on rx.

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\_HardwareOverrun  
USART hardware over flow.

enumerator kStatus\_USART\_BaudrateNotSupport  
Baudrate is not support in current clock source

enumerator kStatus\_USART\_Timeout  
USART times out.

enum \_usart\_parity\_mode  
USART parity mode.

*Values:*

enumerator kUSART\_ParityDisabled  
Parity disabled

enumerator kUSART\_ParityEven  
Parity enabled, type even, bit setting: PARITYSEL = 10

enumerator kUSART\_ParityOdd  
Parity enabled, type odd, bit setting: PARITYSEL = 11

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\_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\_interrupt\_enable

USART interrupt configuration structure, default settings all disabled.

*Values:*

enumerator kUSART\_RxReadyInterruptEnable

Receive ready interrupt.

enumerator kUSART\_TxReadyInterruptEnable

Transmit ready interrupt.

enumerator kUSART\_TxIdleInterruptEnable

Transmit idle interrupt.

enumerator kUSART\_DeltaCtsInterruptEnable

Cts pin change interrupt.

enumerator kUSART\_TxDisableInterruptEnable

Transmit disable interrupt.

enumerator kUSART\_HardwareOverRunInterruptEnable

hardware ove run interrupt.

enumerator kUSART\_RxBreakInterruptEnable

Receive break interrupt.

enumerator kUSART\_RxStartInterruptEnable

Receive ready interrupt.

enumerator kUSART\_FramErrorInterruptEnable

Receive start interrupt.

enumerator kUSART\_ParityErrorInterruptEnable

Receive frame error interrupt.

enumerator kUSART\_RxNoiseInterruptEnable  
Receive noise error interrupt.

enumerator kUSART\_AutoBaudErrorInterruptEnable  
Receive auto baud error interrupt.

enumerator kUSART\_AllInterruptEnable  
All interrupt.

enum \_usart\_flags

USART status flags.

This provides constants for the USART status flags for use in the USART functions.

*Values:*

enumerator kUSART\_RxReady  
Receive ready flag.

enumerator kUSART\_RxIdleFlag  
Receive IDLE flag.

enumerator kUSART\_TxReady  
Transmit ready flag.

enumerator kUSART\_TxIdleFlag  
Transmit idle flag.

enumerator kUSART\_CtsState  
Cts pin status.

enumerator kUSART\_DeltaCtsFlag  
Cts pin change flag.

enumerator kUSART\_TxDisableFlag  
Transmit disable flag.

enumerator kUSART\_HardwareOverrunFlag  
Hardware over run flag.

enumerator kUSART\_RxBreakFlag  
Receive break flag.

enumerator kUSART\_RxStartFlag  
receive start flag.

enumerator kUSART\_FramErrorFlag  
Frame error flag.

enumerator kUSART\_ParityErrorFlag  
Parity error flag.

enumerator kUSART\_RxNoiseFlag  
Receive noise flag.

enumerator kUSART\_AutoBaudErrorFlag  
Auto baud error flag.

typedef enum *\_usart\_parity\_mode* usart\_parity\_mode\_t  
USART parity mode.

typedef enum *\_usart\_sync\_mode* usart\_sync\_mode\_t  
USART synchronous 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 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.

FSL_SDK_ENABLE_USART_DRIVER_TRANSACTIONAL_APIS
    Macro gate for enable transaction API. 1 for enable, 0 for disable.

FSL_SDK_USART_DRIVER_ENABLE_BAUDRATE_AUTO_GENERATE
    USART baud rate auto generate switch gate. 1 for enable, 0 for disable.

UART_RETRY_TIMES
    Retry times for waiting flag.
    Defining to zero means to keep waiting for the flag until it is assert/deassert.

struct _usart_config
    #include <fsl_usart.h> USART configuration structure.
```

### Public Members

```
uint32_t baudRate_Bps
    USART baud rate

bool enableRx
    USART receive enable.

bool enableTx
    USART transmit enable.

bool loopback
    Enable peripheral loopback

bool enableContinuousSCLK
    USART continuous Clock generation enable in synchronous master mode.

bool enableHardwareFlowControl
    Enable hardware control RTS/CTS

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

*usart\_sync\_mode\_t* syncMode

Transfer mode - asynchronous, synchronous master, synchronous slave.

*usart\_clock\_polarity\_t* clockPolarity

Selects the clock polarity and sampling edge in sync 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
union __unnamed6__
```

### 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.
```

## 2.24 WKT: Self-wake-up Timer

```
void WKT_Init(WKT_Type *base, const wkt_config_t *config)
    Ungates the WKT clock and configures the peripheral for basic operation.
```

---

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

---

### Parameters

- base – WKT peripheral base address
- config – Pointer to user's WKT config structure.

```
void WKT_Deinit(WKT_Type *base)
    Gate the WKT clock.
```

### Parameters

- base – WKT peripheral base address

```
static inline void WKT_GetDefaultConfig(wkt_config_t *config)
    Initializes the WKT configuration structure.
```

This function initializes the WKT configuration structure to default values. The default values are as follows.

```
config->clockSource = kWKT_DividedFROClockSource;
```

### See also:

wkt\_config\_t

### Parameters

- config – Pointer to the WKT configuration structure.

```
static inline uint32_t WKT_GetCounterValue(WKT_Type *base)
    Read actual WKT counter value.
```

### Parameters

- base – WKT peripheral base address

```
static inline uint32_t WKT_GetStatusFlags(WKT_Type *base)
```

Gets the WKT status flags.

**Parameters**

- base – WKT peripheral base address

**Returns**

The status flags. This is the logical OR of members of the enumeration `wkt_status_flags_t`

```
static inline void WKT_ClearStatusFlags(WKT_Type *base, uint32_t mask)
```

Clears the WKT status flags.

**Parameters**

- base – WKT peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `wkt_status_flags_t`

```
static inline void WKT_StartTimer(WKT_Type *base, uint32_t count)
```

Starts the timer counting.

After calling this function, timer loads a count value, counts down to 0, then stops.

---

**Note:** User can call the utility macros provided in `fsl_common.h` to convert to ticks Do not write to Counter register while the counting is in progress

---

**Parameters**

- base – WKT peripheral base address.
- count – The value to be loaded into the WKT Count register

```
static inline void WKT_StopTimer(WKT_Type *base)
```

Stops the timer counting.

This function Clears the counter and stops the timer from counting.

**Parameters**

- base – WKT peripheral base address

```
FSL_WKT_DRIVER_VERSION
```

Version 2.0.2

```
enum _wkt_clock_source
```

Describes WKT clock source.

*Values:*

```
enumerator kWKT_DividedFROClockSource
```

WKT clock sourced from the divided FRO clock

```
enumerator kWKT_LowPowerClockSource
```

WKT clock sourced from the Low power clock Use this clock, LPOSCEN bit of DPDCTRL register must be enabled

```
enumerator kWKT_ExternalClockSource
```

WKT clock sourced from the Low power clock Use this clock, WAKECLKPAD\_DISABLE bit of DPDCTRL register must be enabled

enum `_wkt_status_flags`

List of WKT flags.

*Values:*

enumerator `kWKT_AlarmFlag`

Alarm flag

typedef enum `_wkt_clock_source` `wkt_clock_source_t`

Describes WKT clock source.

typedef struct `_wkt_config` `wkt_config_t`

Describes WKT configuration structure.

typedef enum `_wkt_status_flags` `wkt_status_flags_t`

List of WKT flags.

struct `_wkt_config`

*#include* `<fsl_wkt.h>` Describes WKT configuration structure.

### Public Members

`wkt_clock_source_t` `clockSource`

External or internal clock source select

## 2.25 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 = 0xFFFFFFFFU;
config->timeoutValue = 0xFFFFFFFFU;
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 W DEN written with a 1, it can not be re-written with a 0.

This function write value into WWDT\_MOD register to disable the WWDT.

### Parameters

- base – WWDT peripheral base address

static inline uint32\_t WWDT\_GetStatusFlags(WWDT\_Type \*base)

Gets all WWDT status flags.

This function gets all status flags.

Example for getting Timeout Flag:

```
uint32_t status;
status = WWDT_GetStatusFlags(wwdt_base) & kWWDT_TimeoutFlag;
```

### Parameters

- base – WWDT peripheral base address

### Returns

The status flags. This is the logical OR of members of the enumeration `_wwdt_status_flags_t`

```
void WWDT_ClearStatusFlags(WWDT_Type *base, uint32_t mask)
```

Clear WWDT flag.

This function clears WWDT status flag.

Example for clearing warning flag:

```
WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
```

#### Parameters

- base – WWDT peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `_wwdt_status_flags_t`

```
static inline void WWDT_SetWarningValue(WWDT_Type *base, uint32_t warningValue)
```

Set the WWDT warning value.

The `WDWARNINT` register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by `WARNINT`, an interrupt will be generated after the subsequent `WDCLK`.

#### Parameters

- base – WWDT peripheral base address
- warningValue – WWDT warning value.

```
static inline void WWDT_SetTimeoutValue(WWDT_Type *base, uint32_t timeoutCount)
```

Set the WWDT timeout value.

This function sets the timeout value. Every time a feed sequence occurs the value in the `TC` register is loaded into the Watchdog timer. Writing a value below `0xFF` will cause `0xFF` to be loaded into the `TC` register. Thus the minimum time-out interval is `TWDCLK*256*4`. If `enableWatchdogProtect` flag is true in `wwdt_config_t` config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the `WDTOF` flag.

#### Parameters

- base – WWDT peripheral base address
- timeoutCount – WWDT timeout value, count of WWDT clock tick.

```
static inline void WWDT_SetWindowValue(WWDT_Type *base, uint32_t windowValue)
```

Sets the WWDT window value.

The `WINDOW` register determines the highest `TV` value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in `WINDOW`, a watchdog event will occur. To disable windowing, set `windowValue` to `0xFFFFFFFF` (maximum possible timer value) so windowing is not in effect.

#### Parameters

- base – WWDT peripheral base address
- windowValue – WWDT window value.

```
void WWDT_Refresh(WWDT_Type *base)
```

Refreshes the WWDT timer.

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

#### Parameters

- base – WWDT peripheral base address

FSL\_WWDT\_DRIVER\_VERSION

Defines WWDT driver version.

WWDT\_FIRST\_WORD\_OF\_REFRESH

First word of refresh sequence

WWDT\_SECOND\_WORD\_OF\_REFRESH

Second word of refresh sequence

enum `_wwdt_status_flags_t`

WWDT status flags.

This structure contains the WWDT status flags for use in the WWDT functions.

*Values:*

enumerator `kWWDT_TimeoutFlag`

Time-out flag, set when the timer times out

enumerator `kWWDT_WarningFlag`

Warning interrupt flag, set when timer is below the value `WDWARNINT`

typedef struct `_wwdt_config` `wwdt_config_t`

Describes WWDT configuration structure.

struct `_wwdt_config`

*#include <fsl\_wwdt.h>* Describes WWDT configuration structure.

### Public Members

bool `enableWwdt`

Enables or disables WWDT

bool `enableWatchdogReset`

true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset

bool `enableWatchdogProtect`

true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time

bool `enableLockOscillator`

true: Disabling or powering down the watchdog oscillator is prevented Once set, this bit can only be cleared by a reset false: Do not lock oscillator

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 Motor Control

#### 3.1.1 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\_pd\_bdm** 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 */
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 */
```

(continues on next page)

(continued from previous page)

```

/* 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 responseDataAddr, FMSTR_SIZE responseDataLen);
```

- 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 <a href="#">FM-STR_PipeOpen</a> 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>FM-STR_U16</i>	Unsigned 16-bit integer.
<i>FM-STR_U32</i>	Unsigned 32-bit integer.
<i>FMSTR_S8</i>	Signed 8-bit integer.
<i>FM-STR_S16</i>	Signed 16-bit integer.
<i>FM-STR_S32</i>	Signed 32-bit integer.
<i>FM-STR_FLOAT</i>	4-byte standard IEEE floating-point type.
<i>FM-STR_FLAGS</i>	Data type forming a union with a structure of flag bit-fields.
<i>FM-STR_SIZES</i>	Data type holding a general size value, at least 8 bits wide.
<i>FM-STR_INDEX</i>	General for-loop index. Must be signed, at least 16 bits wide.
<i>FM-STR_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>FM-STR_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](http://www.nxp.com/freemaster)
- FreeMASTER community area: [community.nxp.com/community/freemaster](http://community.nxp.com/community/freemaster)
- FreeMASTER GitHub code repo: <https://github.com/nxp-mcuxpresso/mcux-freemaster>
- MCUXpresso SDK home: [www.nxp.com/mcuxpresso](http://www.nxp.com/mcuxpresso)
- MCUXpresso SDK builder: [mcuxpresso.nxp.com/en](http://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.

# Chapter 4

## RTOS

### 4.1 FreeRTOS

#### 4.1.1 FreeRTOS kernel

Open source RTOS kernel for small devices.

[FreeRTOS kernel for MCUXpresso SDK Readme](#)

[FreeRTOS kernel for MCUXpresso SDK ChangeLog](#)

[FreeRTOS kernel Readme](#)

#### 4.1.2 FreeRTOS drivers

This is set of NXP provided FreeRTOS reentrant bus drivers.

#### 4.1.3 backoffalgorithm

Algorithm for calculating exponential backoff with jitter for network retry attempts.

[Readme](#)

#### 4.1.4 corehttp

C language HTTP client library designed for embedded platforms.

#### 4.1.5 corejson

JSON parser.

**Readme**

#### **4.1.6 coremqtt**

MQTT publish/subscribe messaging library.

#### **4.1.7 corepkcs11**

PKCS #11 key management library.

**Readme**

#### **4.1.8 freertos-plus-tcp**

Open source RTOS FreeRTOS Plus TCP.

**Readme**