



MCUXpresso SDK Documentation

Release 26.03.00-pvw2



NXP
Feb 20, 2026



Table of contents

1	LPCXpresso860MAX	3
1.1	Overview	3
1.2	Getting Started with MCUXpresso SDK Package	3
1.2.1	Getting Started with MCUXpresso SDK Package	3
1.3	Getting Started with MCUXpresso SDK GitHub	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	69
1.5.1	MCUXpresso SDK Changelog	69
1.6	Driver API Reference Manual	105
1.7	Middleware Documentation	105
1.7.1	FreeMASTER	106
1.7.2	FreeRTOS	106
2	LPC865	107
2.1	Clock Driver	107
2.2	CRC: Cyclic Redundancy Check Driver	118
2.3	DMA: Direct Memory Access Controller Driver	121
2.4	FTM: FlexTimer Driver	138
2.5	I2C: Inter-Integrated Circuit Driver	161
2.6	I2C Driver	161
2.7	I2C Master Driver	162
2.8	I2C Slave Driver	171
2.9	I3C: I3C Driver	180
2.10	I3C Common Driver	183
2.11	I3C Master DMA Driver	185
2.12	I3C Master Driver	188
2.13	I3C Slave DMA Driver	214
2.14	I3C Slave Driver	216
2.15	IAP: In Application Programming Driver	229
2.16	INPUTMUX: Input Multiplexing Driver	235
2.17	Common Driver	237
2.18	LPC_ACOMP: Analog comparator Driver	252
2.19	ADC: 12-bit SAR Analog-to-Digital Converter Driver	254
2.20	GPIO: General Purpose I/O	266
2.21	IOCON: I/O pin configuration	268
2.22	MRT: Multi-Rate Timer	269
2.23	PINT: Pin Interrupt and Pattern Match Driver	273
2.24	Power	282
2.25	Reset	287
2.26	SPI: Serial Peripheral Interface Driver	290
2.27	SPI Driver	290
2.28	SWM: Switch Matrix Module	302
2.29	SYSCON: System Configuration	310
2.30	USART: Universal Asynchronous Receiver/Transmitter Driver	313

2.31	USART Driver	313
2.32	WKT: Self-wake-up Timer	326
2.33	WWDT: Windowed Watchdog Timer Driver	328
3	Middleware	333
3.1	Motor Control	333
3.1.1	FreeMASTER	333
4	RTOS	371
4.1	FreeRTOS	371
4.1.1	FreeRTOS kernel	371
4.1.2	FreeRTOS drivers	371
4.1.3	backoffalgorithm	371
4.1.4	corehttp	371
4.1.5	corejson	371
4.1.6	coremqtt	372
4.1.7	corepkcs11	372
4.1.8	freertos-plus-tcp	372

This documentation contains information specific to the lpcxpresso860max board.

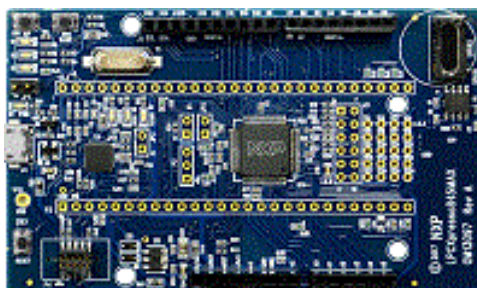
Chapter 1

LPCXpresso860MAX

1.1 Overview

The LPC86x are an Arm Cortex-M0+ based, low-cost 32-bit MCU family operating at CPU frequencies of up to 48 MHz. The LPC86x support up to 64 KB of flash memory and 8 KB of SRAM.

The peripheral complement of the LPC86x includes a CRC engine, one I2C-bus interface, one I3C-MIPI bus interface, up to three USARTs, up to two SPI interfaces, one multi-rate timer, self-wake-up timer, two FlexTimers, a DMA, one 12-bit ADC, one analog comparator, function-configurable I/O ports through a switch matrix, an input pattern match engine, and up to 54 general-purpose I/O pins.



MCU device and part on board is shown below:

- Device: LPC865
- PartNumber: LPC865M201JBD64

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.

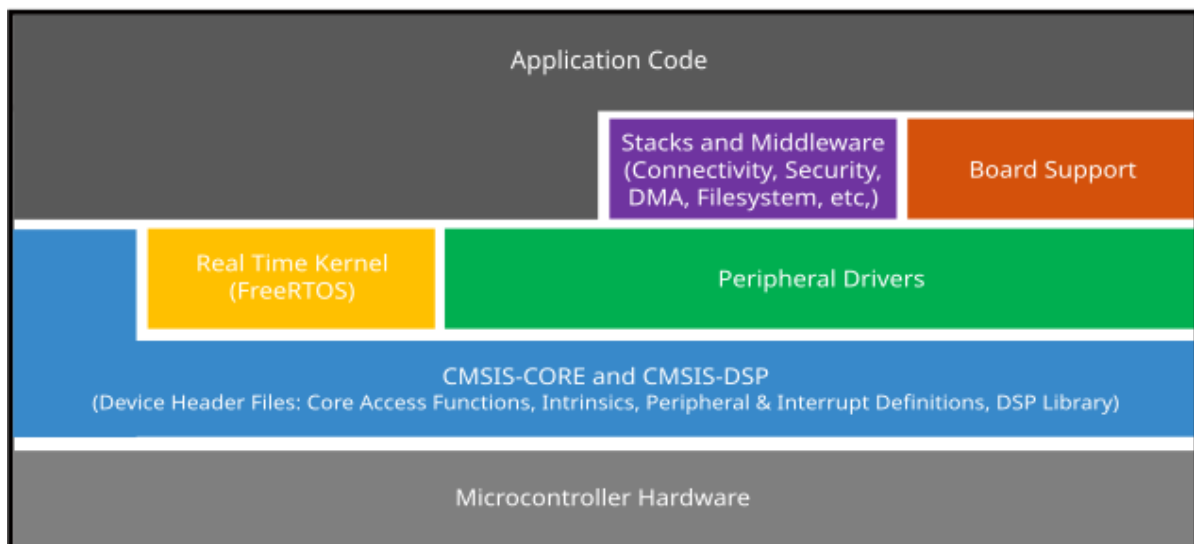
Note: The Repository-Layout SDK package was first introduced in version 25.09.00, but initially only for MCXW23x platforms.

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

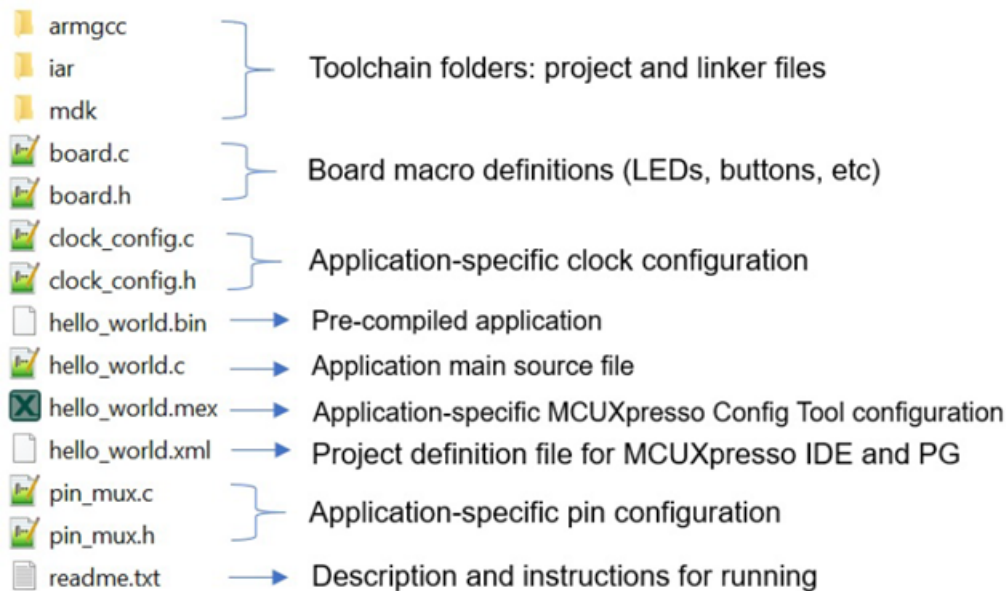
but there are cases where multiple peripherals are used (for example, SPI conversion using DMA).

- `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.

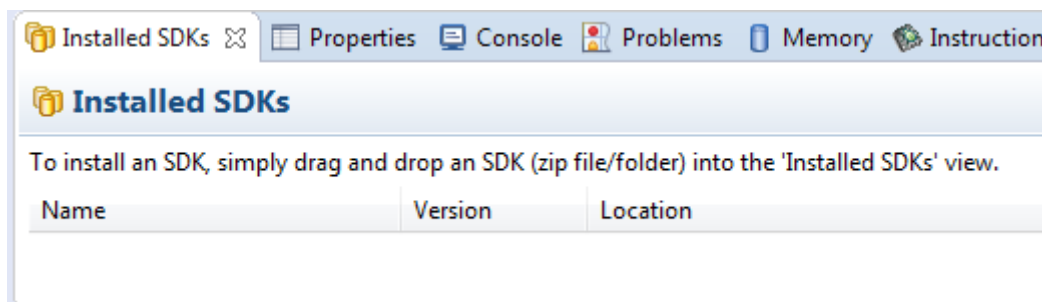
Run a demo using MCUXpresso IDE **Note:** Ensure that the MCUXpresso IDE toolchain is included when generating the MCUXpresso SDK package.

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.

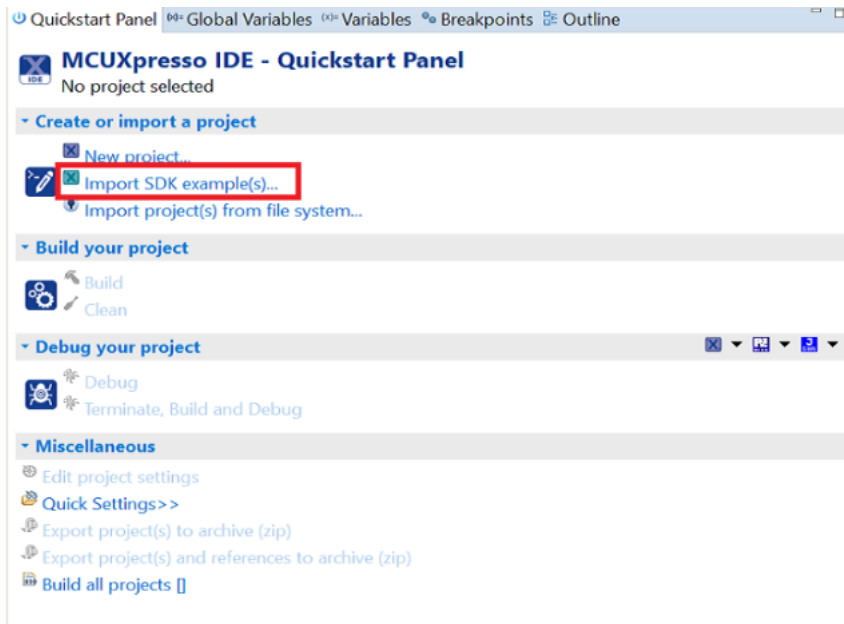
Select the workspace location Every time MCUXpresso IDE launches, it prompts the user to select a workspace location. MCUXpresso IDE is built on top of Eclipse which uses workspace to store information about its current configuration, and in some use cases, source files for the projects are in the workspace.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

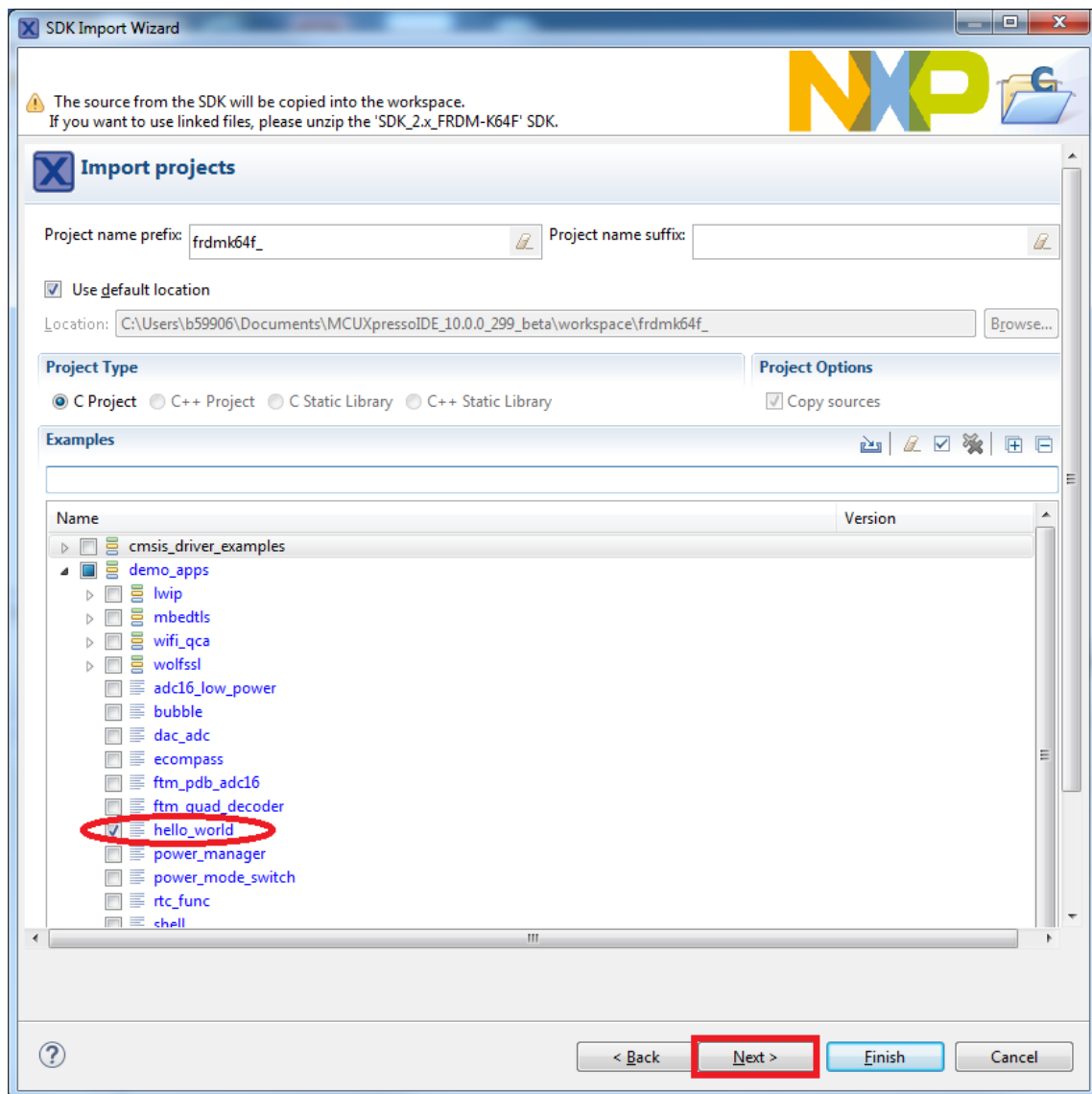
1. Drag and drop the SDK zip file into the **Installed SDKs** view to install an SDK. In the window that appears, click **OK** and wait until the import has finished.



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



3. Expand the demo_apps folder and select hello_world.
4. Click **Next**.



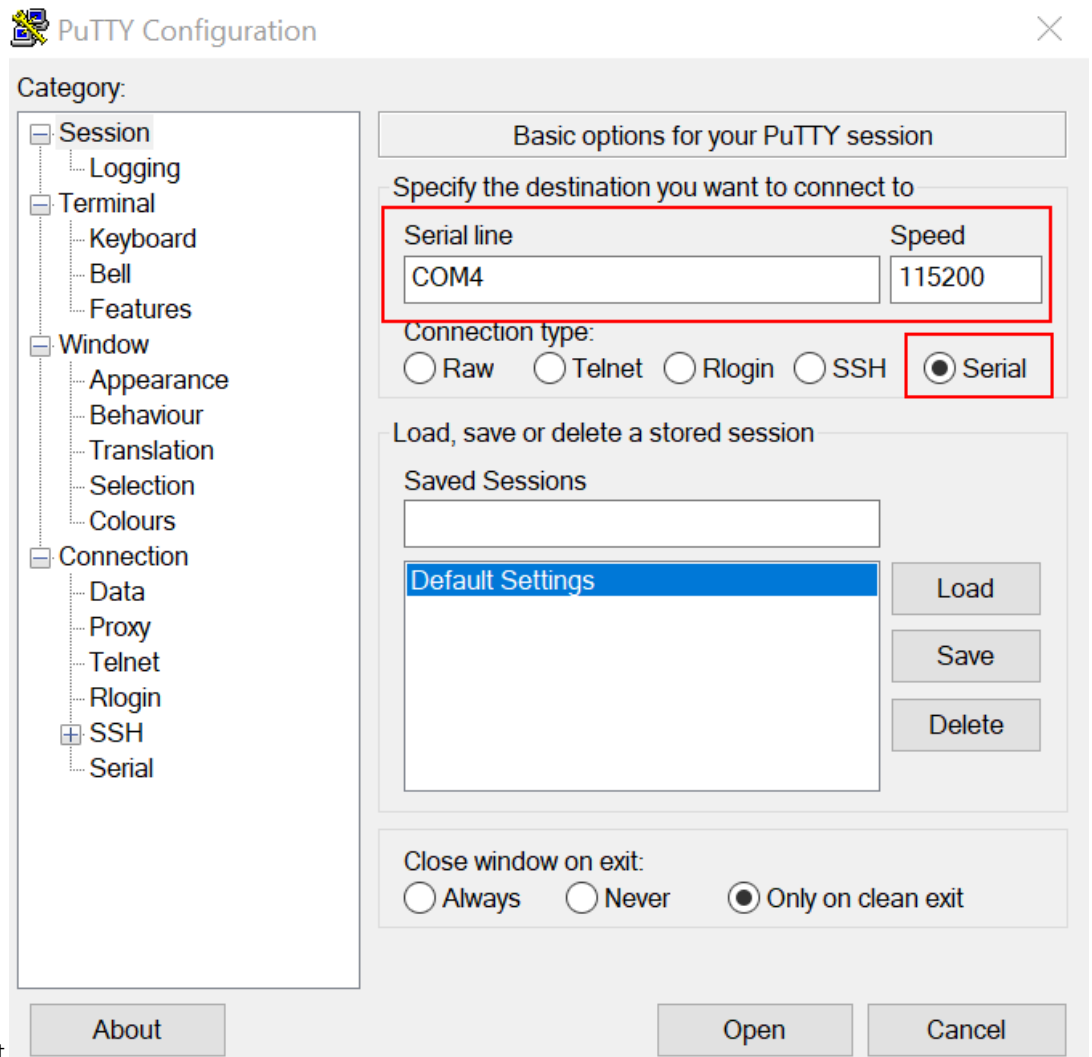
5. Ensure **Redlib: Use floating-point version of printf** is selected if the example prints floating-point numbers on the terminal for demo applications such as `adc_basic`, `adc_burst`, `adc_dma`, and `adc_interrupt`. Otherwise, it is not necessary to select this option. Then, click **Finish**.

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

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

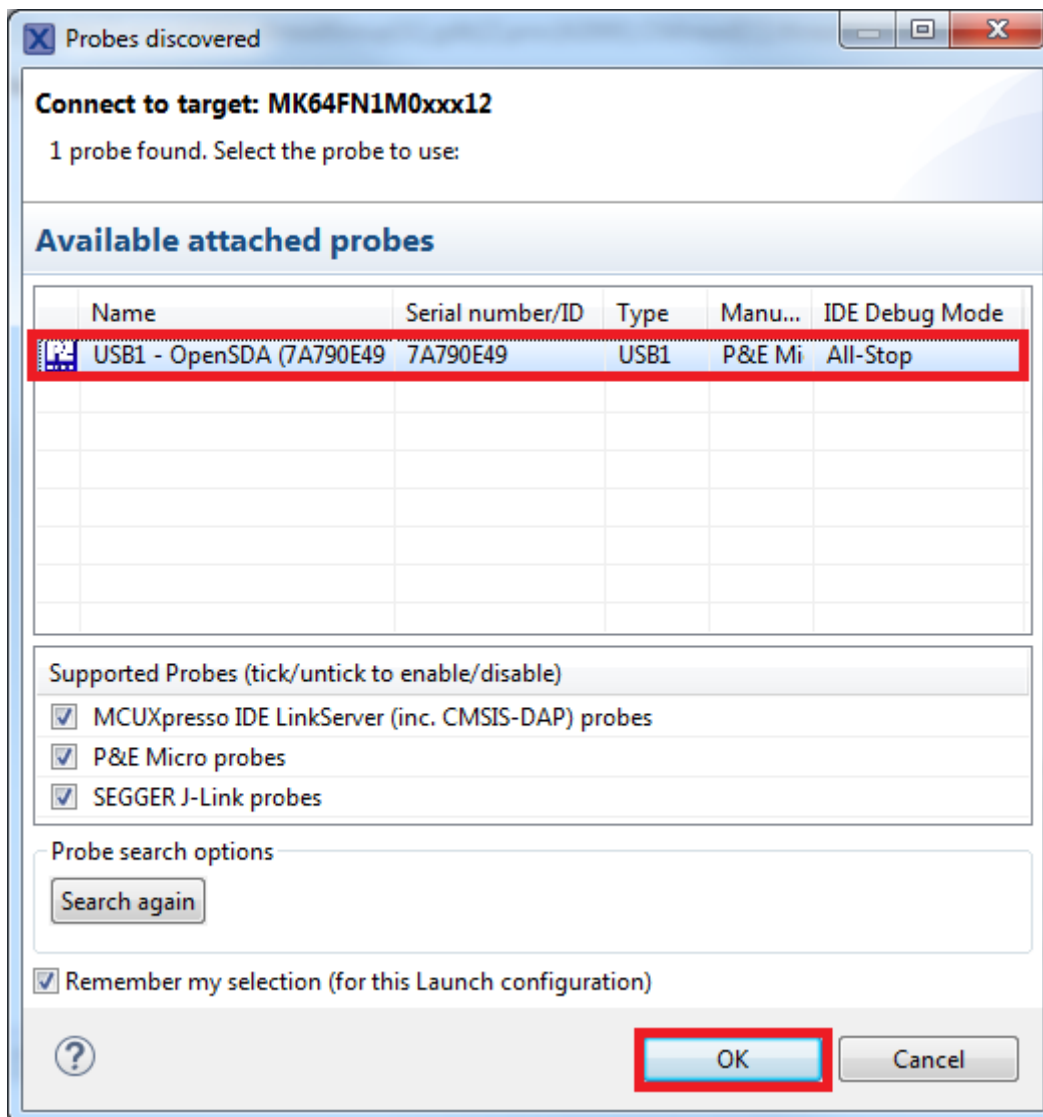
1. Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
2. Connect the development platform to your PC via a USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug serial port number (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference `BOARD_DEBUG_UART_BAUDRATE` variable in `board.h` file)
 2. No parity

3. 8 data bits



4. 1 stop bit

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



- The application is downloaded to the target and automatically runs to `main()`.
- Start the application by clicking **Resume**.

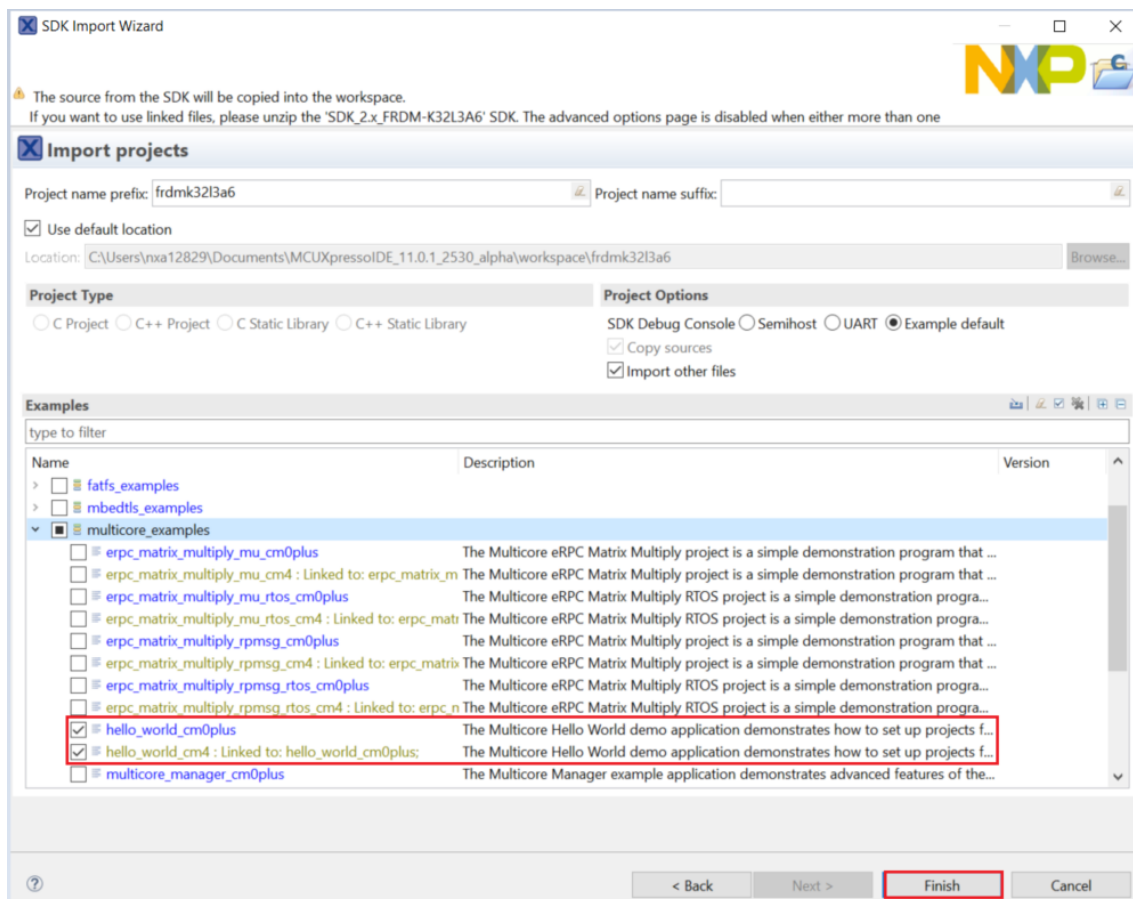


The `hello_world` application is now running and a banner is displayed on the terminal. If not, check your terminal settings and connections.

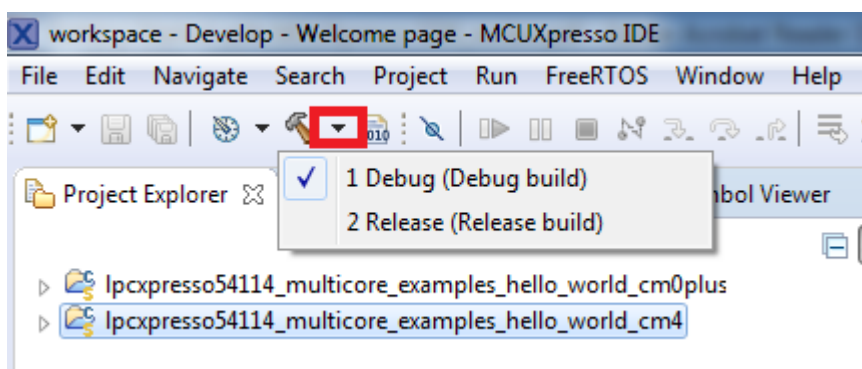


Build a multicore example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug multicore example applications. The following steps can be applied to any multicore example application in the MCUXpresso SDK. Here, the dual-core version of hello_world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

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

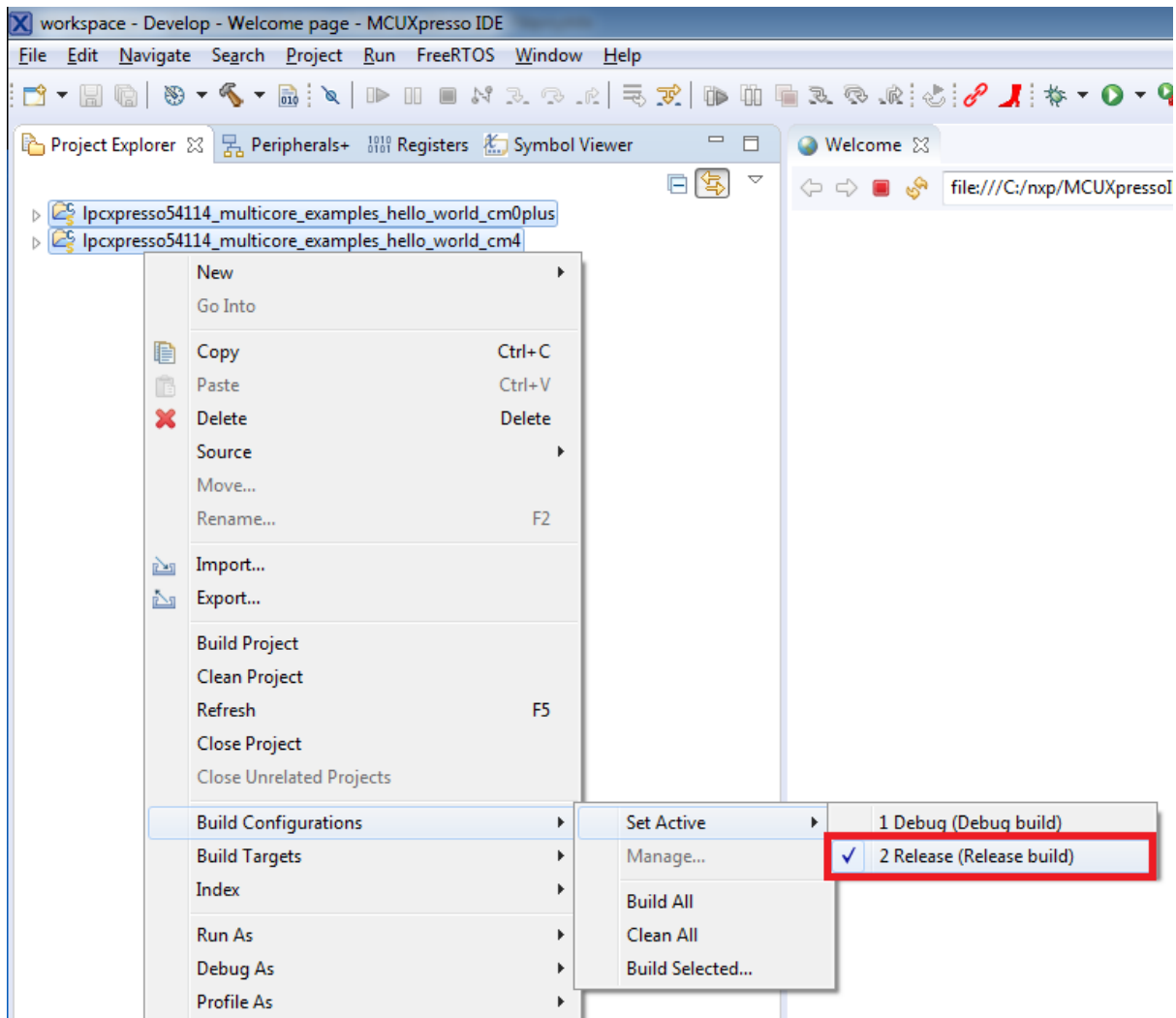


3. Now, two projects should be imported into the workspace. To start building the multicore application, highlight the `lpcxpresso54114_multicore_examples_hello_world_cm4` project (multicore master project) in the Project Explorer. Then choose the appropriate build target, **Debug** or **Release**, by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

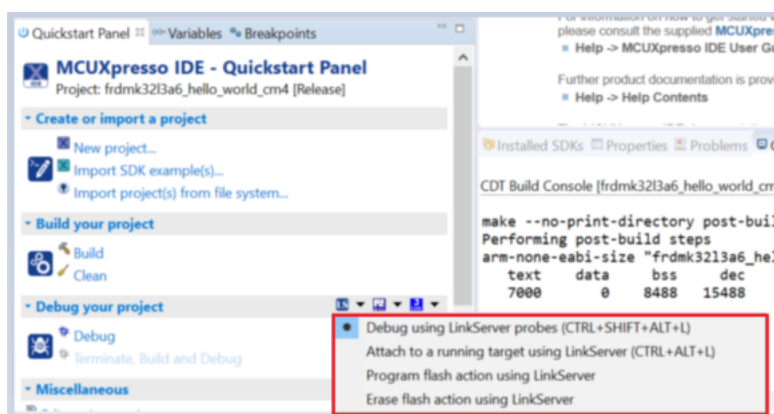


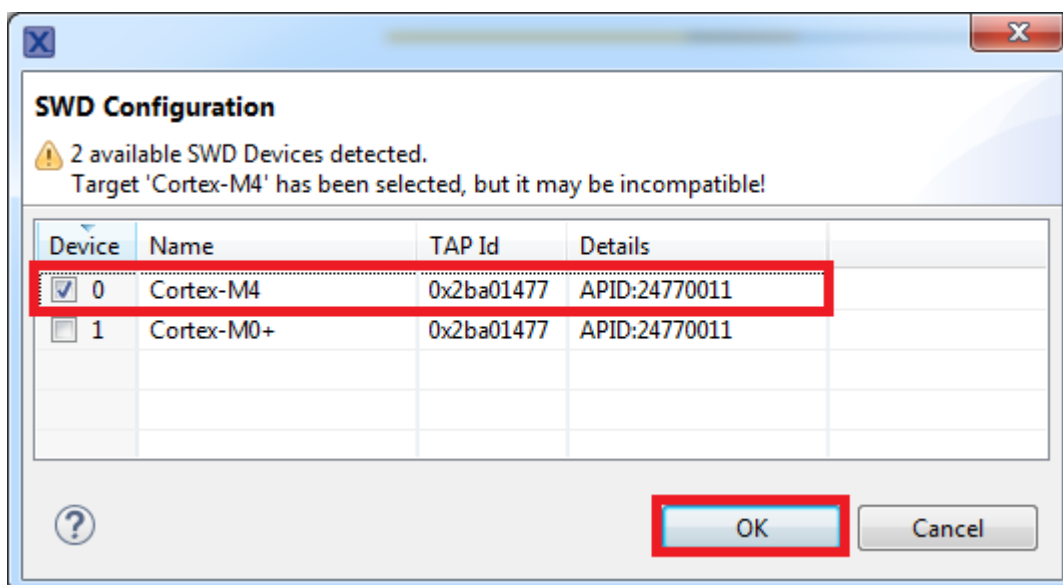
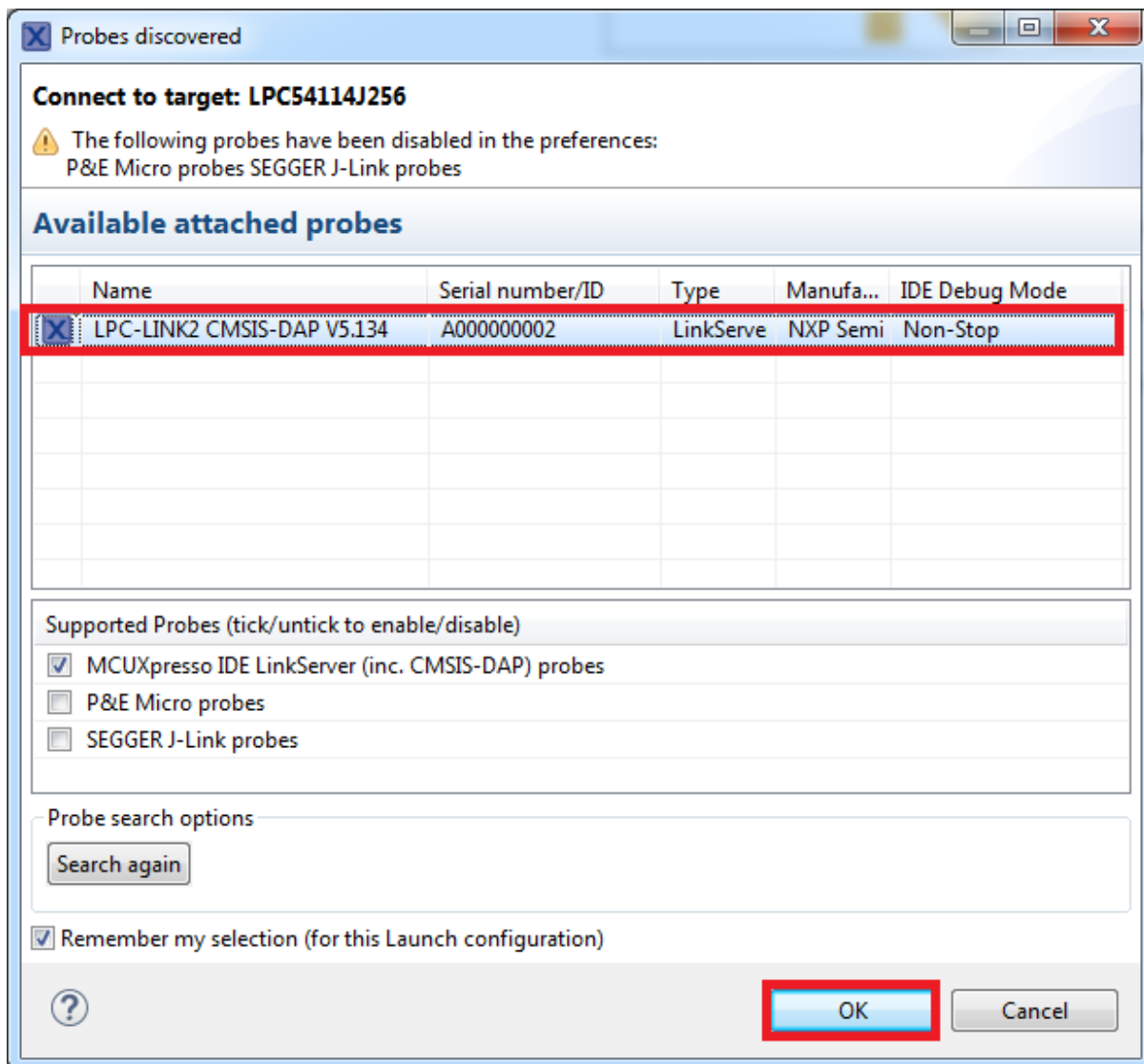
The project starts building after the build target is selected. Because of the project reference settings in multicore projects, triggering the build of the primary core application (cm4) also causes the referenced auxiliary core application (cm0plus) to build.

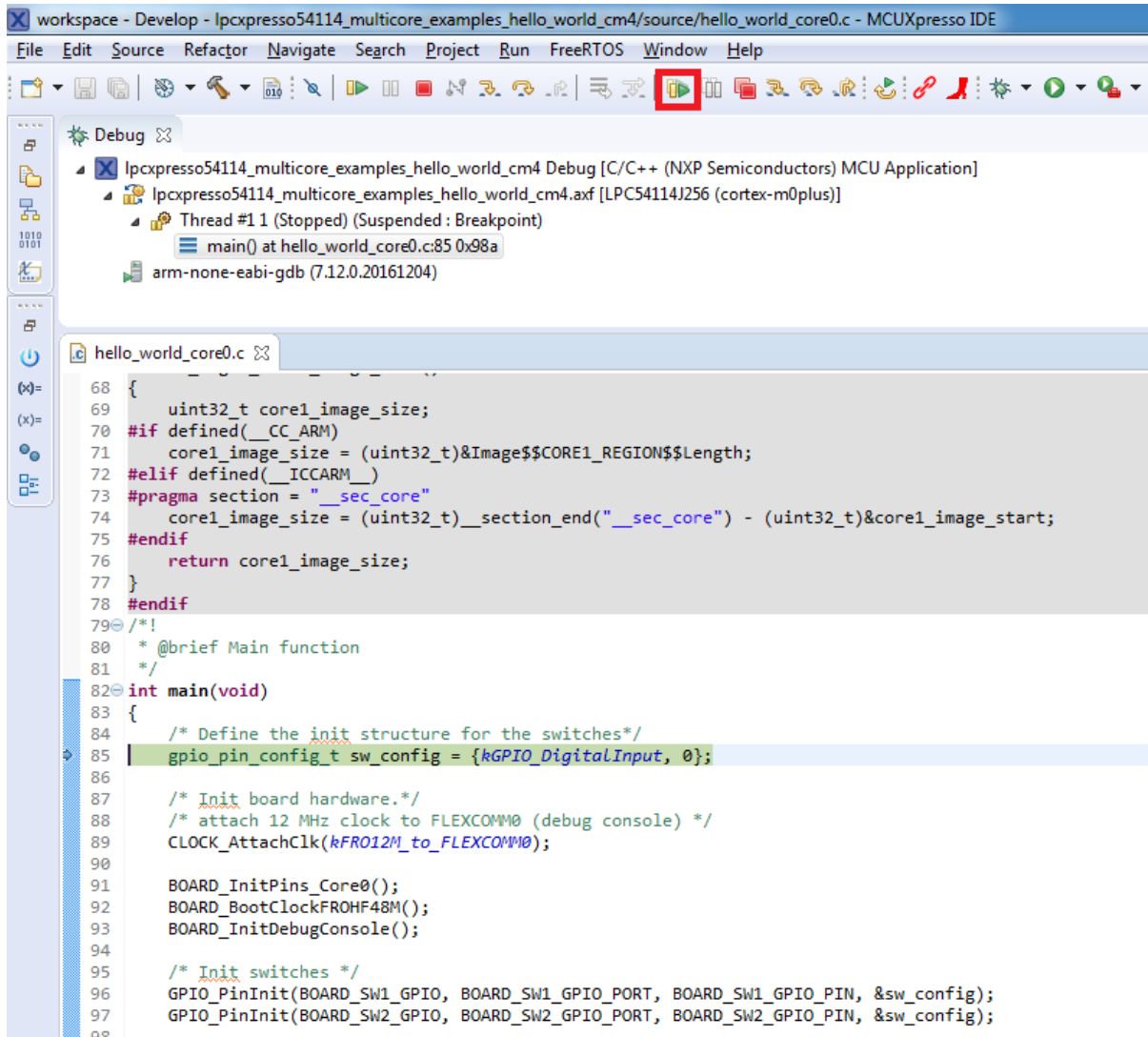
Note: When the **Release** build is requested, it is necessary to change the build configuration of both the primary and auxiliary core application projects first. To do this, select both projects in the Project Explorer view and then right click which displays the context-sensitive menu. Select **Build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



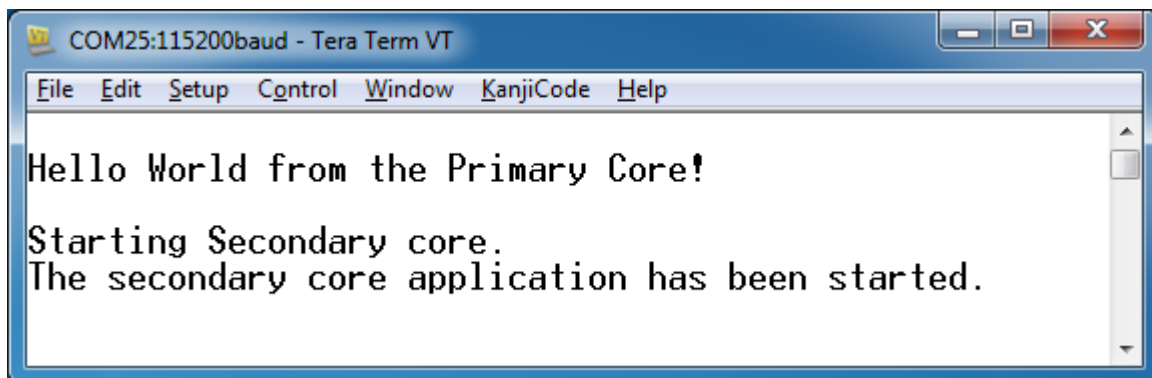
Run a multicore example application The primary core debugger handles flashing of both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform all steps as described in **Run an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





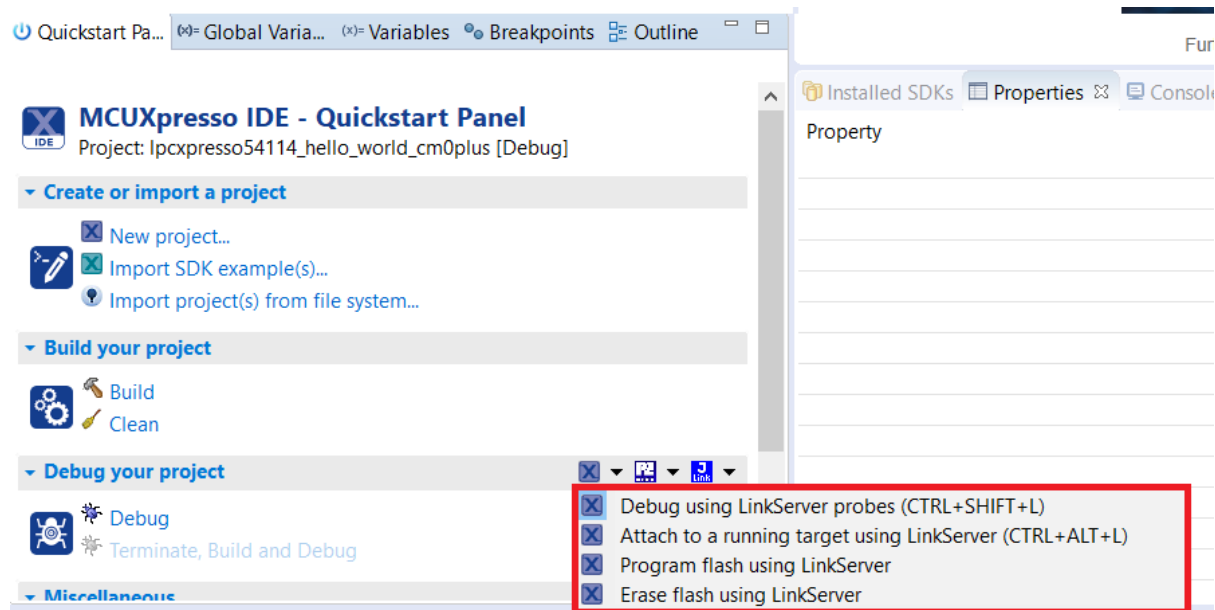


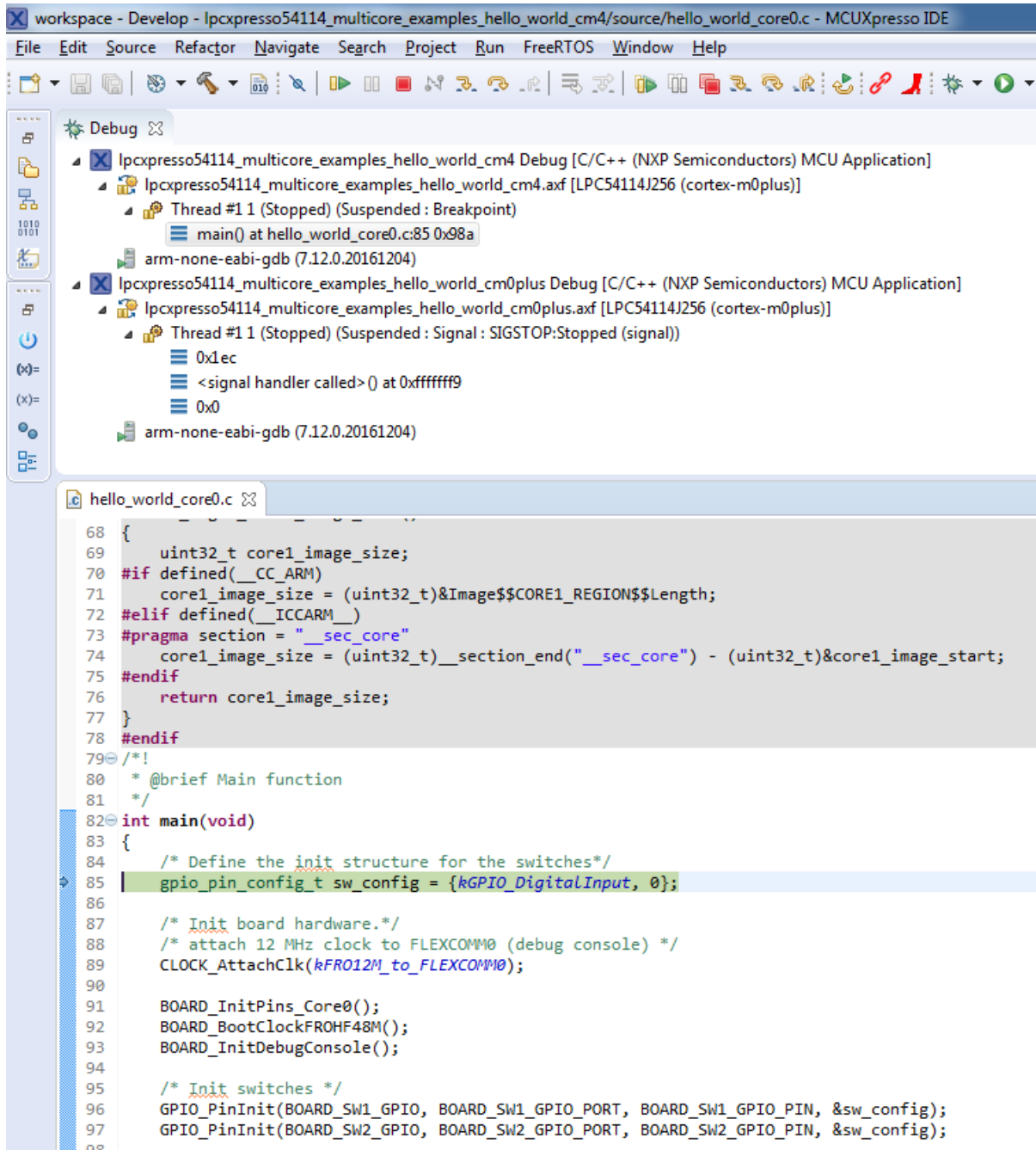
After clicking the “Resume All Debug sessions” button, the `hello_world` multicore application runs and a banner is displayed on the terminal. If this is not the case, check your terminal settings and connections.



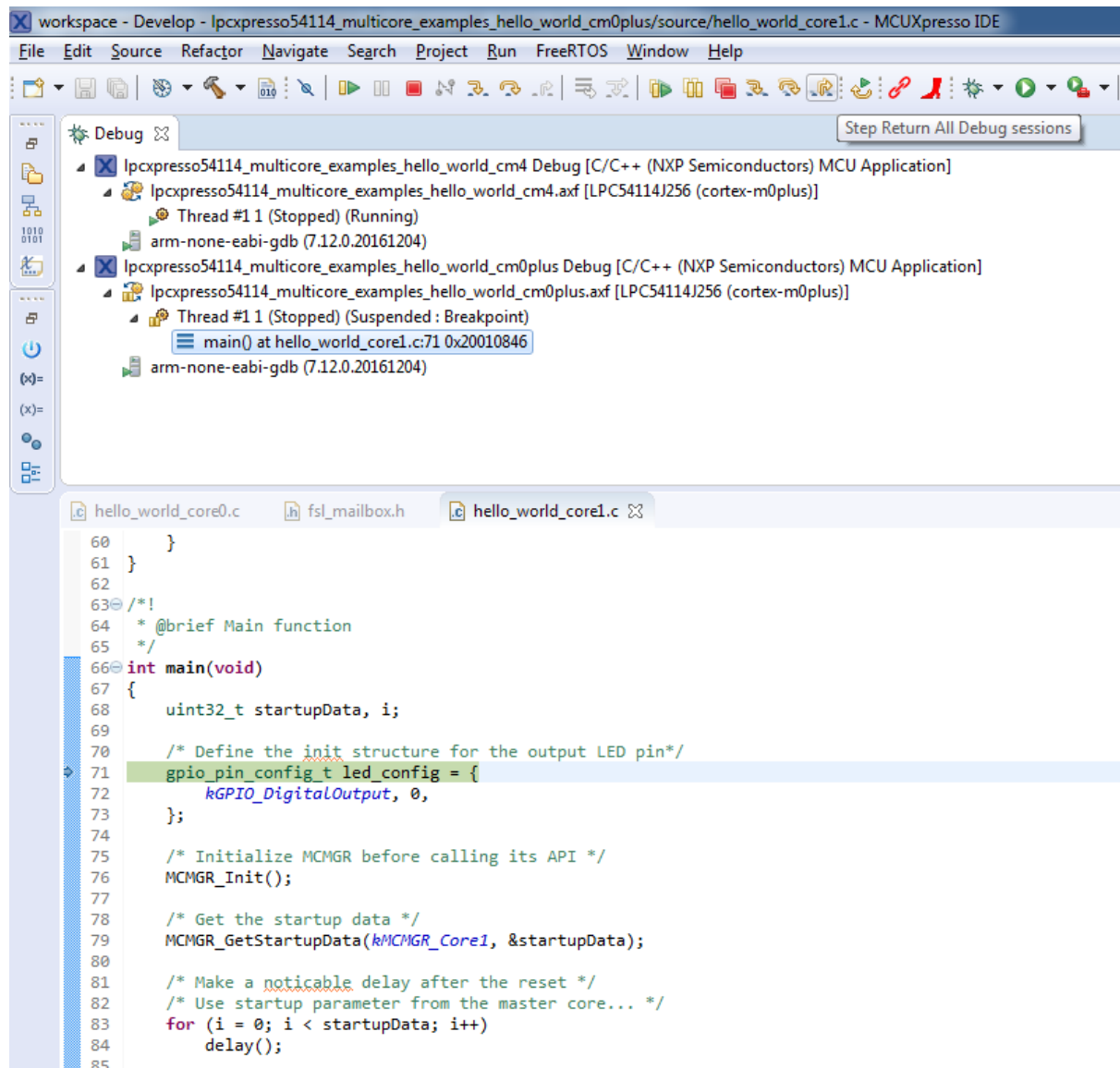
An LED controlled by the auxiliary core starts flashing, indicating that the auxiliary core has been released from the reset and running correctly. It is also possible to debug both sides of the multicore application in parallel. After creating the debug session for the primary core, perform same steps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

session.

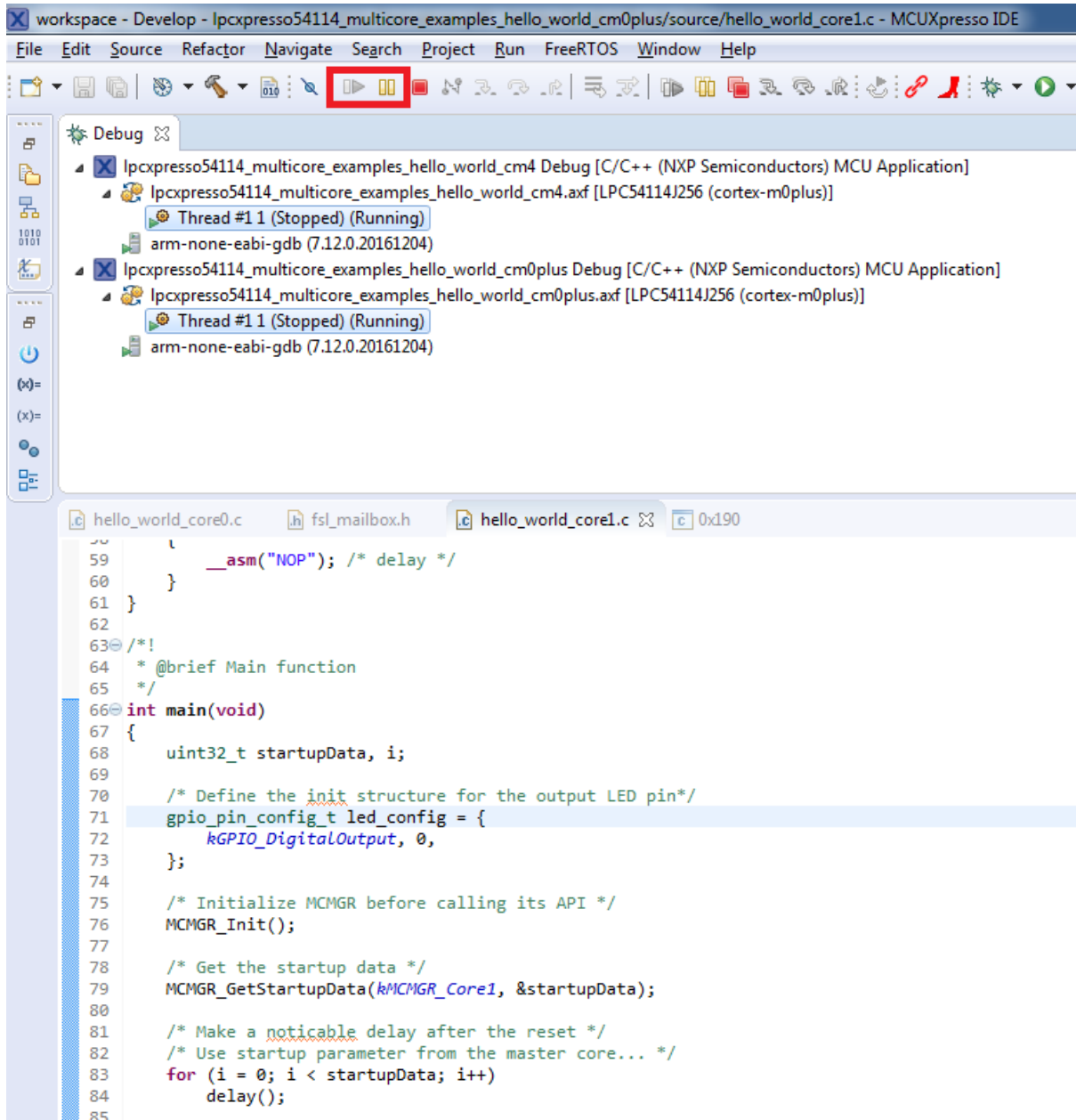


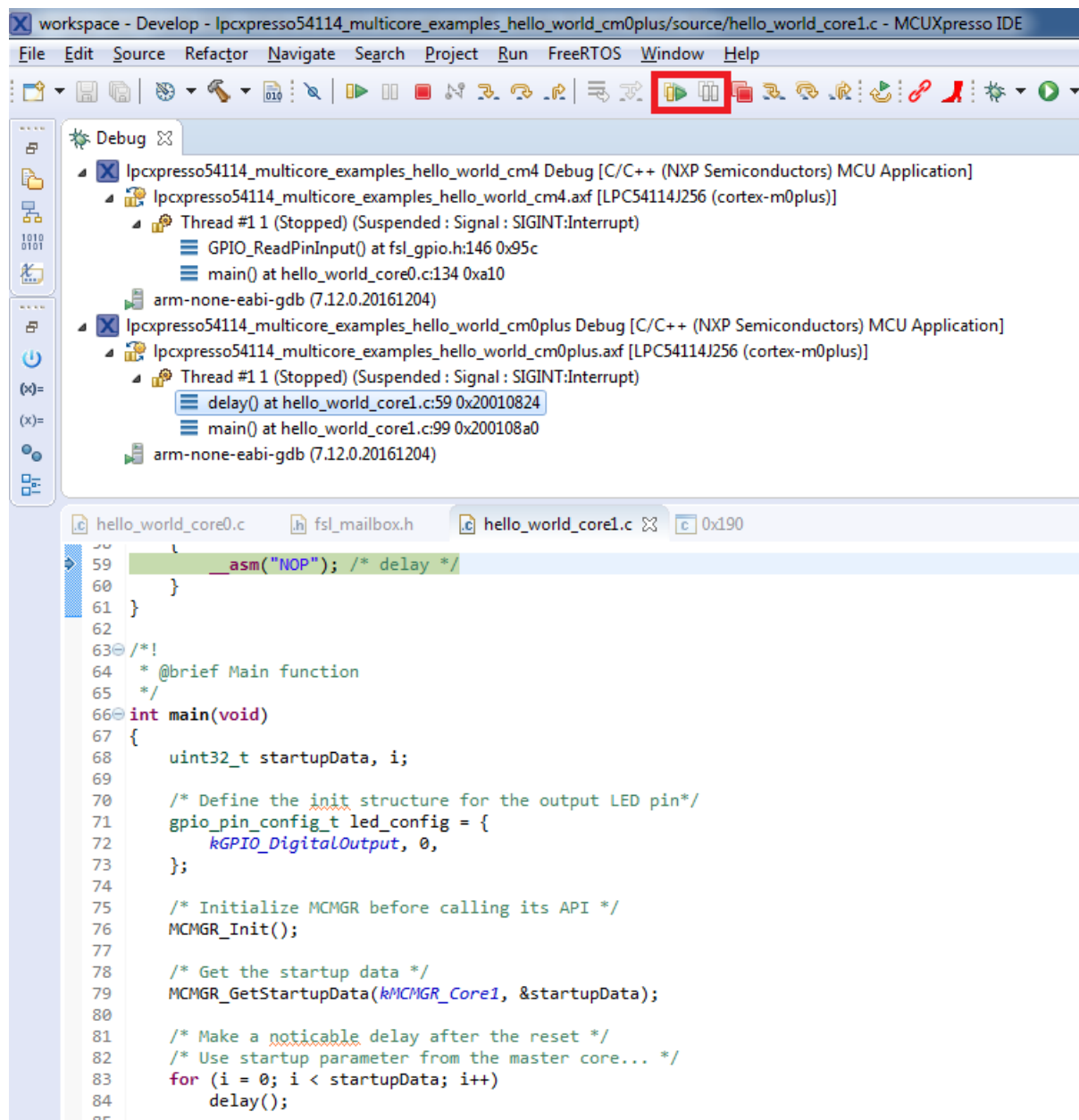


Now, the two debug sessions should be opened, and the debug controls can be used for both debug sessions depending on the debug session selection. Keep the primary core debug session selected by clicking the “Resume” button. The hello_world multicore application then starts running. The primary core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



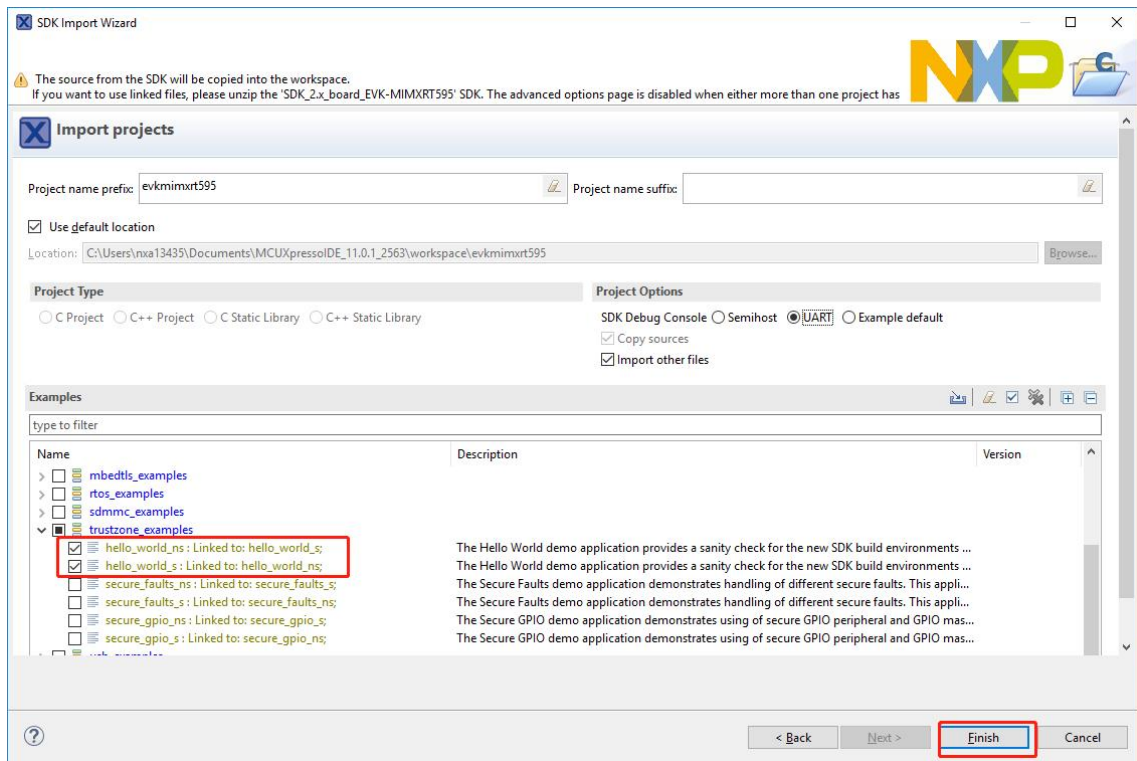
At this point, it is possible to suspend and resume individual cores independently. It is also possible to make synchronous suspension and resumption of both the cores. This is done either by selecting both opened debug sessions (multiple selections) and clicking the “Suspend” / “Resume” control button,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



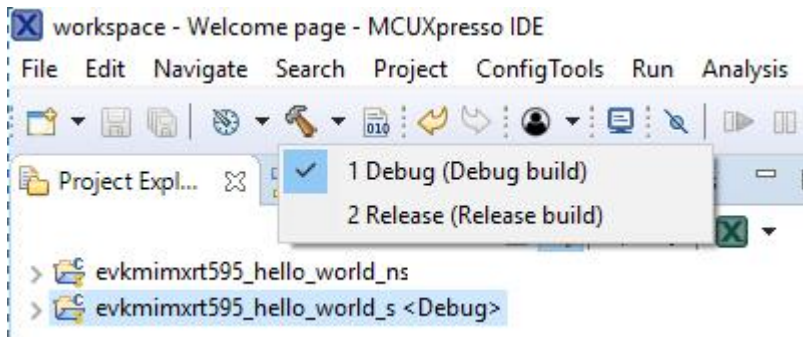


Build a TrustZone example application This section describes the steps required to configure MCUXpresso IDE to build, run, and debug TrustZone example applications. The TrustZone version of the `hello_world` example application targeted for the MIMXRT595-EVK hardware platform is used as an example, though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

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

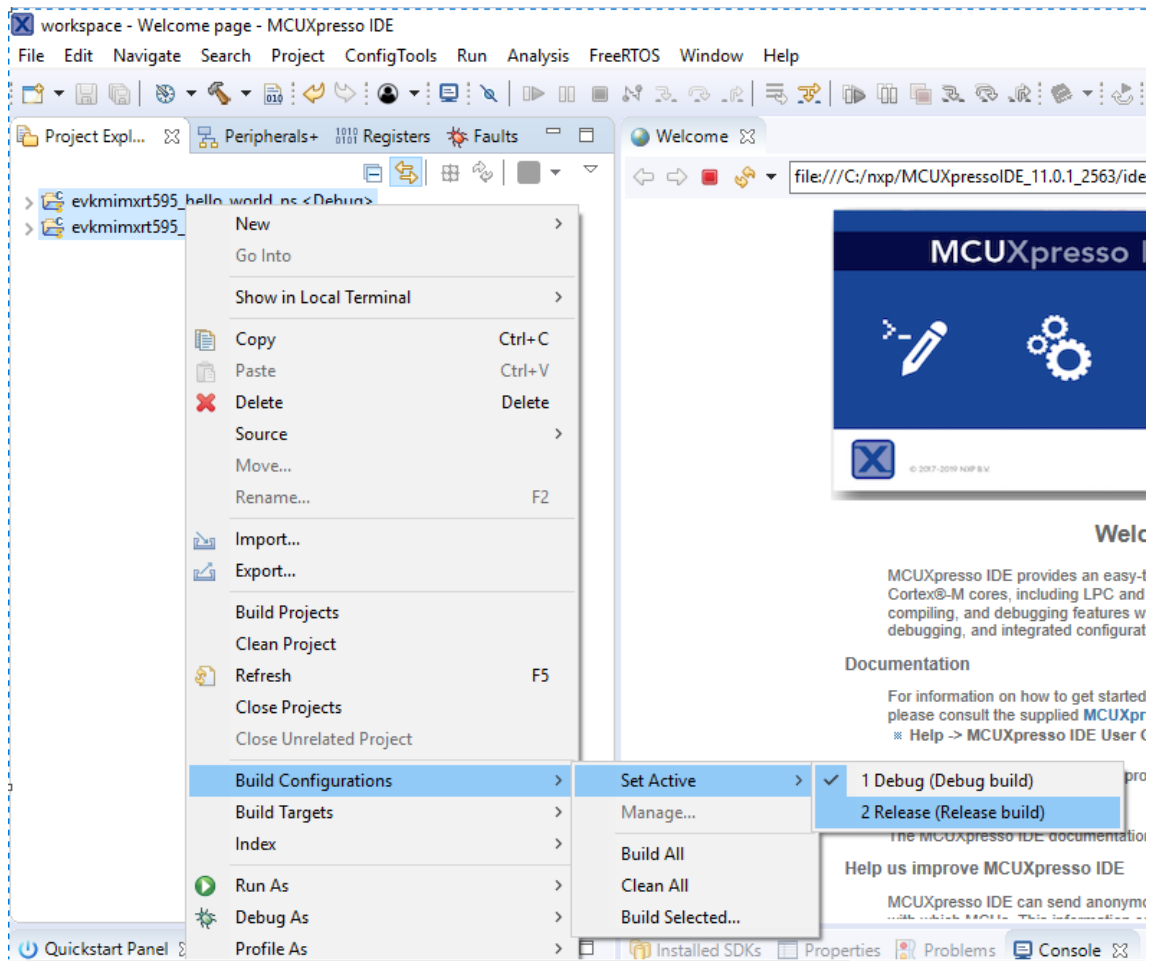


3. Now, two projects should be imported into the workspace. To start building the TrustZone application, highlight the `evkmimxrt595_hello_world_s` project (TrustZone master project) in the Project Explorer. Then, choose the appropriate build target, **Debug** or **Release**, by clicking the downward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



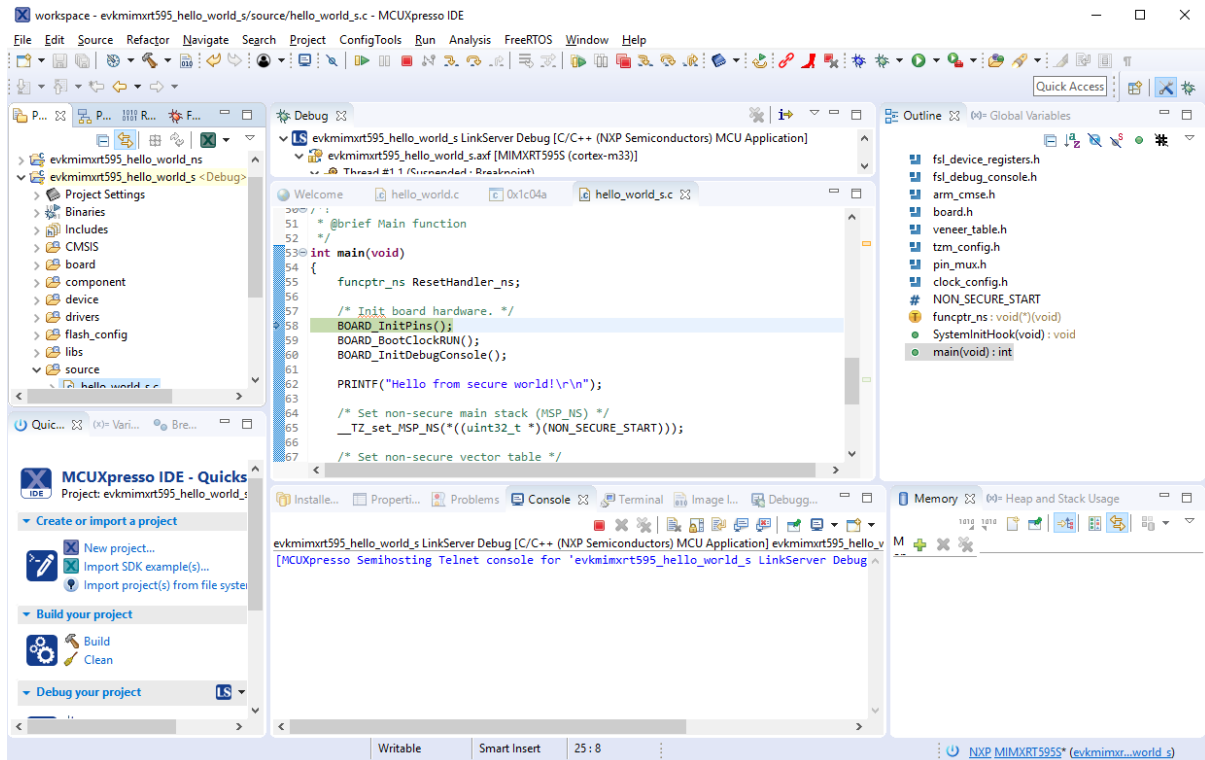
The project starts building after the build target is selected. It is requested to build the application for the secure project first, because the non-secure project must know the secure project since CMSE library when running the linker. It is not possible to finish the non-secure project linker when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

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



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

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



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

Run a demo application using IAR This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

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

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

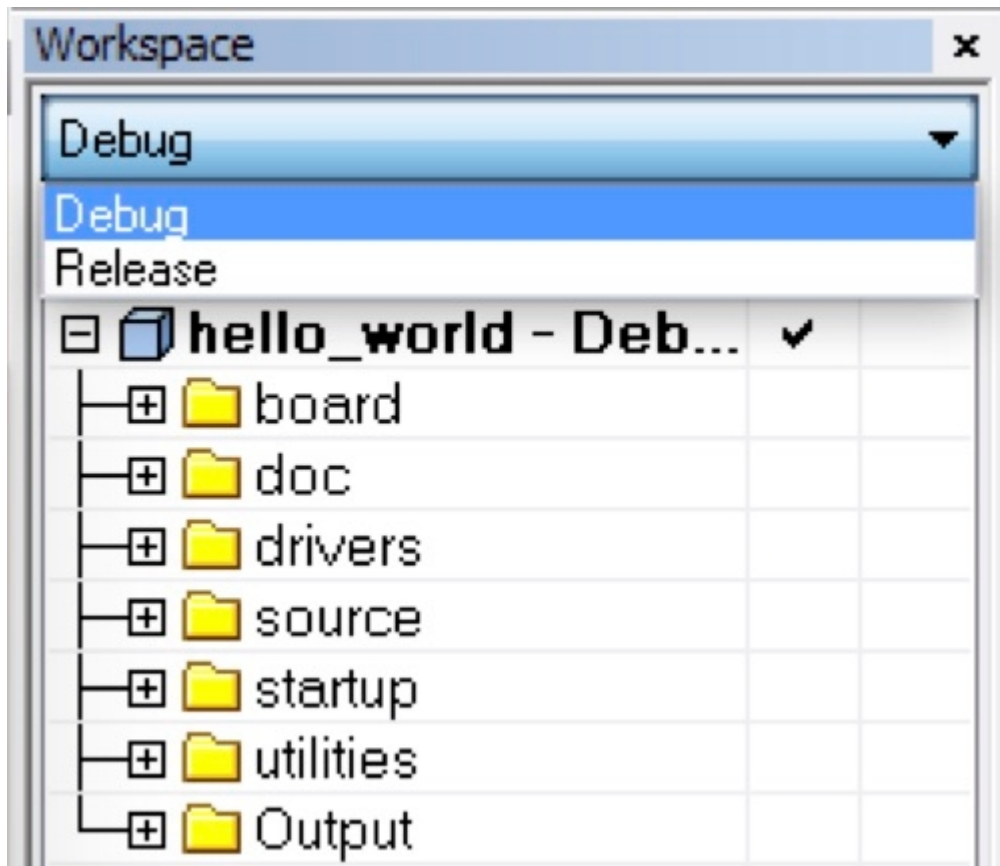
1. Open the desired demo application workspace. Most example application workspace files can be located using the following path:

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

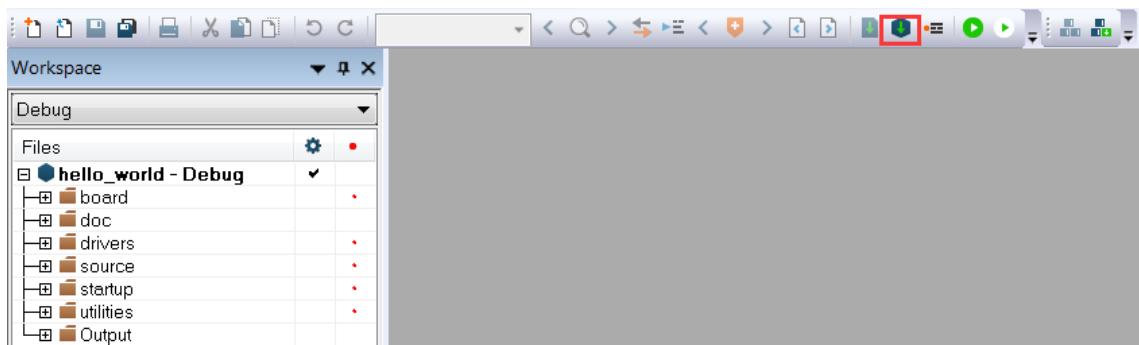
Other example applications may have additional folders in their path.

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

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



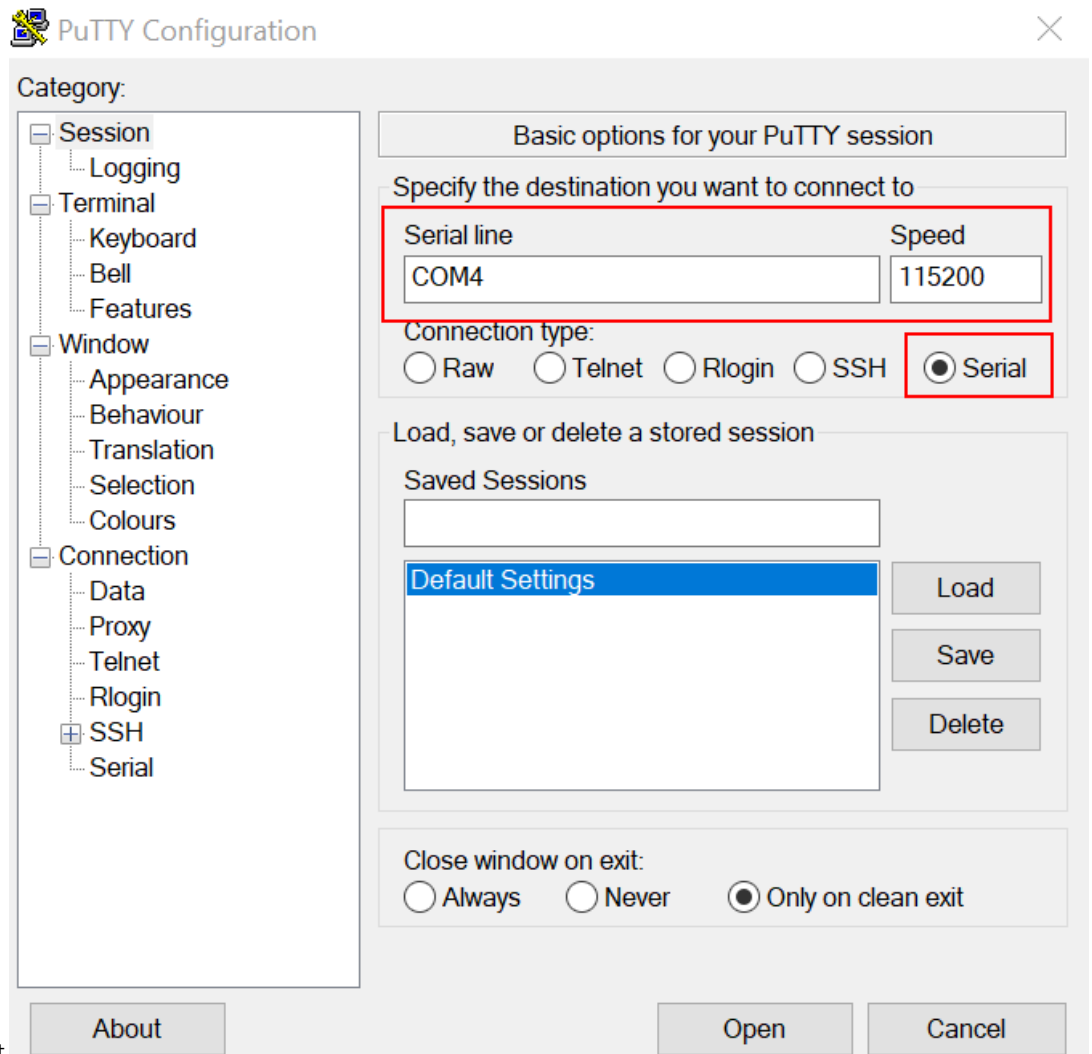
3. To build the demo application, click **Make**, highlighted in red in following figure.



4. The build completes without errors.

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

1. Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
2. Connect the development platform to your PC via USB cable.
3. Open the terminal application on the PC, such as PuTTY or TeraTerm, and connect to the debug COM port (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 1. 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 2. No parity
 3. 8 data bits

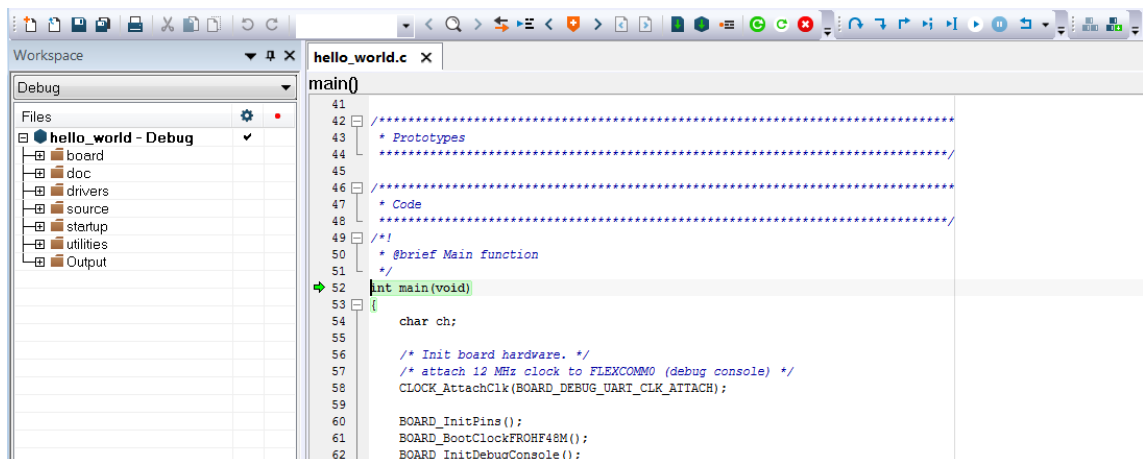


4. 1 stop bit

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



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



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



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



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

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

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

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

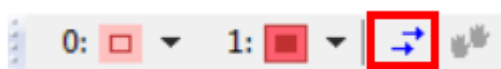
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm4/iar/hello_world_cm4.eww
```

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

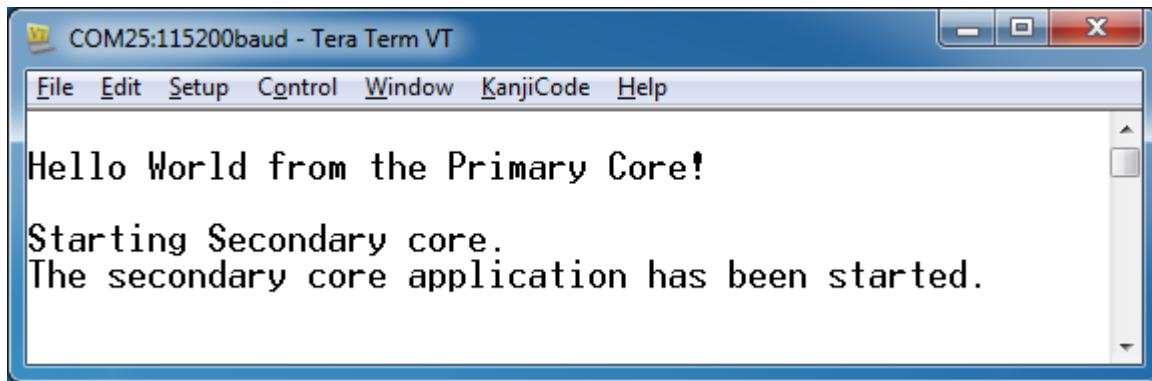
Run a multicore example application The primary core debugger handles flashing both primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 4 as described in **Run an example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

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

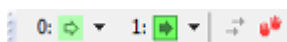
Run both cores by clicking the “Start all cores” button to start the multicore application.



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



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



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

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

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

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

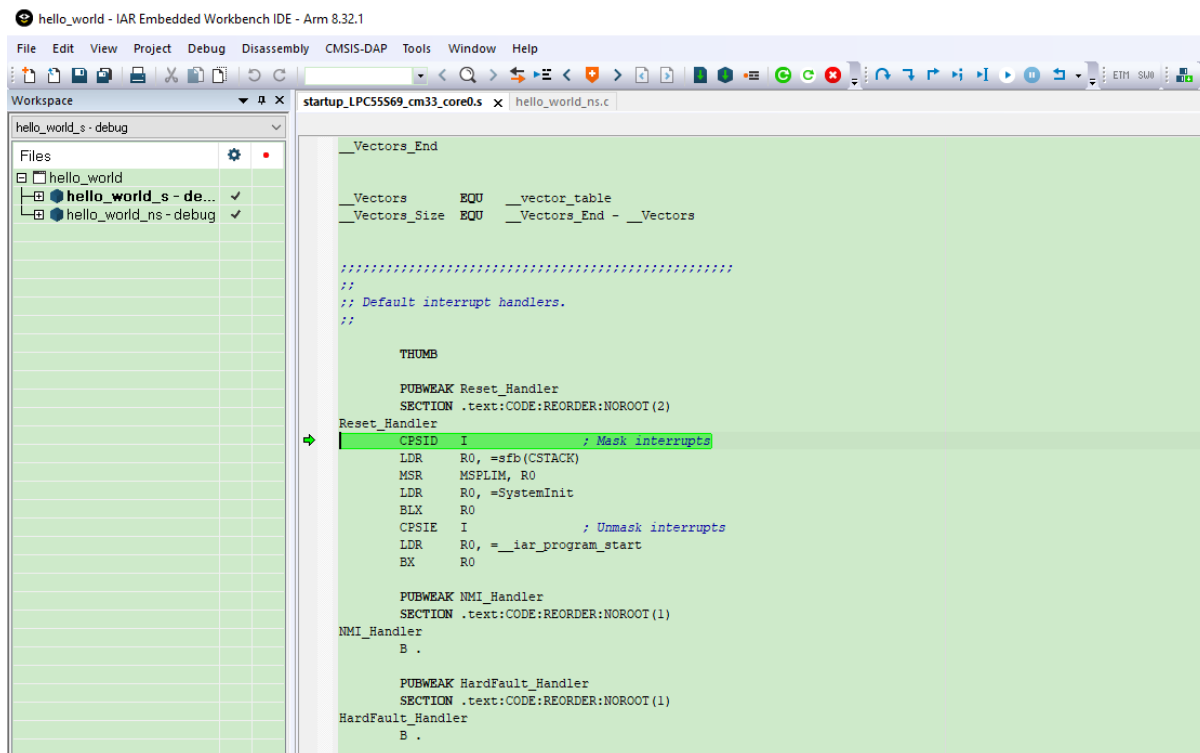
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_ns/iar/hello_world_  
↔ns.eww
```

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

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

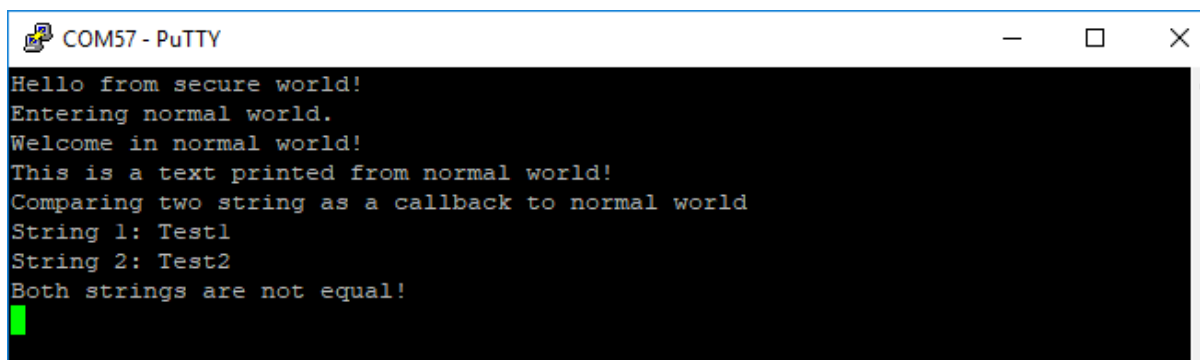
This project `hello_world.eww` contains both secure and non-secure projects in one workspace and it allows the user to easily transition from one project to another. Build both applications separately by clicking **Make**. It is requested to build the application for the secure project first, because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

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

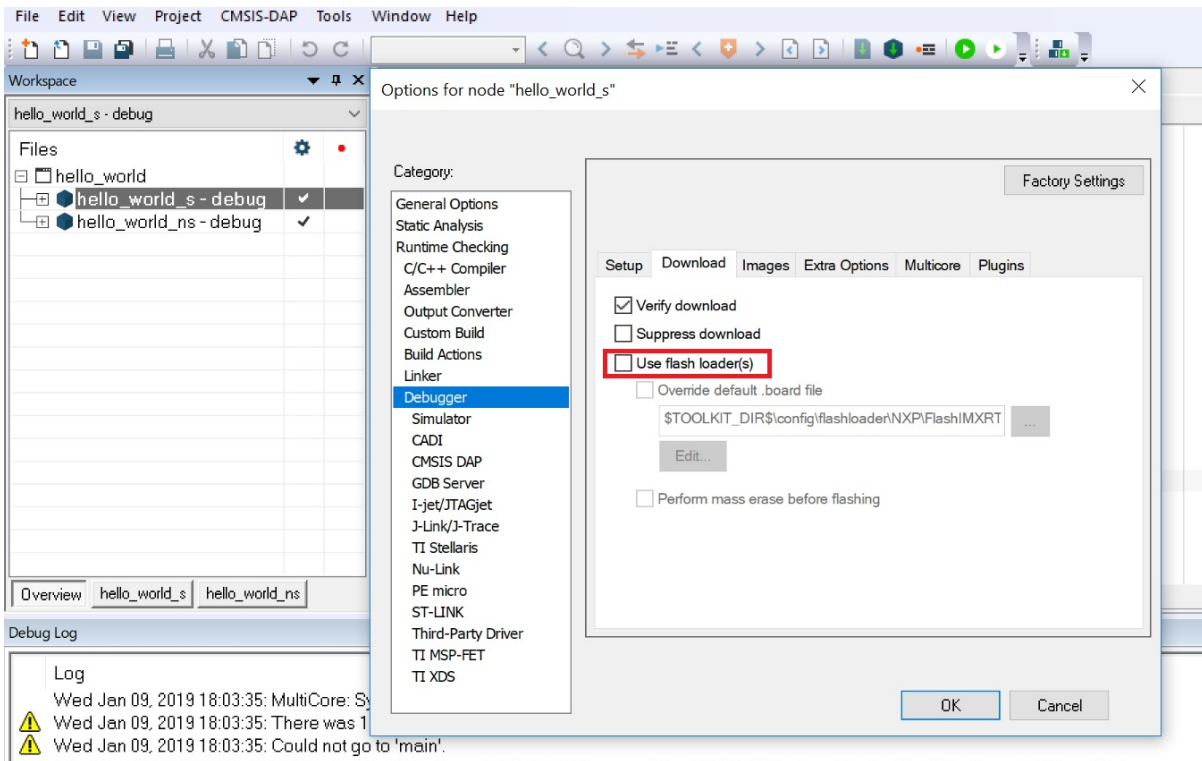


Run the code by clicking **Go** to start the application.

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



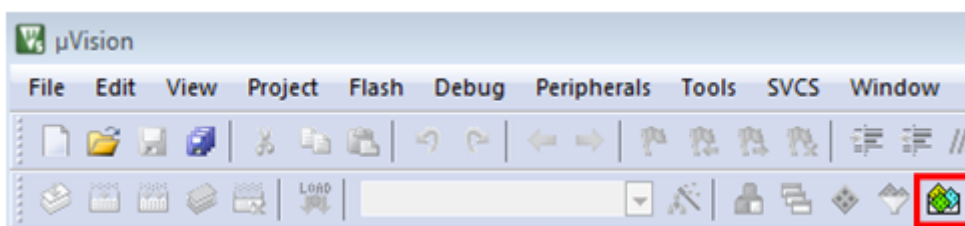
Note: If the application is running in RAM (debug/release build target), in **Options**>**Debugger > Download** tab, disable **Use flash loader(s)**. This can avoid the `_ns` download issue on i.MXRT500.



Run a demo using Keil MDK/µVision This section describes the steps required to build, run, and debug example applications provided in the MCUXpresso SDK.

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

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



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

Build an example application

1. Open the desired example application workspace in:

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

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

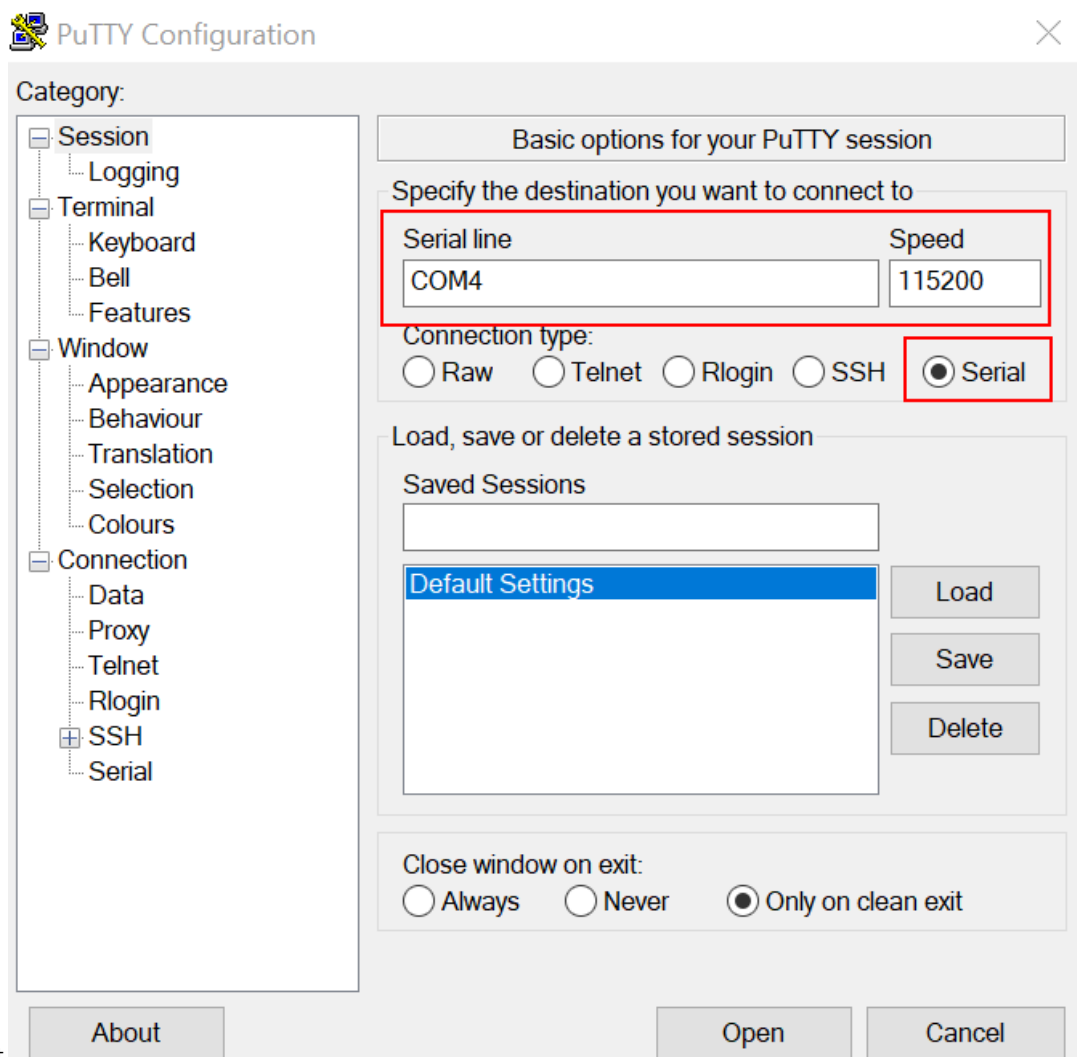
- To build the demo project, select **Rebuild**, highlighted in red.



- The build completes without errors.

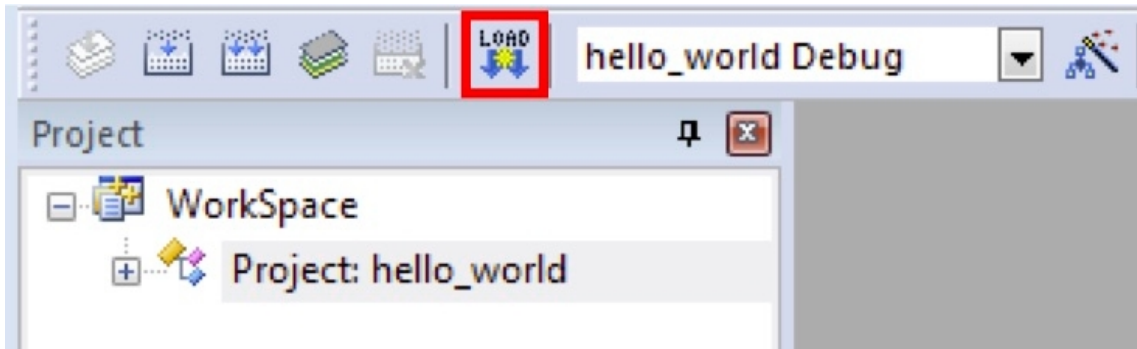
Run an example application To download and run the application, perform these steps:

- Ensure the host driver for the debugger firmware has been installed. See [On-board debugger](#).
- Connect the development platform to your PC via USB cable using USB connector.
- Open the terminal application on the PC, such as PuTTY or TeraTerm and connect to the debug serial port number (to determine the COM port number, see [How to determine COM port](#)). Configure the terminal with these settings:
 - 115200 or 9600 baud rate, depending on your board (reference BOARD_DEBUG_UART_BAUDRATE variable in the board.h file)
 - No parity
 - 8 data bits

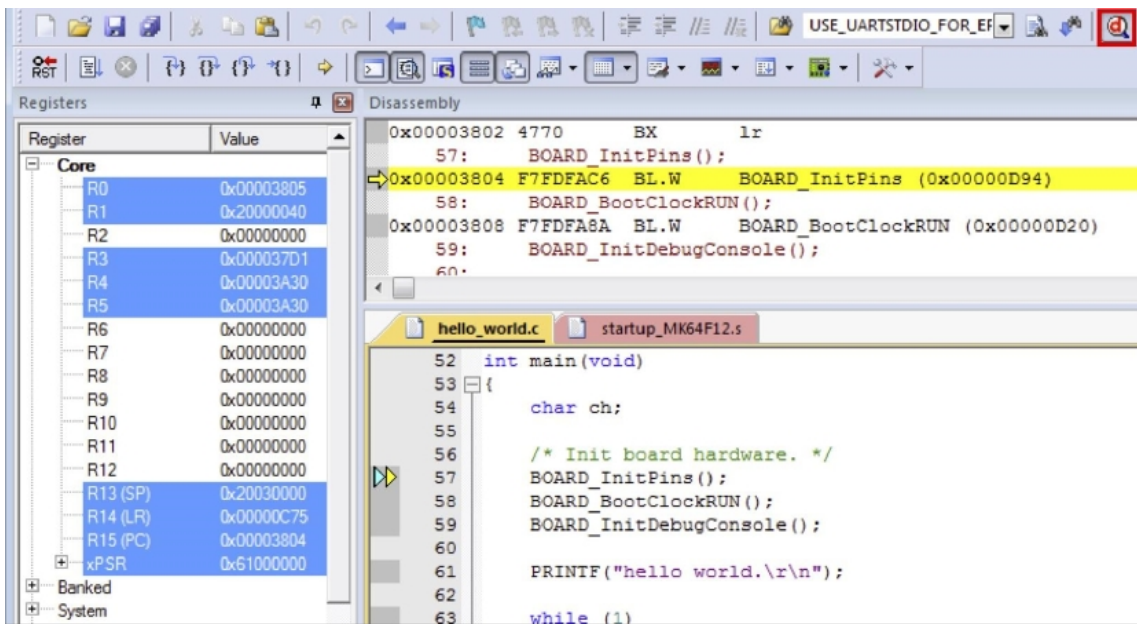


- 1 stop bit

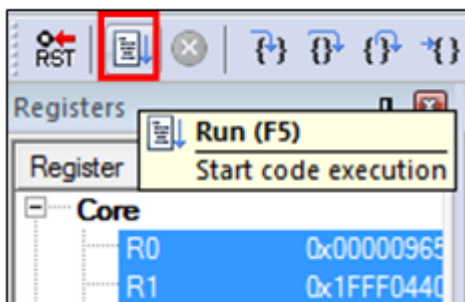
- In μ Vision, after the application is built, click the **Download** button to download the application to the target.



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



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



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



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

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

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

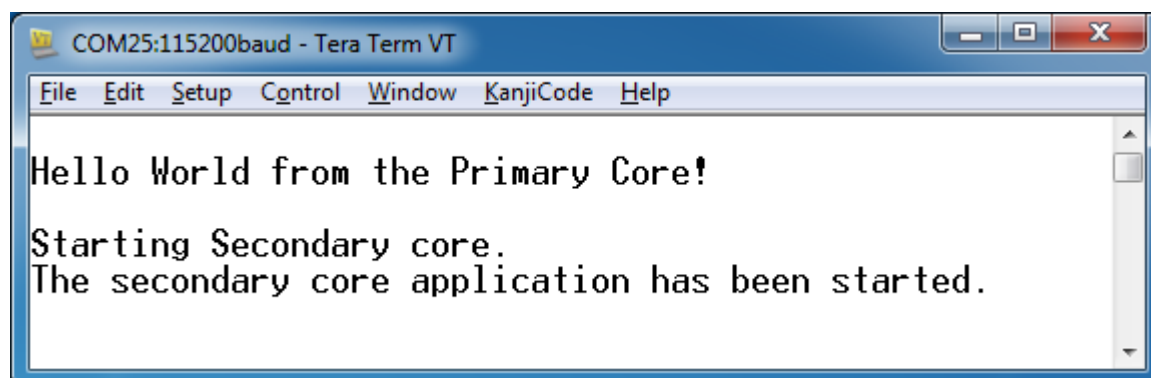
```
<install_dir>/boards/lpcxpresso54114/multicore_examples/hello_world/cm0plus/mdk/hello_world_
↪cm0plus.uvmpw
```

```
<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.

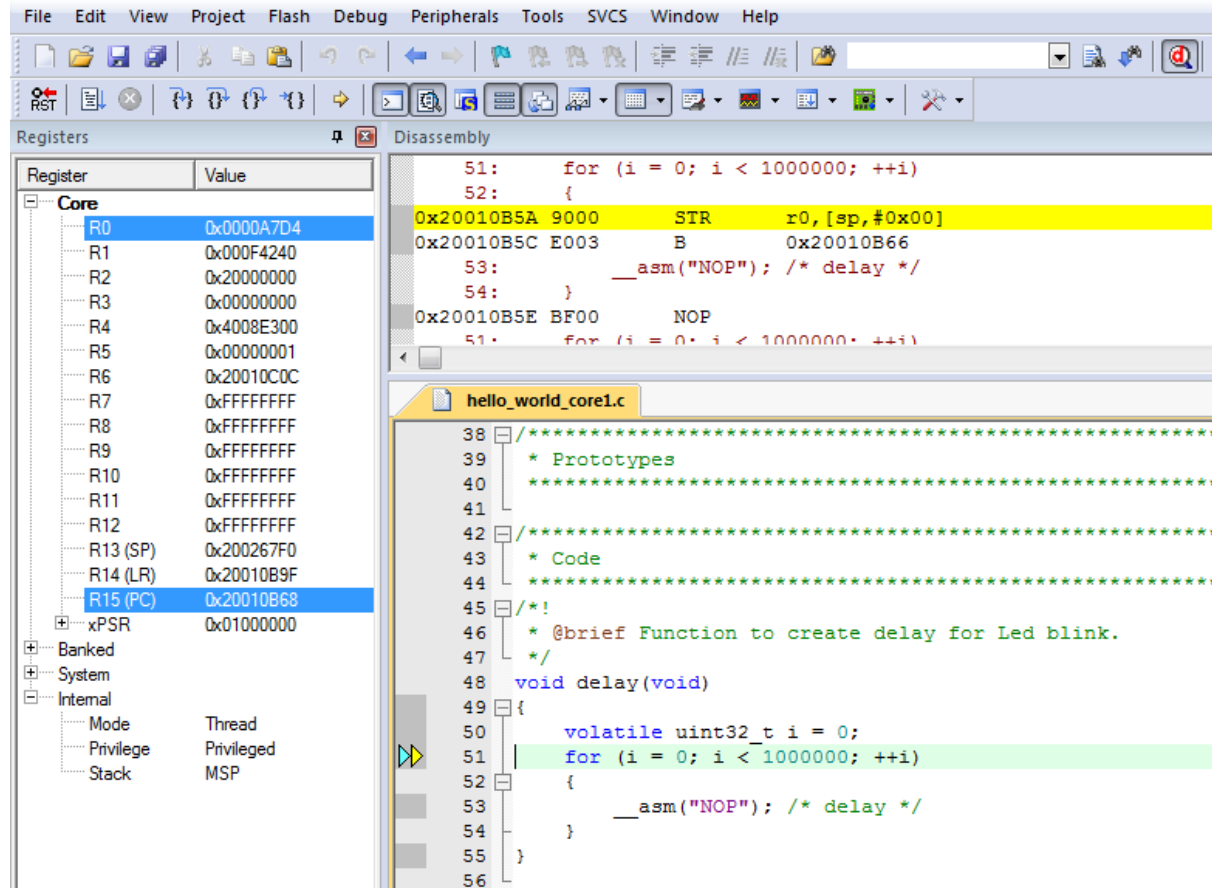
Run a multicore example application The primary core debugger flashes both the primary and the auxiliary core applications into the SoC flash memory. To download and run the multicore application, switch to the primary core application project and perform steps 1 – 5 as described in **Run an example application**. These steps are common for both single-core and dual-core applications in μ 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 μ Vision instance and clicking the “Start/Stop Debug Session” button. After this, the second debug session is opened and the auxiliary core application can be debugged.



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

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

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

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

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

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

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

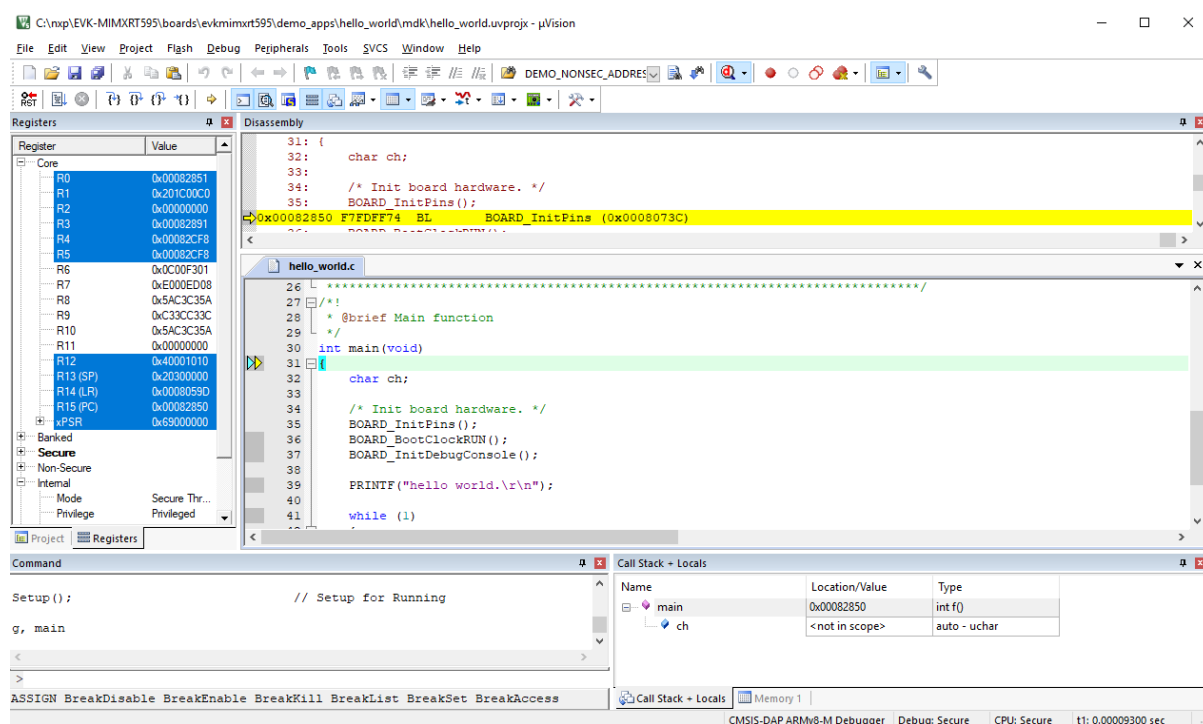
```
<install_dir>/boards/<board_name>/trustzone_examples/hello_world/hello_world_s/mdk/hello_world.  
↪ uvmpw
```

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

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

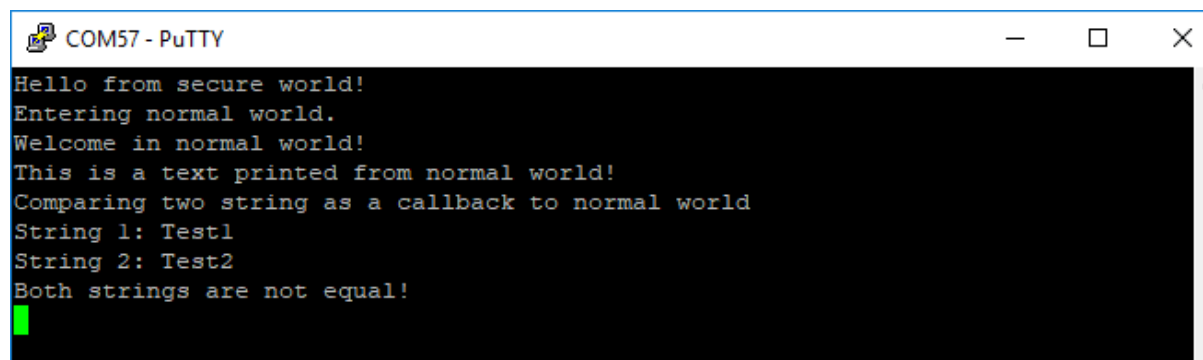
Run a TrustZone example application The secure project is configured to download both secure and non-secure output files so debugging can be fully managed from the secure project.

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



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

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








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

MCUXpresso Config Tools can be accessed in the following products:

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

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

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

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

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

There are two ports, one is for core0 debug console and the other is for core1.

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

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

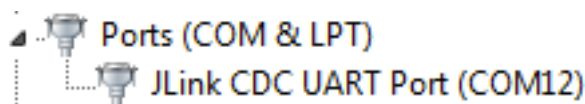
1. **CMSIS-DAP/mbed/DAPLink** interface:



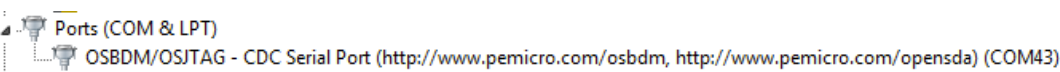
2. **P&E Micro:**



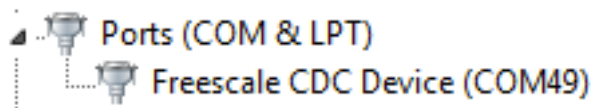
3. **J-Link:**



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 and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.

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

Updating MCU-Link firmware This firmware in this debug interface may be updated using the host computer utility called MCU-Link. This typically used when switching between the default debugger protocol (CMSIS-DAP) to SEGGER J-Link, or for updating this firmware with new releases of these. This section contains the steps to reprogram the debug probe firmware.

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

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

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

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

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

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

The corresponding host driver must be installed before debugging.

- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
- If using J-Link with either a standalone debug pod or MCU-Link, install the J-Link software (drivers and utilities) from www.segger.com/jlink-software.html.

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

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

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

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

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

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

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

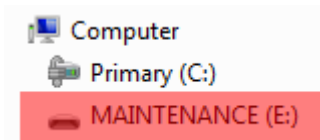
- For boards with CMSIS-DAP firmware, visit developer.mbed.org/handbook/Windows-serial-configuration and follow the instructions to install the Windows operating system serial driver. If running on Linux OS, this step is not required.
- For boards with a P&E Micro interface, see [PE micro](#) to download and install the P&E Micro Hardware Interface Drivers package.

Updating OpenSDA firmware Any NXP hardware platform that comes with an OpenSDA-compatible debug interface has the ability to update the OpenSDA firmware. This typically means to switch from the default application (either CMSIS-DAP or P&E Micro) to a SEGGER J-Link. This section contains the steps to switch the OpenSDA firmware to a J-Link interface. However, the steps can be applied to restoring the original image also. For reference, OpenSDA firmware files can be found at the links below:

- J-Link: Download appropriate image from www.segger.com/opensda.html. Choose the appropriate J-Link binary based on the table in [Default debug interfaces](#). Any OpenSDA v1.0 interface should use the standard OpenSDA download (in other words, the one with no version). For OpenSDA 2.0 or 2.1, select the corresponding binary.
- CMSIS-DAP: CMSIS-DAP OpenSDA firmware is available at www.nxp.com/opensda.
- P&E Micro: Downloading P&E Micro OpenSDA firmware images requires registration with P&E Micro (www.pemicro.com).

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

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



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

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

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

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

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

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

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

The host driver must be installed before debugging.

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

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

The host driver must be installed before debugging.

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

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

Hardware platform	Default debugger firmware	On-board debugger probe
EVK-MCIMX7ULP	N/A	N/A
EVK-MIMX8MM	N/A	N/A
EVK-MIMX8MN	N/A	N/A
EVK-MIMX8MNDDDR3L	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
- 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
- **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
- **Linux:** Use package manager or download from cmake.org
- **macOS:** Use Homebrew (`brew install cmake`) or download from 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
- **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
ARM GCC (Recommended)	Most users, free	ARM Toolchain GNU	ARMGCC_DIR
IAR EWARM	Professional development	IAR Systems	IAR_DIR
Keil MDK ARM Compiler	ARM ecosystem Advanced optimization	ARM Developer ARM Developer	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

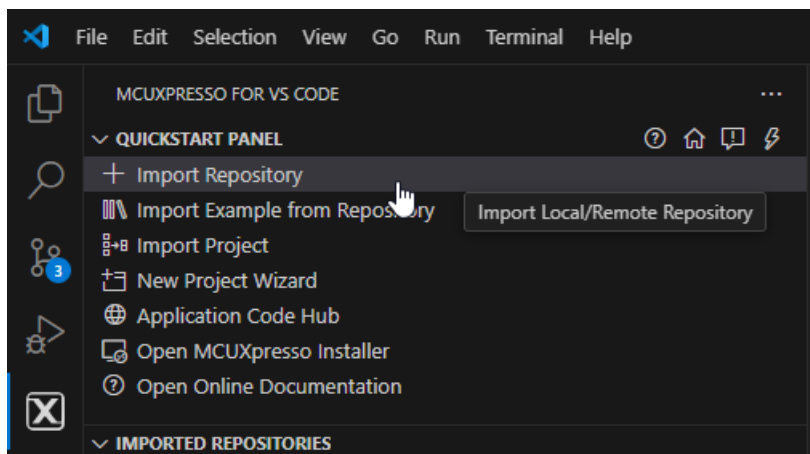
Extension Installation

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.

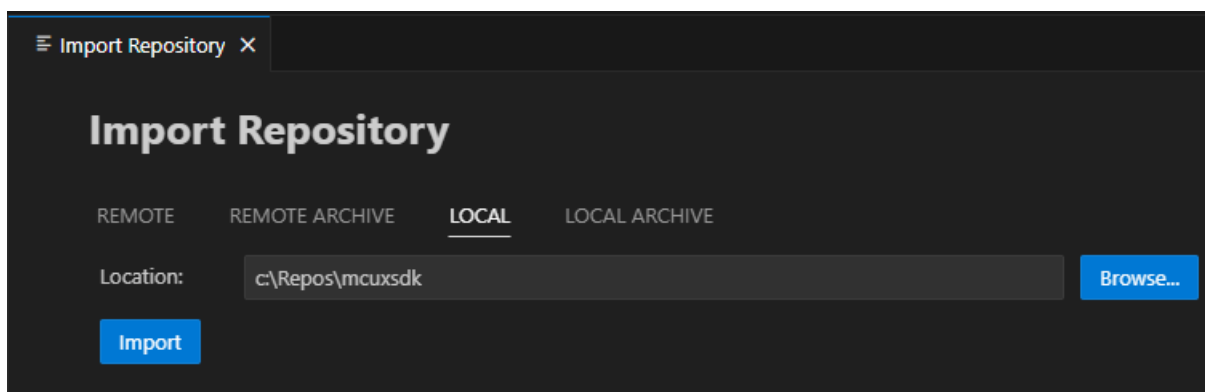
SDK Import and Setup

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.

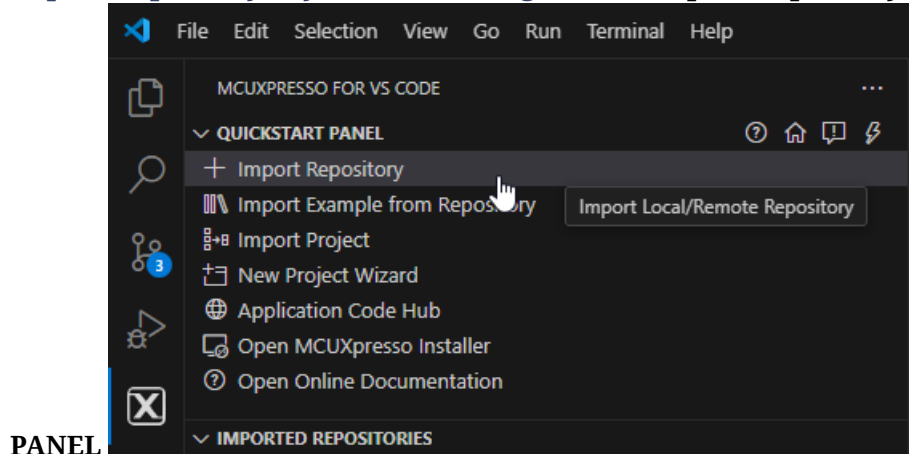
Import GitHub Repository SDK Click **Import Repository** from the **QUICKSTART PANEL**



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**.

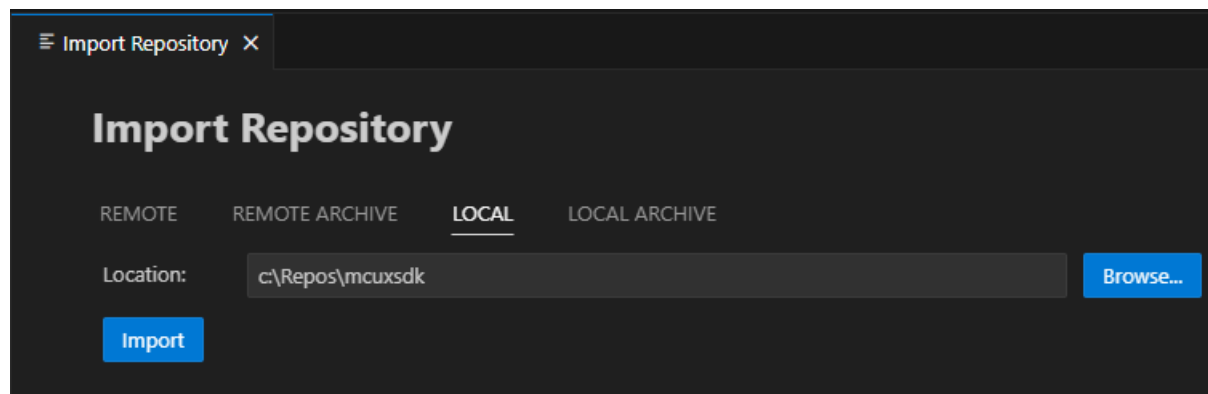


Import Repository-Layout SDK Package Click **Import Repository** from the **QUICKSTART**

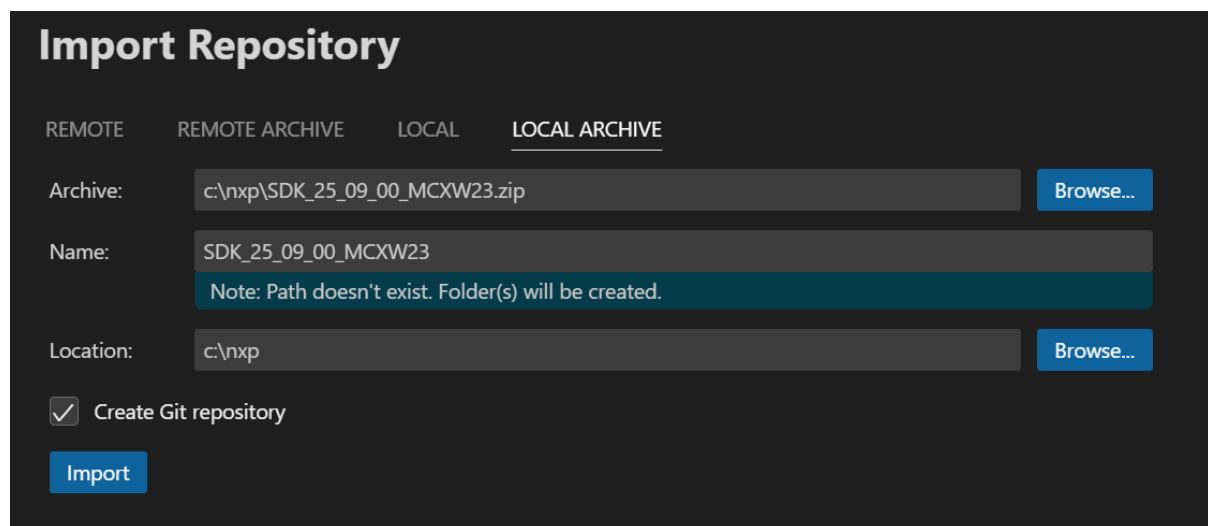


PANEL

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



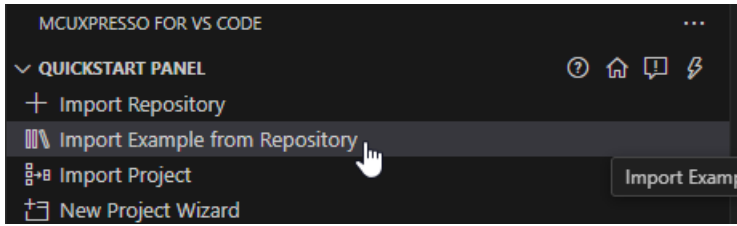
Else if the SDK is ZIP archive, select **Local Archive**, browse to the downloaded SDK ZIP file, fill the link of expect location, then click **Import**.



Building Example Applications

Import Example Project

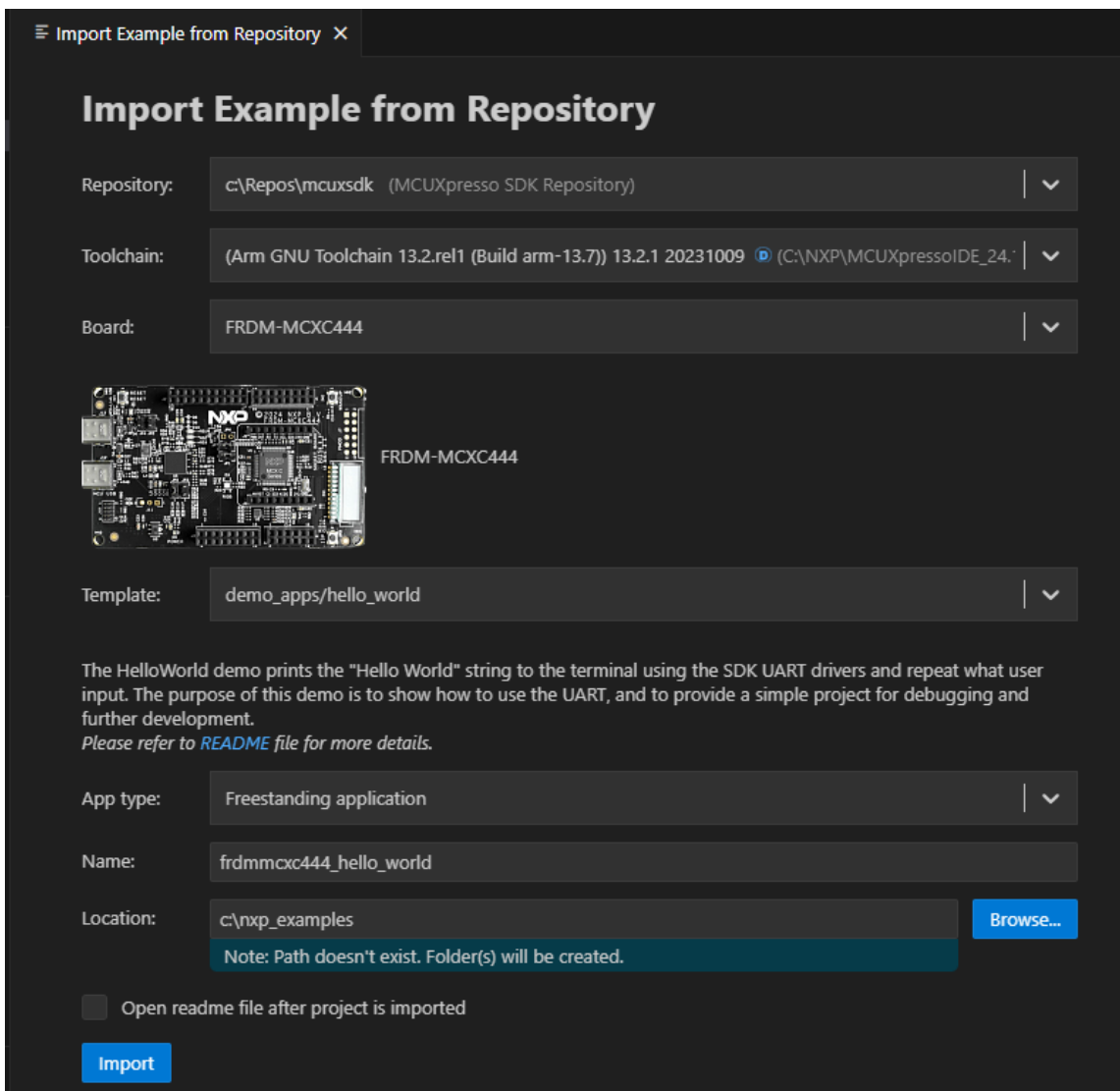
1. Click **Import Example from Repository** from the **QUICKSTART PANEL**



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:

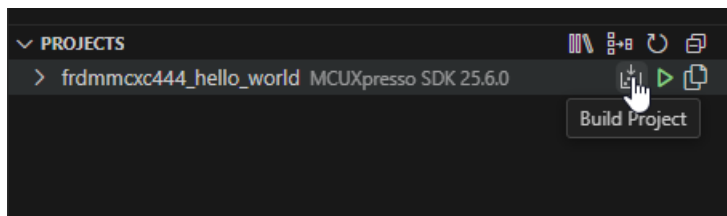
- Imported to user-defined location
- Independent of SDK location

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

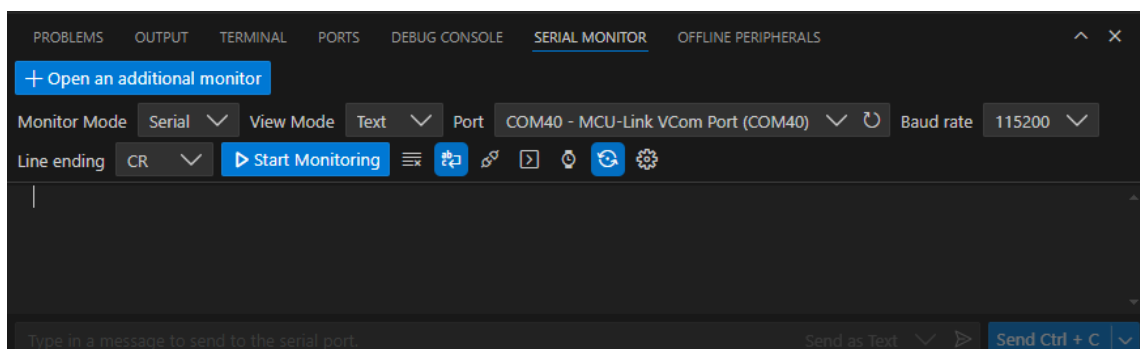


The integrated terminal will display build output at the bottom of the VS Code window.

Running and Debugging

Serial Monitor Setup

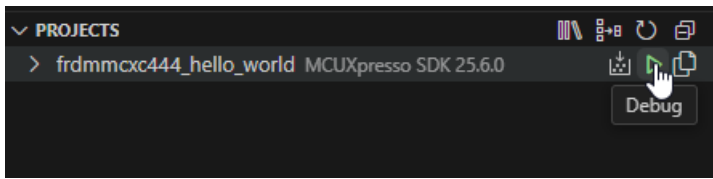
1. Open **Serial Monitor** from VS Code's integrated terminal



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

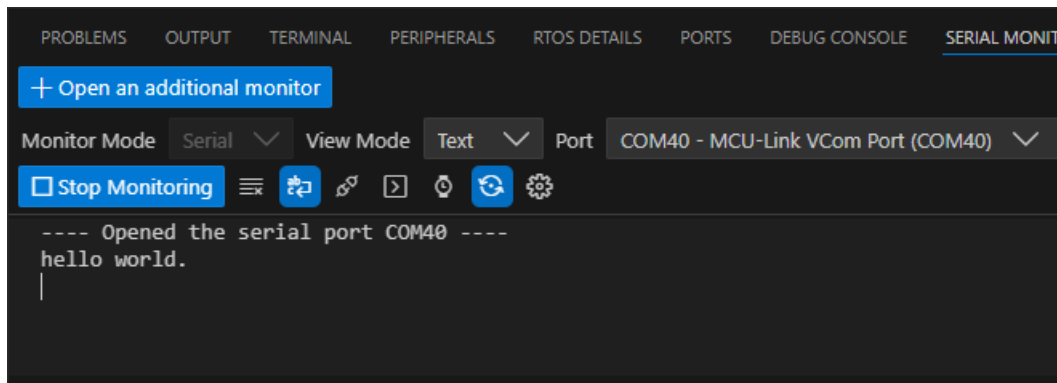
```

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
LPCX-presso860MAX	LPC864M201JBD64, LPC865M201JBD64, LPC864M201JHI33, LPC865M201JHI33, LPC864M201JHI48, LPC865M201JHI48

MCUXpresso SDK release package

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

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

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

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

RTOS

FreeRTOS Real-time operating system for microcontrollers from Amazon

Middleware

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

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

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.

The spi_transfer_interrupt examples don't work

Boards cannot transfer data successfully.

Affected toolchains: mcux **Affected platforms:** lpc845breakout, lpcxpresso860max

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]

- Improvements
 - Added `CLOCK_SetFLASHAccessCyclesForFreq`.

[2.3.3]

- Improvements
 - Added lost comments for some enumerations.
-

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.
-

LPC_DMA

[2.5.4]

- Bug Fixes
 - Fixed coverity issues with CERT INT30-C, CERT INT31-C compliance.

[2.5.3]

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

[2.5.2]

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

[2.5.1]

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

[2.5.0]

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

[2.4.4]

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

[2.4.3]

- Improvements
 - Added features FSL_FEATURE_DMA_DESCRIPTOR_ALIGN_SIZE_n/FSL_FEATURE_DMA0_DESCRIPTOR_ALIGN_SIZE_n to support the descriptor align size not constant in the two instances.

[2.4.2]

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

[2.4.1]

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

[2.4.0]

- Improvements
 - Added new APIs DMA_LoadChannelDescriptor/DMA_ChannelIsBusy to support polling transfer case.
- Bug Fixes
 - Added address alignment check for descriptor source and destination address.
 - Added DMA_ALLOCATE_DATA_TRANSFER_BUFFER for application buffer allocation.
 - Fixed the sign-compare warning.

- Fixed violations of the MISRA C-2012 rules 18.1, 10.4, 11.6, 10.7, 14.4, 16.3, 20.7, 10.8, 16.1, 17.7, 10.3, 3.1, 18.1.

[2.3.0]

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

[2.2.5]

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

[2.2.4]

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

[2.2.3]

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

[2.2.2]

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

[2.2.1]

- Bug Fixes
 - Fixed Coverity issue.

[2.2.0]

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

[2.0.1]

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

[2.0.0]

- Initial version.
-

FTM

[2.7.5]

- Bug Fixes
 - Corrected the bit shift calculation for fault input filter enable in FTM_SetupFaultInput().

[2.7.4]

- Bug Fixes
 - Fixed violations of the CERT INT31-C.
 - Fixed MISRA C-2012 issue: rule 10.1.

[2.7.3]

- Bug Fixes
 - Fixed violations of the CERT INT30-C INT31-C.

[2.7.2]

- Improvements
 - Add API FTM_ERRATA_010856 for ERR010856 workaround.

[2.7.1]

- Bug Fixes
 - Added function macro when accsee FLTCTRL register FSTATE bit to prevent access nonexistent register.
 - Added function macro to prevent access nonexistent FTM channel for API FTM_ConfigSinglePWM() and FTM_ConfigCombinePWM().

[2.7.0]

- Improvements
 - Support period dithering and edge dithering feature with new APIs:
 - * FTM_SetPeriodDithering()
 - * FTM_SetEdgeDithering()
 - Support get channel n output and input state feature with new APIs:
 - * FTM_GetChannelOutputState()
 - * FTM_GetChannelInputState()
 - Support configure deadtime for specific combined channel pair with new API:
 - * FTM_SetPairDeadTime()
 - Support filter clock prescale, fault output state.
 - Support new APIs to configure PWM and Modified Combine PWM:
 - * FTM_ConfigSinglePWM()
 - * FTM_ConfigCombinePWM()
 - Support new API to configure channel software output control:
 - * FTM_SetSoftwareOutputCtrl()
 - * FTM_GetSoftwareOutputValue()
 - * FTM_GetSoftwareOutputEnable()
 - Support new API to update FTM counter initial value, modulo value and chanle value:
 - * FTM_SetInitialModuloValue()
 - * FTM_SetChannelValue()

[2.6.1]

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

[2.6.0]

- Improvements
 - Added support to half and full cycle reload feature with new APIs:
 - * FTM_SetLdok()
 - * FTM_SetHalfCycPeriod()
 - * FTM_LoadFreq()
- Bug Fixes
 - Set the HWRSTCNT and SWRSTCNT bits to optional at initialization.

[2.5.0]

- Improvements
 - Added FTM_CalculateCounterClkDiv to help calculates the counter clock prescaler.
 - Modify FTM_UpdatePwmDutycycle API to make it return pwm duty cycles status.
- Bug Fixes
 - Fixed TPM_SetupPwm can't configure 100% center align combined PWM issues.

[2.4.1]

- Bug Fixes
 - Added function macro to determine if FTM instance has only basic features, to prevent access to protected register bits.

[2.4.0]

- Improvements
 - Added CNTIN register initialization in FTM_SetTimerPeriod API.
 - Added a new API to read the captured value of a FTM channel configured in capture mode:
 - * FTM_GetInputCaptureValue()

[2.3.0]

- Improvements
 - Added support of EdgeAligned/CenterAligned/Asymmetrical combine PWM mode in FTM_SetupPWM() and FTM_SetupPwmMode() APIs.
 - Remove kFTM_ComplementaryPwm from support PWM mode, and add new parameter “enableComplementary” in structure ftm_chnl_pwm_signal_param_t.
 - Rename FTM_SetupFault() API to FTM_SetupFaultInput() to avoid ambiguity.

[2.2.3]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 14.4 and 17.7.

[2.2.2]

- Bug Fixes
 - Fixed the issue that when FTM instance has only TPM features cannot be initialized by FTM_Init() function. By added function macro to assert FTM is TPM only instance.

[2.2.1]

- Bug Fixes
 - MISRA C-2012 issue fixed: rule 10.1, 10.3, 10.4, 10.6, 10.7 and 11.9.

[2.2.0]

- Bug Fixes
 - Fixed the issue of comparison between signed and unsigned integer expressions.
- Improvements
 - Added support of complementary mode in FTM_SetupPWM() and FTM_SetupPwmMode() APIs.
 - Added new parameter “enableDeadtime” in structure `ftm_chnl_pwm_signal_param_t`.

[2.1.1]

- Bug Fixes
 - Fixed COVERITY integer handing issue where the right operand of a left bit shift statement should not be a negative value. This appears in FTM_SetReloadPoints().

[2.1.0]

- Improvements
 - Added a new API FTM_SetupPwmMode() to allow the user to set the channel match value in units of timer ticks. New configure structure called `ftm_chnl_pwm_config_param_t` was added to configure the channel's PWM parameters. This API is similar with FTM_SetupPwm() API, but the new API will not set the timer period(MOD value), it will be useful for users to set the PWM parameters without changing the timer period.
- Bug Fixes
 - Added feature macro to enable/disable the external trigger source configuration.

[2.0.4]

- Improvements
 - Added a new API to enable DMA transfer:
 - * FTM_EnableDmaTransfer()

[2.0.3]

- Bug Fixes
 - Updated the FTM driver to enable fault input after configuring polarity.

[2.0.2]

- Improvements
 - Added support to Quad Decoder feature with new APIs:
 - * FTM_GetQuadDecoderFlags()
 - * FTM_SetQuadDecoderModuloValue()
 - * FTM_GetQuadDecoderCounterValue()
 - * FTM_ClearQuadDecoderCounterValue()

[2.0.1]

- Bug Fixes
 - Updated the FTM driver to fix write to ELSA and ELSB bits.
 - FTM combine mode: set the COMBINE bit before writing to CnV register.

[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 transfered 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_WAIT_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.
-

I3C

[2.14.4]

- Improvements
 - Fixed MSG violations.

[2.14.3]

- Improvements
 - Fixed Coverity CERT-C violations.
 - Used I3C_RSTS instead of I3C special feature macro.
 - Adapted the driver to support new platform.

[2.14.2]

- Improvements
 - Added timeout for ENTDAAs process API.
 - Added build system macro to control the timeout setting.

[2.14.1]

- Improvements
 - Split the function `I3C_MasterTransferBlocking` to meet the HIS-CCM requirement.

[2.14.0]

- Improvements
 - Added the choice to set fast start header with push-pull speed when all targets addresses have MSB 0 instead of forcing to set it.
 - Deleted duplicated busy check in `I3C_MasterStart` function.

[2.13.1]

- Bug Fixes
 - Disabled Rx auto-termination in repeated start interrupt event while transfer API doesn't enable it.
 - Waited the completion event after loading all Tx data in Tx FIFO.
- Improvements
 - Conditionally compile interrupt handling code to solve the problem of using this driver on CPU cores that do not support interrupts.

[2.13.0]

- New features
 - Added the hot-join support for I3C bus initialization API.
- Bug Fixes
 - Set read termination with START at the same time in case unknown issue.
 - Set `MCTRL[TYPE]` as 0 for DDR force exit.
- Improvements
 - Added the API to reset device count assigned by ENTDAAs.
 - Provided the method to set global macro `I3C_MAX_DEVCNT` to determine how many device addresses ENTDAAs can allocate at one time.
 - Initialized target management static array based on instance number for the case that multiple instances are used at the same time.

[2.12.0]

- Improvements
 - Added the slow clock parameter for Controller initialization function to calculate accurate timeout.
- Bug Fixes
 - Fixed the issue that BAMATCH field can't be 0. BAMATCH should be 1 for 1MHz slow clock.

[2.11.1]

- Bug Fixes
 - Fixed the issue that interrupt API transmits extra byte when subaddress and data size are null.
 - Fixed the slow clock calculation issue.

[2.11.0]

- New features
 - Added the START/ReSTART SCL delay setting for the Soc which supports this feature.
- Bug Fixes
 - Fixed the issue that ENTDA process waits Rx pending flag which causes problem when Rx watermark isn't 0. Just check the Rx FIFO count.

[2.10.8]

- Improvements
 - Support more instances.

[2.10.7]

- Improvements
 - Fixed the potential compile warning.

[2.10.6]

- New features
 - Added the I3C private read/write with 0x7E address as start.

[2.10.5]

- New features
 - Added I3C HDR-DDR transfer support.

[2.10.4]

- Improvements
 - Added one more option for master to not set RDTERM when doing I3C Common Command Code transfer.

[2.10.3]

- Improvements
 - Masked the slave IBI/MR/HJ request functions with feature macro.

[2.10.2]

- Bug Fixes
 - Added workaround for errata ERR051617: I3C working with I2C mode creates the unintended Repeated START before actual STOP on some platforms.

[2.10.1]

- Bug Fixes
 - Fixed the issue that DAA function doesn't wait until all Rx data is read out from FIFO after master control done flag is set.
 - Fixed the issue that DAA function could return directly although the disabled interrupts are not enabled back.

[2.10.0]

- New features
 - Added I3C extended IBI data support.

[2.9.0]

- Improvements
 - Added adaptive termination for master blocking transfer. Set termination with start signal when receiving bytes less than 256.

[2.8.2]

- Improvements
 - Fixed the build warning due to armgcc strict check.

[2.8.1]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 17.7.

[2.8.0]

- Improvements
 - Added API I3C_MasterProcessDAASpecifiedBaudrate for temporary baud rate adjustment when I3C master assigns dynamic address.

[2.7.1]

- Bug Fixes
 - Fixed the issue that I3C slave handle STOP event before finishing data transmission.

[2.7.0]

- Fixed the CCM problem in file fsl_i3c.c.
- Fixed the FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND usage issue in I3C_GetDefaultConfig and I3C_Init.

[2.6.0]

- Fixed the FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND usage issue in fsl_i3c.h.
- Changed some static functions in fsl_i3c.c as non-static and define the functions in fsl_i3c.h to make I3C DMA driver reuse:
 - I3C_GetIBIType
 - I3C_GetIBIAddress
 - I3C_SlaveCheckAndClearError
- Changed the handle pointer parameter in IRQ related functions to void * type to make it reuse in I3C DMA driver.
- Added new API I3C_SlaveRequestIBIWithSingleData for slave to request single data byte, this API could be used regardless slave is working in non-blocking interrupt or non-blocking dma.
- Added new API I3C_MasterGetDeviceListAfterDAA for master application to get the device information list built up in DAA process.

[2.5.4]

- Improved I3C driver to avoid setting state twice in the SendCommandState of I3C_RunTransferStateMachine.
- Fixed MISRA violation of rule 20.9.
- Fixed the issue that I3C_MasterEmitRequest did not use Type I3C SDR.

[2.5.3]

- Updated driver for new feature FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH and FSL_FEATURE_I3C_HAS_NO_SCONFIG_IDRAND.

[2.5.2]

- Updated driver for new feature FSL_FEATURE_I3C_HAS_NO_MERRWARN_TERM.
- Fixed the issue that call to I3C_MasterTransferBlocking API did not generate STOP signal when NAK status was returned.

[2.5.1]

- Improved the receive terminate size setting for interrupt transfer read, now it's set at beginning of transfer if the receive size is less than 256 bytes.

[2.5.0]

- Added new API `I3C_MasterRepeatedStartWithRxSize` to send repeated start signal with receive terminate size specified.
- Fixed the status used in `I3C_RunTransferStateMachine`, changed to use pending interrupts as status to be handled in the state machine.
- Fixed MISRA 2012 violation of rule 10.3, 10.7.

[2.4.0]

- Bug Fixes
 - Fixed `kI3C_SlaveMatchedFlag` interrupt is not properly handled in `I3C_SlaveTransferHandleIRQ` when it comes together with interrupt `kI3C_SlaveBusStartFlag`.
 - Fixed the inaccurate I2C baudrate calculation in `I3C_MasterSetBaudRate`.
 - Added new API `I3C_MasterGetIBIRules` to get registered IBI rules.
 - Added new variable `isReadTerm` in struct `_i3c_master_handle` for transfer state routine to check if `MCTRL.RDTERM` is configured for read transfer.
 - Changed to emit Auto IBI in transfer state routine for slave start flag assertion.
 - Fixed the slave `maxWriteLength` and `maxReadLength` does not be configured into `SMAXLIMITS` register issue.
 - Fixed incorrect state for IBI in I3C master interrupt transfer IRQ handle routine.
 - Added `isHotJoin` in `i3c_slave_config_t` to request hot-join event during slave init.

[2.3.2]

- Bug Fixes
 - Fixed violations of the MISRA C-2012 rules 8.4, 17.7.
 - Fixed incorrect HotJoin event index in `I3C_GetIBIType`.

[2.3.1]

- Bug Fixes
 - Fixed the issue that call of `I3C_MasterTransferBlocking/I3C_MasterTransferNonBlocking` fails for the case which receive length 1 byte of data.
 - Fixed the issue that STOP signal is not sent when NAK status is detected during execution of `I3C_MasterTransferBlocking` function.

[2.3.0]

- Improvements
 - Added I3C common driver APIs to initialize I3C with both master and slave configuration.
 - Updated I3C master transfer callback to function set structure to include callback invoke for IBI event and slave2master event.
 - Updated I3C master non-blocking transfer model and always enable the interrupts to be able to re-act to the slave start event and handle slave IBI.

[2.2.0]

- Bug Fixes
 - Fixed the issue that I3C transfer size limit to 255 bytes.

[2.1.2]

- Bug Fixes
 - Reset default hkeep value to kI3C_MasterHighKeeperNone in I3C_MasterGetDefaultConfig

[2.1.1]

- Bug Fixes
 - Fixed incorrect FIFO reset operation in I3C Master Transfer APIs.
 - Fixed i3c slave IRQ handler issue, slave transmit could be underrun because tx FIFO is not filled in time right after start flag detected.

[2.1.0]

- Added definitions and APIs for I3C slave functionality, updated previous I3C APIs to support I3C functionality.

[2.0.0]

- Initial version.
-

I3C_DMA

[2.1.10]

- Improvements
 - Added DMA chain optimization for Errata 052123.

[2.1.9]

- Improvements
 - Fixed Coverity CERT-C violations.

[2.1.8]

- Bug Fixes
 - Updated the logic to handle Rx termination and complete event to adapt different situation.
- Improvements
 - Added the MCTRLDONE flag check after STOP request to ensure the completion of whole transfer operation.

[2.1.7]

- Bug Fixes
 - Fixed the issue to use subaddress to read/write data with RT500/600 DMA.

[2.1.6]

- Improvements
 - Added the FSL_FEATURE_I3C_HAS_NO_MASTER_DMA_WDATA_REG to select the correct register to write data based on specific Soc.

[2.1.5]

- New features
 - Supported I3C HDR-DDR transfer with DMA.
- Improvements
 - Added workaround for RT500/600 I3C DMA transfer.
 - Removed I3C IRQ handler calling in the Tx EDMA callback. Previously driver doesn't use the END byte which can trigger the complete interrupt for controller sending and receiving, now let I3C event handler deal with I3C events.
 - Used linked DMA to transfer all I3C subaddress and data without handling of intermediate states, simplifying code logic.
 - Prepare the Tx DMA before I3C START to ensure there's no time delay between START and transmitting data.

[2.1.4]

- Improvements
 - Used linked DMA transfer to reduce the latency between DMA transfers previous data and the END byte.

[2.1.3]

- Bug Fixes
 - Fixed the MISRA issue rule 10.4, 11.3.

[2.1.2]

- Bug Fixes
 - Fixed the issue that I3C slave send the last byte data without using the END type register.

[2.1.1]

- Bug Fixes
 - Fixed MISRA issue rule 9.1.

[2.1.0]

- Improvements
 - Deleted legacy IBI data request code.

[2.0.1]

- Bug Fixes
 - Fixed issue that bus STOP occurs when Tx FIFO still takes data.
- Improvements
 - Fixed the build warning due to armgcc strict check.

[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.
-

INPUTMUX

[2.0.10]

- Bug Fixes
 - Fixed CERT-C violations.

[2.0.9]

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

[2.0.8]

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

[2.0.7]

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

[2.0.6]

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

[2.0.5]

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

[2.0.4]

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

[2.0.3]

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

[2.0.2]

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

[2.0.1]

- Support channel mux setting in INPUTMUX_EnableSignal().

[2.0.0]

- Initial version.
-

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]

- initial version.
-

RESET

[2.4.0]

- Improvements
 - Add RESET_ReleasePeripheralReset API.

[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
 - Chek 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 WWWDWT driver.
 - * WWDT_Disable

[2.1.4]

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

[2.1.3]

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

[2.1.2]

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

[2.1.1]

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

[2.1.0]

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

[2.0.0]

- Initial version.
-

1.6 Driver API Reference Manual

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

[LPC865](#)

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

1.7.2 FreeRTOS

FreeRTOS

Chapter 2

LPC865

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_1

Clock gate name: Ram0_1.

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_Spi1

Clock gate name: Spi1.

enumerator kCLOCK_Crc

Clock gate name: Crc.

enumerator kCLOCK_Uart0

Clock gate name: Uart0.

enumerator kCLOCK_Uart1
Clock gate name: Uart1.

enumerator kCLOCK_Uart2
Clock gate name: Uart2.

enumerator kCLOCK_Wwdt
Clock gate name: Wwdt.

enumerator kCLOCK_Iocon
Clock gate name: Iocon.

enumerator kCLOCK_Acmp
Clock gate name: Acmp.

enumerator kCLOCK_Gpio1
Clock gate name: Gpio1.

enumerator kCLOCK_Ftm0
Clock gate name: Ftm0.

enumerator kCLOCK_Ftm1
Clock gate name: Ftm1.

enumerator kCLOCK_I3c0
Clock gate name: I3c0.

enumerator kCLOCK_Adc
Clock gate name: Adc.

enumerator kCLOCK_GpioInt
Clock gate name: GpioInt.

enumerator kCLOCK_Dma
Clock gate name: Dma.

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_PllOut
PLL Output

enumerator kCLOCK_LpOsc
Low power Oscillator

enumerator kCLOCK_Frg0
fractional rate0

enumerator kCLOCK_Frg1
fractional rate1

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_SysPll_DIV
Mux ADC_Clk from SysPllDiv.

enumerator kEXT_Clk_From_SysOsc
Mux EXT_Clk from SysOsc.

enumerator kEXT_Clk_From_ClkIn
Mux EXT_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_Frg1Clk
Mux UART0_Clk from Frg1Clk.

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_Frg1Clk
Mux UART1_Clk from Frg1Clk.

enumerator kUART1_Clk_From_Fro_Div
Mux UART1_Clk from Fro_Div.

enumerator kUART2_Clk_From_Fro
Mux UART2_Clk from Fro.

enumerator kUART2_Clk_From_MainClk
Mux UART2_Clk from MainClk.

enumerator kUART2_Clk_From_Frg0Clk
Mux UART2_Clk from Frg0Clk.

enumerator kUART2_Clk_From_Frg1Clk
Mux UART2_Clk from Frg1Clk.

enumerator kUART2_Clk_From_Fro_Div
Mux UART2_Clk from Fro_Div.

enumerator kI2C0_Clk_From_Fro
Mux I2C0_Clk from Fro.

enumerator kI2C0_Clk_From_MainClk
Mux I2C0_Clk from MainClk.

enumerator kI2C0_Clk_From_Frg0Clk
Mux I2C0_Clk from Frg0Clk.

enumerator kI2C0_Clk_From_Frg1Clk
Mux I2C0_Clk from Frg1Clk.

enumerator kI2C0_Clk_From_Fro_Div
Mux I2C0_Clk from Fro_Div.

enumerator kI3C_Clk_From_Fro
Mux I3C_Clk from Fro.

enumerator kI3C_Clk_From_ExtClk
Mux I3C_Clk from ExtClk.

enumerator kI3C_TC_Clk_From_I3C_Clk
Mux I3C_TC_Clk from I3C_Clk.

enumerator kI3C_TC_Clk_From_LpOsc
Mux I3C_TC_Clk from LpOsc.

enumerator kSPI0_Clk_From_Fro
Mux SPI0_Clk from Fro.

enumerator kSPI0_Clk_From_MainClk
Mux SPI0_Clk from MainClk.

enumerator kSPI0_Clk_From_Frg0Clk
Mux SPI0_Clk from Frg0Clk.

enumerator kSPI0_Clk_From_Frg1Clk
Mux SPI0_Clk from Frg1Clk.

enumerator kSPI0_Clk_From_Fro_Div
Mux SPI0_Clk from Fro_Div.

enumerator kSPI1_Clk_From_Fro
Mux SPI1_Clk from Fro.

enumerator kSPI1_Clk_From_MainClk
Mux SPI1_Clk from MainClk.

enumerator kSPI1_Clk_From_Frg0Clk
Mux SPI1_Clk from Frg0Clk.

enumerator kSPI1_Clk_From_Frg1Clk
Mux SPI1_Clk from Frg1Clk.

enumerator kSPI1_Clk_From_Fro_Div
Mux SPI1_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 kFRG0_Clk_From_SysPll_DIV
Mux FRG0_Clk from SysPllDiv.

enumerator kFRG1_Clk_From_Fro
Mux FRG1_Clk from Fro.

enumerator kFRG1_Clk_From_MainClk
Mux FRG1_Clk from MainClk.

enumerator kFRG1_Clk_From_SysPll_DIV
Mux FRG1_Clk from SysPllDiv.

enumerator kCLKOUT_From_Fro
Mux CLKOUT from Fro.

enumerator kCLKOUT_From_MainClk
Mux CLKOUT from MainClk.

enumerator kCLKOUT_From_SysPll_DIV
Mux CLKOUT from SysPllDiv.

enumerator kCLKOUT_From_ExtClk
Mux CLKOUT from ExtClk.

enumerator kCLKOUT_From_LpOsc
Mux CLKOUT from LpOsc.

enumerator kWKT_Clk_From_Fro
Mux Wkt_Clk from FroOsc.

enumerator kWKT_Clk_From_LpOsc
Mux Wkt_Clk from LpOsc.

enum _clock_divider

Clock divider.

Values:

enumerator kCLOCK_DivPllClk
Pll Clock Divider.

enumerator kCLOCK_DivAdcClk
Adc Clock Divider.

enumerator kCLOCK_DivClkOut
Clk Out Divider.

enumerator kCLOCK_IOCONCLKDiv6
IOCON Clock Div6 Divider.

enumerator kCLOCK_IOCONCLKDiv5
IOCON Clock Div5 Divider.

enumerator kCLOCK_IOCONCLKDiv4
IOCON Clock Div4 Divider.

enumerator kCLOCK_IOCONCLKDiv3
IOCON Clock Div3 Divider.

enumerator kCLOCK_IOCONCLKDiv2
IOCON Clock Div2 Divider.

enumerator kCLOCK_IOCONCLKDiv1
IOCON Clock Div1 Divider.

enumerator kCLOCK_IOCONCLKDiv0
IOCON Clock Div0 Divider.

enum _clock_fro_src
fro output frequency source definition

Values:

enumerator kCLOCK_FroSrcFroOscDiv
fro source from the fro oscillator divided by 2

enumerator kCLOCK_FroSrcFroOsc
fre source from the fro oscillator directly

enum _clock_fro_osc_freq
fro oscillator output frequency value definition

Values:

enumerator kCLOCK_FroOscOut36M
FRO oscillator output 36M

enumerator kCLOCK_FroOscOut48M
FRO oscillator output 48M

enumerator kCLOCK_FroOscOut60M
FRO oscillator output 60M

enum _clock_sys_pll_src
PLL clock definition.

Values:

enumerator kCLOCK_SysPllSrcFRO
system pll source from FRO

enumerator kCLOCK_SysPllSrcExtClk
system pll source from external clock

enumerator kCLOCK_SysPllSrcLpOsc
system pll source from Low power oscillator

enumerator kCLOCK_SysPllSrcFroDiv
system pll source from FRO divided clock

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 Low power oscillator

enumerator kCLOCK_MainClkSrcFroDiv
main clock source from FRO Div

enumerator kCLOCK_MainClkSrcSysPll
main clock source from system pll

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_sys_pll_src* clock_sys_pll_src
PLL clock definition.

typedef enum *_clock_main_clk_src* clock_main_clk_src_t
Main clock source definition.

typedef struct *_clock_sys_pll* clock_sys_pll_t
PLL configuration structure.

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 2.3.3.

SDK_DEVICE_MAXIMUM_CPU_CLOCK_FREQUENCY

g_Lp_Osc_Freq
Low power oscilltor clock frequency.

Definition for the low power oscillator frequency which is 1 Mhz at default, and it is returned by CLOCK_GetLpOscFreq.

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.

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.

I2C_CLOCKS

Clock ip name array for I2C.

USART_CLOCKS

Clock ip name array for I2C.

SPI_CLOCKS

Clock ip name array for SPI.

MRT_CLOCKS

Clock ip name array for MRT.

WKT_CLOCKS

Clock ip name array for WKT.

FTM_CLOCKS

Clock ip name array for FLEXTMR.

I3C_CLOCKS

Clock ip name array for I3C.

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
SYS_AHB_CLK_CTRL1
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)
static inline void CLOCK_SetI3CFClkDiv(uint32_t value)
static inline void CLOCK_SetI3CTCClkDiv(uint32_t value)
static inline void CLOCK_SetI3CSClkDiv(uint32_t value)
void CLOCK_SetMainClkSrc(clock_main_clk_src_t src)
    Set main clock reference source.

```

Parameters

- *src* – Refer to *clock_main_clk_src_t* to set the main clock source.

```

void CLOCK_SetFroOutClkSrc(clock_fro_src_t src)
    Set FRO clock source.

```

Parameters

- *src* – Please refer to *_clock_fro_src* definition.

```

static inline void CLOCK_SetFRGClkMul(uint32_t *base, uint32_t mul)

```

void CLOCK_SetFLASHAccessCyclesForFreq(uint32_t iFreq)

Set the flash wait states for the input frequency.

Parameters

- iFreq – : Input frequency

uint32_t CLOCK_GetFRG0ClkFreq(void)

Return Frequency of FRG0 Clock.

Returns

Frequency of FRG0 Clock.

uint32_t CLOCK_GetFRG1ClkFreq(void)

Return Frequency of FRG1 Clock.

Returns

Frequency of FRG1 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_GetUart2ClkFreq(void)

Get UART2 frequency.

Return values

UART2 – frequency value.

uint32_t CLOCK_GetUart3ClkFreq(void)

Get UART3 frequency.

Return values

UART3 – frequency value.

uint32_t CLOCK_GetUart4ClkFreq(void)

Get UART4 frequency.

Return values

UART4 – frequency value.

uint32_t CLOCK_GetI3cClkFreq(void)

Get I3C frequency.

Return values

I3C – frequency value.

uint32_t CLOCK_GetLpOscClkFreq(void)

Get LP_OSC frequency.

Return values

LP_OSC – frequency value.

uint32_t CLOCK_GetFreq(*clock_name_t* clockName)

Return Frequency of selected clock.

Returns

Frequency of selected clock

uint32_t CLOCK_GetSystemPLLInClockRate(void)

Return System PLL input clock rate.

Returns

System PLL input clock rate

static inline uint32_t CLOCK_GetSystemPLLFreq(void)

Return Frequency of System PLL.

Returns

Frequency of PLL

static inline uint32_t CLOCK_GetLpOscFreq(void)

Get low power OSC frequency.

Return values

low – power OSC frequency value.

static inline uint32_t CLOCK_GetExtClkFreq(void)

Get external clock frequency.

Return values

external – clock frequency value.

void CLOCK_InitSystemPll(const *clock_sys_pll_t* *config)

System PLL initialize.

Parameters

- config – System PLL configurations.

static inline void CLOCK_DenitSystemPll(void)

System PLL Deinitialize.

bool CLOCK_SetFRG0ClkFreq(uint32_t freq)

Set FRG0 output frequency.

Parameters

- freq – Target output frequency, $freq < input$ and $(input / freq) < 2$ should be satisfy.

Return values

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

bool CLOCK_SetFRG1ClkFreq(uint32_t freq)

Set FRG1 output frequency.

Parameters

- freq – Target 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.

void CLOCK_InitSysOsc(uint32_t oscFreq)

Init SYS OSC.

Parameters

- oscFreq – oscillator frequency value.

void CLOCK_InitXtalin(uint32_t xtalinFreq)

XTALIN init function system oscillator is bypassed, sys_osc_clk is fed directly from the XTALIN.

Parameters

- xtalinFreq – XTALIN frequency value

Returns

Frequency of PLL

static inline void CLOCK_DeinitSysOsc(void)

Deinit SYS OSC.

static inline void CLOCK_SetFroOscFreq(*clock_fro_osc_freq_t* freq)

Set FRO oscillator output frequency. Initialize the FRO clock to given frequency (36, 48 or 60 MHz).

Parameters

- freq – Please refer to *clock_fro_osc_freq_t* definition, frequency must be one of 36000, 48000 or 60000 KHz.

uint32_t targetFreq

System pll fclk output frequency, the output frequency should be lower than 100MHz

clock_sys_pll_src src

System pll clock source

struct _clock_sys_pll

#include <fsl_clock.h> PLL configuration structure.

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 DMA: Direct Memory Access Controller Driver

```
void DMA_Init(DMA_Type *base)
```

Initializes DMA peripheral.

This function enable the DMA clock, set descriptor table and enable DMA peripheral.

Parameters

- base – DMA peripheral base address.

```
void DMA_Deinit(DMA_Type *base)
```

Deinitializes DMA peripheral.

This function gates the DMA clock.

Parameters

- base – DMA peripheral base address.

void DMA_InstallDescriptorMemory(DMA_Type *base, void *addr)

Install DMA descriptor memory.

This function used to register DMA descriptor memory for linked transfer, a typical case is ping pong transfer which will request more than one DMA descriptor memory space, although current DMA driver has a default DMA descriptor buffer, but it support one DMA descriptor for one channel only.

Parameters

- base – DMA base address.
- addr – DMA descriptor address

static inline bool DMA_ChannelIsActive(DMA_Type *base, uint32_t channel)

Return whether DMA channel is processing transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

True for active state, false otherwise.

static inline bool DMA_ChannelIsBusy(DMA_Type *base, uint32_t channel)

Return whether DMA channel is busy.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

True for busy state, false otherwise.

static inline void DMA_EnableChannelInterrupts(DMA_Type *base, uint32_t channel)

Enables the interrupt source for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableChannelInterrupts(DMA_Type *base, uint32_t channel)

Disables the interrupt source for the DMA transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_EnableChannel(DMA_Type *base, uint32_t channel)

Enable DMA channel.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DisableChannel(DMA_Type *base, uint32_t channel)

Disable DMA channel.

Parameters

- base – DMA peripheral base address.

- channel – DMA channel number.

```
static inline void DMA_EnableChannelPeriphRq(DMA_Type *base, uint32_t channel)
```

Set PERIPHREQEN of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

```
static inline void DMA_DisableChannelPeriphRq(DMA_Type *base, uint32_t channel)
```

Get PERIPHREQEN value of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

True for enabled PeriphRq, false for disabled.

```
void DMA_ConfigureChannelTrigger(DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger)
```

Set trigger settings of DMA channel.

Deprecated:

Do not use this function. It has been superceded by DMA_SetChannelConfig.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- trigger – trigger configuration.

```
void DMA_SetChannelConfig(DMA_Type *base, uint32_t channel, dma_channel_trigger_t *trigger, bool isPeriph)
```

set channel config.

This function provide a interface to configure channel configuration registers.

Parameters

- base – DMA base address.
- channel – DMA channel number.
- trigger – channel configurations structure.
- isPeriph – true is periph request, false is not.

```
static inline uint32_t DMA_SetChannelXferConfig(bool reload, bool clrTrig, bool intA, bool intB, uint8_t width, uint8_t srcInc, uint8_t dstInc, uint32_t bytes)
```

DMA channel xfer transfer configurations.

Parameters

- reload – true is reload link descriptor after current exhaust, false is not
- clrTrig – true is clear trigger status, wait software trigger, false is not
- intA – enable interruptA
- intB – enable interruptB

- width – transfer width
- srcInc – source address interleave size
- dstInc – destination address interleave size
- bytes – transfer bytes

Returns

The vaule of xfer config

uint32_t DMA_GetRemainingBytes(DMA_Type *base, uint32_t channel)

Gets the remaining bytes of the current DMA descriptor transfer.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

The number of bytes which have not been transferred yet.

static inline void DMA_SetChannelPriority(DMA_Type *base, uint32_t channel, dma_priority_t priority)

Set priority of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.
- priority – Channel priority value.

static inline dma_priority_t DMA_GetChannelPriority(DMA_Type *base, uint32_t channel)

Get priority of channel configuration register.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

Returns

Channel priority value.

static inline void DMA_SetChannelConfigValid(DMA_Type *base, uint32_t channel)

Set channel configuration valid.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_DoChannelSoftwareTrigger(DMA_Type *base, uint32_t channel)

Do software trigger for the channel.

Parameters

- base – DMA peripheral base address.
- channel – DMA channel number.

static inline void DMA_LoadChannelTransferConfig(DMA_Type *base, uint32_t channel, uint32_t xfer)

Load channel transfer configurations.

Parameters

- base – DMA peripheral base address.

- channel – DMA channel number.
- xfer – transfer configurations.

```
void DMA_CreateDescriptor(dma_descriptor_t *desc, dma_xfercfg_t *xfercfg, void *srcAddr, void *dstAddr, void *nextDesc)
```

Create application specific DMA descriptor to be used in a chain in transfer.

Deprecated:

Do not use this function. It has been superceded by DMA_SetupDescriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcAddr – Address of last item to transmit
- dstAddr – Address of last item to receive.
- nextDesc – Address of next descriptor in chain.

```
void DMA_SetupDescriptor(dma_descriptor_t *desc, uint32_t xfercfg, void *srcStartAddr, void *dstStartAddr, void *nextDesc)
```

setup dma descriptor

Note: This function do not support configure wrap descriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcStartAddr – Start address of source address.
- dstStartAddr – Start address of destination address.
- nextDesc – Address of next descriptor in chain.

```
void DMA_SetupChannelDescriptor(dma_descriptor_t *desc, uint32_t xfercfg, void *srcStartAddr, void *dstStartAddr, void *nextDesc, dma_burst_wrap_t wrapType, uint32_t burstSize)
```

setup dma channel descriptor

Note: This function support configure wrap descriptor.

Parameters

- desc – DMA descriptor address.
- xfercfg – Transfer configuration for DMA descriptor.
- srcStartAddr – Start address of source address.
- dstStartAddr – Start address of destination address.
- nextDesc – Address of next descriptor in chain.
- wrapType – burst wrap type.
- burstSize – burst size, reference `_dma_burst_size`.

```
void DMA_LoadChannelDescriptor(DMA_Type *base, uint32_t channel, dma_descriptor_t
                             *descriptor)
```

load channel transfer decriptor.

This function can be used to load decriptor to driver internal channel descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, it is useful for the case:

- a. for the polling transfer, application can allocate a local descriptor memory table to prepare a descriptor firstly and then call this api to load the configured descriptor to driver descriptor table.

```
DMA_Init(DMA0);
DMA_EnableChannel(DMA0, DEMO_DMA_CHANNEL);
DMA_SetupDescriptor(desc, xferCfg, s_srcBuffer, &s_destBuffer[0], NULL);
DMA_LoadChannelDescriptor(DMA0, DEMO_DMA_CHANNEL, (dma_descriptor_t *)desc);
DMA_DoChannelSoftwareTrigger(DMA0, DEMO_DMA_CHANNEL);
while(DMA_ChannelIsBusy(DMA0, DEMO_DMA_CHANNEL))
{ }
```

Parameters

- base – DMA base address.
- channel – DMA channel.
- descriptor – configured DMA descriptor.

```
void DMA_AbortTransfer(dma_handle_t *handle)
```

Abort running transfer by handle.

This function aborts DMA transfer specified by handle.

Parameters

- handle – DMA handle pointer.

```
void DMA_CreateHandle(dma_handle_t *handle, DMA_Type *base, uint32_t channel)
```

Creates the DMA handle.

This function is called if using transaction API for DMA. This function initializes the internal state of DMA handle.

Parameters

- handle – DMA handle pointer. The DMA handle stores callback function and parameters.
- base – DMA peripheral base address.
- channel – DMA channel number.

```
void DMA_SetCallback(dma_handle_t *handle, dma_callback callback, void *userData)
```

Installs a callback function for the DMA transfer.

This callback is called in DMA IRQ handler. Use the callback to do something after the current major loop transfer completes.

Parameters

- handle – DMA handle pointer.
- callback – DMA callback function pointer.
- userData – Parameter for callback function.

```
void DMA_PrepareTransfer(dma_transfer_config_t *config, void *srcAddr, void *dstAddr,
                        uint32_t byteWidth, uint32_t transferBytes, dma_transfer_type_t
                        type, void *nextDesc)
```

Prepares the DMA transfer structure.

Deprecated:

Do not use this function. It has been superceded by DMA_PrepareChannelTransfer. This function prepares the transfer configuration structure according to the user input.

Note: The data address and the data width must be consistent. For example, if the SRC is 4 bytes, so the source address must be 4 bytes aligned, or it shall result in source address error(SAE).

Parameters

- config – The user configuration structure of type *dma_transfer_t*.
- srcAddr – DMA transfer source address.
- dstAddr – DMA transfer destination address.
- byteWidth – DMA transfer destination address width(bytes).
- transferBytes – DMA transfer bytes to be transferred.
- type – DMA transfer type.
- nextDesc – Chain custom descriptor to transfer.

```
void DMA_PrepareChannelTransfer(dma_channel_config_t *config, void *srcStartAddr, void
                               *dstStartAddr, uint32_t xferCfg, dma_transfer_type_t type,
                               dma_channel_trigger_t *trigger, void *nextDesc)
```

Prepare channel transfer configurations.

This function used to prepare channel transfer configurations.

Parameters

- config – Pointer to DMA channel transfer configuration structure.
- srcStartAddr – source start address.
- dstStartAddr – destination start address.
- xferCfg – xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
- type – transfer type.
- trigger – DMA channel trigger configurations.
- nextDesc – address of next descriptor.

```
status_t DMA_SubmitTransfer(dma_handle_t *handle, dma_transfer_config_t *config)
```

Submits the DMA transfer request.

Deprecated:

Do not use this function. It has been superceded by DMA_SubmitChannelTransfer.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time.

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.

Return values

- kStatus_DMA_Success – It means submit transfer request succeed.
- kStatus_DMA_QueueFull – It means TCD queue is full. Submit transfer request is not allowed.
- kStatus_DMA_Busy – It means the given channel is busy, need to submit request later.

```
void DMA_SubmitChannelTransferParameter(dma_handle_t *handle, uint32_t xferCfg, void
                                     *srcStartAddr, void *dstStartAddr, void *nextDesc)
```

Submit channel transfer paramter directly.

This function used to configue channel head descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, it is useful for the case:

- for the single transfer, application doesn't need to allocate descriptor table, the head descriptor can be used for it.

```
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig,
↪ intA, intB, width, srcInc, dstInc,
bytes), srcStartAddr, dstStartAddr, NULL);
DMA_StartTransfer(handle)
```

- for the linked transfer, application should responsible for link descriptor, for example, if 4 transfer is required, then application should prepare three descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[3]);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, NULL);
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelTransferParameter(handle, DMA_CHANNEL_XFER(reload, clrTrig,
↪ intA, intB, width, srcInc, dstInc,
bytes), srcStartAddr, dstStartAddr, nextDesc0);
DMA_StartTransfer(handle);
```

Parameters

- handle – Pointer to DMA handle.
- xferCfg – xfer configuration, user can reference DMA_CHANNEL_XFER about to how to get xferCfg value.
- srcStartAddr – source start address.
- dstStartAddr – destination start address.

- nextDesc – address of next descriptor.

void DMA_SubmitChannelDescriptor(*dma_handle_t* *handle, *dma_descriptor_t* *descriptor)

Submit channel descriptor.

This function used to configure channel head descriptor that is used to start DMA transfer, the head descriptor table is defined in DMA driver, this function is typical for the ping pong case:

- for the ping pong case, application should be responsible for the descriptor, for example, application should prepare two descriptor table with macro.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc[2]);

DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc0);
DMA_SetChannelConfig(base, channel, trigger, isPeriph);
DMA_CreateHandle(handle, base, channel)
DMA_SubmitChannelDescriptor(handle, nextDesc0);
DMA_StartTransfer(handle);
```

Parameters

- handle – Pointer to DMA handle.
- descriptor – descriptor to submit.

status_t DMA_SubmitChannelTransfer(*dma_handle_t* *handle, *dma_channel_config_t* *config)

Submits the DMA channel transfer request.

This function submits the DMA transfer request according to the transfer configuration structure. If the user submits the transfer request repeatedly, this function packs an unprocessed request as a TCD and enables scatter/gather feature to process it in the next time. It is used for the case:

- for the single transfer, application doesn't need to allocate descriptor table, the head descriptor can be used for it.

```
DMA_CreateHandle(handle, base, channel)
DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,NULL);
DMA_SubmitChannelTransfer(handle, config)
DMA_StartTransfer(handle)
```

- for the linked transfer, application should be responsible for link descriptor, for example, if 4 transfer is required, then application should prepare three descriptor table with macro, the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);
DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc2);
DMA_SetupDescriptor(nextDesc2, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
```

(continues on next page)

(continued from previous page)

```
srcStartAddr, dstStartAddr, NULL);
    DMA_CreateHandle(handle, base, channel)
    DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,
↪nextDesc0);
    DMA_SubmitChannelTransfer(handle, config)
    DMA_StartTransfer(handle)
```

- c. for the ping pong case, application should responsible for link descriptor, for example, application should prepare two descriptor table with macro , the head descriptor in driver can be used for the first transfer descriptor.

```
define link descriptor table in application with macro
DMA_ALLOCATE_LINK_DESCRIPTOR(nextDesc);

    DMA_SetupDescriptor(nextDesc0, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc1);
    DMA_SetupDescriptor(nextDesc1, DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width,
↪ srcInc, dstInc, bytes),
srcStartAddr, dstStartAddr, nextDesc0);
    DMA_CreateHandle(handle, base, channel)
    DMA_PrepareChannelTransfer(config,srcStartAddr,dstStartAddr,xferCfg,type,trigger,
↪nextDesc0);
    DMA_SubmitChannelTransfer(handle, config)
    DMA_StartTransfer(handle)
```

Parameters

- handle – DMA handle pointer.
- config – Pointer to DMA transfer configuration structure.

Return values

- kStatus_DMA_Success – It means submit transfer request succeed.
- kStatus_DMA_QueueFull – It means TCD queue is full. Submit transfer request is not allowed.
- kStatus_DMA_Busy – It means the given channel is busy, need to submit request later.

```
void DMA_StartTransfer(dma_handle_t *handle)
```

DMA start transfer.

This function enables the channel request. User can call this function after submitting the transfer request It will trigger transfer start with software trigger only when hardware trigger is not used.

Parameters

- handle – DMA handle pointer.

```
void DMA_IRQHandle(DMA_Type *base)
```

DMA IRQ handler for descriptor transfer complete.

This function clears the channel major interrupt flag and call the callback function if it is not NULL.

Parameters

- base – DMA base address.

FSL_DMA_DRIVER_VERSION

DMA driver version.

Version 2.5.4.

`_dma_transfer_status` DMA transfer status

Values:

enumerator `kStatus_DMA_Busy`

Channel is busy and can't handle the transfer request.

`_dma_addr_interleave_size` dma address interleave size

Values:

enumerator `kDMA_AddressInterleave0xWidth`

dma source/destination address no interleave

enumerator `kDMA_AddressInterleave1xWidth`

dma source/destination address interleave 1xwidth

enumerator `kDMA_AddressInterleave2xWidth`

dma source/destination address interleave 2xwidth

enumerator `kDMA_AddressInterleave4xWidth`

dma source/destination address interleave 3xwidth

`_dma_transfer_width` dma transfer width

Values:

enumerator `kDMA_Transfer8BitWidth`

dma channel transfer bit width is 8 bit

enumerator `kDMA_Transfer16BitWidth`

dma channel transfer bit width is 16 bit

enumerator `kDMA_Transfer32BitWidth`

dma channel transfer bit width is 32 bit

enum `_dma_priority`

DMA channel priority.

Values:

enumerator `kDMA_ChannelPriority0`

Highest channel priority - priority 0

enumerator `kDMA_ChannelPriority1`

Channel priority 1

enumerator `kDMA_ChannelPriority2`

Channel priority 2

enumerator `kDMA_ChannelPriority3`

Channel priority 3

enumerator `kDMA_ChannelPriority4`

Channel priority 4

enumerator kDMA_ChannelPriority5
Channel priority 5

enumerator kDMA_ChannelPriority6
Channel priority 6

enumerator kDMA_ChannelPriority7
Lowest channel priority - priority 7

enum _dma_int

DMA interrupt flags.

Values:

enumerator kDMA_IntA
DMA interrupt flag A

enumerator kDMA_IntB
DMA interrupt flag B

enumerator kDMA_IntError
DMA interrupt flag error

enum _dma_trigger_type

DMA trigger type.

Values:

enumerator kDMA_NoTrigger
Trigger is disabled

enumerator kDMA_LowLevelTrigger
Low level active trigger

enumerator kDMA_HighLevelTrigger
High level active trigger

enumerator kDMA_FallingEdgeTrigger
Falling edge active trigger

enumerator kDMA_RisingEdgeTrigger
Rising edge active trigger

_dma_burst_size DMA burst size

Values:

enumerator kDMA_BurstSize1
burst size 1 transfer

enumerator kDMA_BurstSize2
burst size 2 transfer

enumerator kDMA_BurstSize4
burst size 4 transfer

enumerator kDMA_BurstSize8
burst size 8 transfer

enumerator kDMA_BurstSize16
burst size 16 transfer

enumerator kDMA_BurstSize32

burst size 32 transfer

enumerator kDMA_BurstSize64

burst size 64 transfer

enumerator kDMA_BurstSize128

burst size 128 transfer

enumerator kDMA_BurstSize256

burst size 256 transfer

enumerator kDMA_BurstSize512

burst size 512 transfer

enumerator kDMA_BurstSize1024

burst size 1024 transfer

enum _dma_trigger_burst

DMA trigger burst.

Values:

enumerator kDMA_SingleTransfer

Single transfer

enumerator kDMA_LevelBurstTransfer

Burst transfer driven by level trigger

enumerator kDMA_EdgeBurstTransfer1

Perform 1 transfer by edge trigger

enumerator kDMA_EdgeBurstTransfer2

Perform 2 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer4

Perform 4 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer8

Perform 8 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer16

Perform 16 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer32

Perform 32 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer64

Perform 64 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer128

Perform 128 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer256

Perform 256 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer512

Perform 512 transfers by edge trigger

enumerator kDMA_EdgeBurstTransfer1024

Perform 1024 transfers by edge trigger

enum `_dma_burst_wrap`

DMA burst wrapping.

Values:

enumerator `kDMA_NoWrap`

Wrapping is disabled

enumerator `kDMA_SrcWrap`

Wrapping is enabled for source

enumerator `kDMA_DstWrap`

Wrapping is enabled for destination

enumerator `kDMA_SrcAndDstWrap`

Wrapping is enabled for source and destination

enum `_dma_transfer_type`

DMA transfer type.

Values:

enumerator `kDMA_MemoryToMemory`

Transfer from memory to memory (increment source and destination)

enumerator `kDMA_PeripheralToMemory`

Transfer from peripheral to memory (increment only destination)

enumerator `kDMA_MemoryToPeripheral`

Transfer from memory to peripheral (increment only source)

enumerator `kDMA_StaticToStatic`

Peripheral to static memory (do not increment source or destination)

typedef struct `_dma_descriptor` `dma_descriptor_t`

DMA descriptor structure.

typedef struct `_dma_xfercfg` `dma_xfercfg_t`

DMA transfer configuration.

typedef enum `_dma_priority` `dma_priority_t`

DMA channel priority.

typedef enum `_dma_int` `dma_irq_t`

DMA interrupt flags.

typedef enum `_dma_trigger_type` `dma_trigger_type_t`

DMA trigger type.

typedef enum `_dma_trigger_burst` `dma_trigger_burst_t`

DMA trigger burst.

typedef enum `_dma_burst_wrap` `dma_burst_wrap_t`

DMA burst wrapping.

typedef enum `_dma_transfer_type` `dma_transfer_type_t`

DMA transfer type.

typedef struct `_dma_channel_trigger` `dma_channel_trigger_t`

DMA channel trigger.

typedef struct `_dma_channel_config` `dma_channel_config_t`

DMA channel trigger.

```
typedef struct _dma_transfer_config dma_transfer_config_t
```

DMA transfer configuration.

```
typedef void (*dma_callback)(struct _dma_handle *handle, void *userData, bool transferDone,
uint32_t intmode)
```

Define Callback function for DMA.

```
typedef struct _dma_handle dma_handle_t
```

DMA transfer handle structure.

```
DMA_MAX_TRANSFER_COUNT
```

DMA max transfer size.

```
FSL_FEATURE_DMA_NUMBER_OF_CHANNELSn(x)
```

DMA channel numbers.

```
FSL_FEATURE_DMA_MAX_CHANNELS
```

```
FSL_FEATURE_DMA_ALL_CHANNELS
```

```
FSL_FEATURE_DMA_LINK_DESCRIPTOR_ALIGN_SIZE
```

DMA head link descriptor table align size.

```
DMA_ALLOCATE_HEAD_DESCRIPTOR(name, number)
```

DMA head descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

```
DMA_ALLOCATE_HEAD_DESCRIPTOR_AT_NONCACHEABLE(name, number)
```

DMA head descriptor table allocate macro at noncacheable section To simplify user interface, this macro will help allocate descriptor memory at noncacheable section, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

```
DMA_ALLOCATE_LINK_DESCRIPTOR(name, number)
```

DMA link descriptor table allocate macro To simplify user interface, this macro will help allocate descriptor memory, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

```
DMA_ALLOCATE_LINK_DESCRIPTOR_AT_NONCACHEABLE(name, number)
```

DMA link descriptor table allocate macro at noncacheable section To simplify user interface, this macro will help allocate descriptor memory at noncacheable section, user just need to provide the name and the number for the allocate descriptor.

Parameters

- name – Allocate decriptor name.
- number – Number of descriptor to be allocated.

DMA_ALLOCATE_DATA_TRANSFER_BUFFER(name, width)

DMA transfer buffer address need to align with the transfer width.

DMA_CHANNEL_GROUP(channel)

DMA_CHANNEL_INDEX(base, channel)

DMA_COMMON_REG_GET(base, channel, reg)

DMA linked descriptor address algin size.

DMA_COMMON_CONST_REG_GET(base, channel, reg)

DMA_COMMON_REG_SET(base, channel, reg, value)

DMA_DESCRIPTOR_END_ADDRESS(start, inc, bytes, width)

DMA descriptor end address calculate.

Parameters

- start – start address
- inc – address interleave size
- bytes – transfer bytes
- width – transfer width

DMA_CHANNEL_XFER(reload, clrTrig, intA, intB, width, srcInc, dstInc, bytes)

struct _dma_descriptor

#include <fsl_dma.h> DMA descriptor structure.

Public Members

volatile uint32_t xfercfg

Transfer configuration

void *srcEndAddr

Last source address of DMA transfer

void *dstEndAddr

Last destination address of DMA transfer

void *linkToNextDesc

Address of next DMA descriptor in chain

struct _dma_xfercfg

#include <fsl_dma.h> DMA transfer configuration.

Public Members

bool valid

Descriptor is ready to transfer

bool reload

Reload channel configuration register after current descriptor is exhausted

bool swtrig

Perform software trigger. Transfer if fired when 'valid' is set

bool clrtrig

Clear trigger

bool intA

Raises IRQ when transfer is done and set IRQA status register flag

bool intB

Raises IRQ when transfer is done and set IRQB status register flag

uint8_t byteWidth

Byte width of data to transfer

uint8_t srcInc

Increment source address by 'srcInc' x 'byteWidth'

uint8_t dstInc

Increment destination address by 'dstInc' x 'byteWidth'

uint16_t transferCount

Number of transfers

struct _dma_channel_trigger

#include <fsl_dma.h> DMA channel trigger.

Public Members

dma_trigger_type_t type

Select hardware trigger as edge triggered or level triggered.

dma_trigger_burst_t burst

Select whether hardware triggers cause a single or burst transfer.

dma_burst_wrap_t wrap

Select wrap type, source wrap or dest wrap, or both.

struct _dma_channel_config

#include <fsl_dma.h> DMA channel trigger.

Public Members

void *srcStartAddr

Source data address

void *dstStartAddr

Destination data address

void *nextDesc

Chain custom descriptor

uint32_t xferCfg

channel transfer configurations

dma_channel_trigger_t *trigger

DMA trigger type

bool isPeriph

select the request type

struct _dma_transfer_config

#include <fsl_dma.h> DMA transfer configuration.

Public Members

`uint8_t *srcAddr`
Source data address

`uint8_t *dstAddr`
Destination data address

`uint8_t *nextDesc`
Chain custom descriptor

`dma_xfercfg_t xfercfg`
Transfer options

`bool isPeriph`
DMA transfer is driven by peripheral

`struct _dma_handle`
#include <fsl_dma.h> DMA transfer handle structure.

Public Members

`dma_callback` callback
Callback function. Invoked when transfer of descriptor with interrupt flag finishes

`void *userData`
Callback function parameter

`DMA_Type *base`
DMA peripheral base address

`uint8_t channel`
DMA channel number

2.4 FTM: FlexTimer Driver

`status_t FTM_Init(FTM_Type *base, const ftm_config_t *config)`
Ungates the FTM clock and configures the peripheral for basic operation.

Note: This API should be called at the beginning of the application which is using the FTM driver. If the FTM instance has only TPM features, please use the TPM driver.

Parameters

- `base` – FTM peripheral base address
- `config` – Pointer to the user configuration structure.

Returns

`kStatus_Success` indicates success; Else indicates failure.

`void FTM_Deinit(FTM_Type *base)`
Gates the FTM clock.

Parameters

- `base` – FTM peripheral base address

```
void FTM_GetDefaultConfig(ftm_config_t *config)
```

Fills in the FTM configuration structure with the default settings.

The default values are:

```
config->prescale = kFTM_Prescale_Divide_1;
config->bdmMode = kFTM_BdmMode_0;
config->pwmSyncMode = kFTM_SoftwareTrigger;
config->reloadPoints = 0;
config->faultMode = kFTM_Fault_Disable;
config->faultFilterValue = 0;
config->deadTimePrescale = kFTM_Deadtime_Prescale_1;
config->deadTimeValue = 0;
config->extTriggers = 0;
config->chnlInitState = 0;
config->chnlPolarity = 0;
config->useGlobalTimeBase = false;
config->hwTriggerResetCount = false;
config->swTriggerResetCount = true;
```

Parameters

- config – Pointer to the user configuration structure.

```
static inline ftm_clock_prescale_t FTM_CalculateCounterClkDiv(FTM_Type *base, uint32_t
counterPeriod_Hz, uint32_t
srcClock_Hz)
```

brief Calculates the counter clock prescaler.

This function calculates the values for SC[PS] bit.

param base FTM peripheral base address param counterPeriod_Hz The desired frequency in Hz which corresponding to the time when the counter reaches the mod value param srcClock_Hz FTM counter clock in Hz

return Calculated clock prescaler value, see *ftm_clock_prescale_t*.

```
status_t FTM_SetupPwm(FTM_Type *base, const ftm_chnl_pwm_signal_param_t *chnlParams,
uint8_t numOfChnls, ftm_pwm_mode_t mode, uint32_t pwmFreq_Hz,
uint32_t srcClock_Hz)
```

Configures the PWM signal parameters.

Call this function to configure the PWM signal period, mode, duty cycle, and edge. Use this function to configure all FTM channels that are used to output a PWM signal.

Parameters

- base – FTM peripheral base address
- chnlParams – Array of PWM channel parameters to configure the channel(s)
- numOfChnls – Number of channels to configure; This should be the size of the array passed in
- mode – PWM operation mode, options available in enumeration *ftm_pwm_mode_t*
- pwmFreq_Hz – PWM signal frequency in Hz
- srcClock_Hz – FTM counter clock in Hz

Returns

kStatus_Success if the PWM setup was successful *kStatus_Error* on failure

```
status_t FTM_UpdatePwmDutycycle(FTM_Type *base, ftm_chnl_t chnlNumber, ftm_pwm_mode_t
                                currentPwmMode, uint8_t dutyCyclePercent)
```

Updates the duty cycle of an active PWM signal.

Parameters

- *base* – FTM peripheral base address
- *chnlNumber* – The channel/channel pair number. In combined mode, this represents the channel pair number
- *currentPwmMode* – The current PWM mode set during PWM setup
- *dutyCyclePercent* – New PWM pulse width; The value should be between 0 to 100 0=inactive signal(0% duty cycle)... 100=active signal (100% duty cycle)

Returns

kStatus_Success if the PWM update was successful *kStatus_Error* on failure

```
void FTM_UpdateChnlEdgeLevelSelect(FTM_Type *base, ftm_chnl_t chnlNumber, uint8_t level)
```

Updates the edge level selection for a channel.

Parameters

- *base* – FTM peripheral base address
- *chnlNumber* – The channel number
- *level* – The level to be set to the ELSnB:ELSnA field; Valid values are 00, 01, 10, 11. See the Kinetis SoC reference manual for details about this field.

```
status_t FTM_SetupPwmMode(FTM_Type *base, const ftm_chnl_pwm_config_param_t
                          *chnlParams, uint8_t numOfChnls, ftm_pwm_mode_t mode)
```

Configures the PWM mode parameters.

Call this function to configure the PWM signal mode, duty cycle in ticks, and edge. Use this function to configure all FTM channels that are used to output a PWM signal. Please note that: This API is similar with `FTM_SetupPwm()` API, but will not set the timer period, and this API will set channel match value in timer ticks, not period percent.

Parameters

- *base* – FTM peripheral base address
- *chnlParams* – Array of PWM channel parameters to configure the channel(s)
- *numOfChnls* – Number of channels to configure; This should be the size of the array passed in
- *mode* – PWM operation mode, options available in enumeration `ftm_pwm_mode_t`

Returns

kStatus_Success if the PWM setup was successful *kStatus_Error* on failure

```
void FTM_ConfigSinglePWM(FTM_Type *base, const ftm_chnl_param_t *chnlParams, ftm_chnl_t
                        chnlNumber)
```

Configure FTM edge aligned PWM or center aligned PWM by each channel.

This function configure PWM signal by setting channel n value register. Need to invoke `FTM_SetInitialModuloValue` to configure FTM period.

Parameters

- *base* – FTM peripheral base address
- *chnlParams* – PWM configuration structure pointer.

- `chnlPairNumber` – Channel number.

```
void FTM_ConfigCombinePWM(FTM_Type *base, const ftm_chnl_param_t *chnlParams,
                          ftm_chnl_t chnlPairNumber)
```

Configure FTM Combine PWM, Modified Combine PWM or Asymmetrical PWM by each channel pair.

This function configure PWM signal by setting channel n value register. Need to invoke `FTM_SetInitialModuloValue` to configure FTM period.

Parameters

- `base` – FTM peripheral base address
- `chnlParams` – PWM configuration structure pointer.
- `chnlPairNumber` – Channel pair number, options are 0, 1, 2, 3.

```
void FTM_SetupInputCapture(FTM_Type *base, ftm_chnl_t chnlNumber,
                           ftm_input_capture_edge_t captureMode, uint32_t filterValue)
```

Enables capturing an input signal on the channel using the function parameters.

When the edge specified in the `captureMode` argument occurs on the channel, the FTM counter is captured into the CnV register. The user has to read the CnV register separately to get this value. The filter function is disabled if the `filterVal` argument passed in is 0. The filter function is available only for channels 0, 1, 2, 3.

Parameters

- `base` – FTM peripheral base address
- `chnlNumber` – The channel number
- `captureMode` – Specifies which edge to capture
- `filterValue` – Filter value, specify 0 to disable filter. Available only for channels 0-3.

```
void FTM_SetupOutputCompare(FTM_Type *base, ftm_chnl_t chnlNumber,
                             ftm_output_compare_mode_t compareMode, uint32_t
                             compareValue)
```

Configures the FTM to generate timed pulses.

When the FTM counter matches the value of `compareVal` argument (this is written into CnV reg), the channel output is changed based on what is specified in the `compareMode` argument.

Parameters

- `base` – FTM peripheral base address
- `chnlNumber` – The channel number
- `compareMode` – Action to take on the channel output when the compare condition is met
- `compareValue` – Value to be programmed in the CnV register.

```
void FTM_SetupDualEdgeCapture(FTM_Type *base, ftm_chnl_t chnlPairNumber, const
                              ftm_dual_edge_capture_param_t *edgeParam, uint32_t
                              filterValue)
```

Configures the dual edge capture mode of the FTM.

This function sets up the dual edge capture mode on a channel pair. The capture edge for the channel pair and the capture mode (one-shot or continuous) is specified in the parameter argument. The filter function is disabled if the `filterVal` argument passed is zero. The filter function is available only on channels 0 and 2. The user has to read the channel CnV registers separately to get the capture values.

Parameters

- base – FTM peripheral base address
- chnlPairNumber – The FTM channel pair number; options are 0, 1, 2, 3
- edgeParam – Sets up the dual edge capture function
- filterValue – Filter value, specify 0 to disable filter. Available only for channel pair 0 and 1.

```
void FTM_EnableInterrupts(FTM_Type *base, uint32_t mask)
```

Enables the selected FTM interrupts.

Parameters

- base – FTM peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `ftm_interrupt_enable_t`

```
void FTM_DisableInterrupts(FTM_Type *base, uint32_t mask)
```

Disables the selected FTM interrupts.

Parameters

- base – FTM peripheral base address
- mask – The interrupts to enable. This is a logical OR of members of the enumeration `ftm_interrupt_enable_t`

```
uint32_t FTM_GetEnabledInterrupts(FTM_Type *base)
```

Gets the enabled FTM interrupts.

Parameters

- base – FTM peripheral base address

Returns

The enabled interrupts. This is the logical OR of members of the enumeration `ftm_interrupt_enable_t`

```
uint32_t FTM_GetInstance(FTM_Type *base)
```

Gets the instance from the base address.

Parameters

- base – FTM peripheral base address

Returns

The FTM instance

```
uint32_t FTM_GetStatusFlags(FTM_Type *base)
```

Gets the FTM status flags.

Parameters

- base – FTM peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `ftm_status_flags_t`

```
void FTM_ClearStatusFlags(FTM_Type *base, uint32_t mask)
```

Clears the FTM status flags.

Parameters

- base – FTM peripheral base address

- `mask` – The status flags to clear. This is a logical OR of members of the enumeration `ftm_status_flags_t`

```
static inline void FTM_SetTimerPeriod(FTM_Type *base, uint32_t ticks)
```

Sets the timer period in units of ticks.

Timers counts from 0 until it equals the count value set here. The count value is written to the MOD register.

Note:

- a. This API allows the user to use the FTM module as a timer. Do not mix usage of this API with FTM's PWM setup API's.
 - b. Call the utility macros provided in the `fsl_common.h` to convert usec or msec to ticks.
-

Parameters

- `base` – FTM peripheral base address
- `ticks` – A timer period in units of ticks, which should be equal or greater than 1.

```
static inline void FTM_SetInitialModuloValue(FTM_Type *base, uint16_t initialValue, uint16_t moduloValue)
```

Set initial value and modulo value for FTM.

Parameters

- `base` – FTM peripheral base address
- `initialValue` – FTM counter initial value.
- `moduloValue` – FTM counter modulo value.

```
static inline uint32_t FTM_GetCurrentTimerCount(FTM_Type *base)
```

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: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- `base` – FTM peripheral base address

Returns

The current counter value in ticks

```
static inline void FTM_SetChannelMatchValue(FTM_Type *base, ftm_chnl_t chnlNumber, uint16_t value)
```

Set channel match value for output.

Parameters

- `base` – FTM peripheral base address
- `chnlNumber` – Channel to set.
- `value` – Channel match value for output.

```
static inline uint32_t FTM_GetInputCaptureValue(FTM_Type *base, ftm_chnl_t chnlNumber)
```

Reads the captured value.

This function returns the captured value of a FTM channel configured in input capture or dual edge capture mode.

Note: Call the utility macros provided in the `fsl_common.h` to convert ticks to usec or msec.

Parameters

- `base` – FTM peripheral base address
- `chnlNumber` – Channel to be read

Returns

The captured FTM counter value of the input modes.

```
static inline void FTM_StartTimer(FTM_Type *base, ftm_clock_source_t clockSource)
```

Starts the FTM counter.

Parameters

- `base` – FTM peripheral base address
- `clockSource` – FTM clock source; After the clock source is set, the counter starts running.

```
static inline void FTM_StopTimer(FTM_Type *base)
```

Stops the FTM counter.

Parameters

- `base` – FTM peripheral base address

```
static inline uint32_t FTM_GetSoftwareOutputValue(FTM_Type *base)
```

Get channel software output status.

Parameters

- `base` – FTM peripheral base address

Returns

Status of channel software output, logical OR value of `ftm_channel_index_t`.

```
static inline uint32_t FTM_GetSoftwareOutputEnable(FTM_Type *base)
```

Get channel software enable status.

Parameters

- `base` – FTM peripheral base address

Returns

Status of channel software enable, logical OR value of `ftm_channel_index_t`.

```
static inline void FTM_SetSoftwareOutputCtrl(FTM_Type *base, uint32_t chnlEnable, uint32_t chnlValue)
```

Enables or disables the channel software output control and set channel software output value.

Parameters

- `base` – FTM peripheral base address
- `chnlEnable` – Channels to enable or disable software output control, logical OR of enumeration `ftm_channel_index_t` members.

- `chnlValue` – Channels output value, logical OR of enumeration `ftm_channel_index_t` members

```
static inline void FTM_SetSoftwareCtrlEnable(FTM_Type *base, ftm_chnl_t chnlNumber, bool value)
```

Enables or disables the channel software output control.

Parameters

- `base` – FTM peripheral base address
- `chnlNumber` – Channel to be enabled or disabled
- `value` – `true`: channel output is affected by software output control `false`: channel output is unaffected by software output control

```
static inline void FTM_SetSoftwareCtrlVal(FTM_Type *base, ftm_chnl_t chnlNumber, bool value)
```

Sets the channel software output control value.

Parameters

- `base` – FTM peripheral base address.
- `chnlNumber` – Channel to be configured
- `value` – `true` to set 1, `false` to set 0

```
static inline void FTM_SetFaultControlEnable(FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
```

This function enables/disables the fault control in a channel pair.

Parameters

- `base` – FTM peripheral base address
- `chnlPairNumber` – The FTM channel pair number; options are 0, 1, 2, 3
- `value` – `true`: Enable fault control for this channel pair; `false`: No fault control

```
static inline void FTM_SetDeadTimeEnable(FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
```

This function enables/disables the dead time insertion in a channel pair.

Parameters

- `base` – FTM peripheral base address
- `chnlPairNumber` – The FTM channel pair number; options are 0, 1, 2, 3
- `value` – `true`: Insert dead time in this channel pair; `false`: No dead time inserted

```
static inline void FTM_SetComplementaryEnable(FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
```

This function enables/disables complementary mode in a channel pair.

Parameters

- `base` – FTM peripheral base address
- `chnlPairNumber` – The FTM channel pair number; options are 0, 1, 2, 3
- `value` – `true`: enable complementary mode; `false`: disable complementary mode

```
static inline void FTM_SetInvertEnable(FTM_Type *base, ftm_chnl_t chnlPairNumber, bool value)
```

This function enables/disables inverting control in a channel pair.

Parameters

- base – FTM peripheral base address
- chnlPairNumber – The FTM channel pair number; options are 0, 1, 2, 3
- value – true: enable inverting; false: disable inverting

```
void FTM_SetupQuadDecode(FTM_Type *base, const ftm_phase_params_t *phaseAParams,  
                        const ftm_phase_params_t *phaseBParams,  
                        ftm_quad_decode_mode_t quadMode)
```

Configures the parameters and activates the quadrature decoder mode.

Parameters

- base – FTM peripheral base address
- phaseAParams – Phase A configuration parameters
- phaseBParams – Phase B configuration parameters
- quadMode – Selects encoding mode used in quadrature decoder mode

```
static inline uint32_t FTM_GetQuadDecoderFlags(FTM_Type *base)
```

Gets the FTM Quad Decoder flags.

Parameters

- base – FTM peripheral base address.

Returns

Flag mask of FTM Quad Decoder, see `_ftm_quad_decoder_flags`.

```
static inline void FTM_SetQuadDecoderModuloValue(FTM_Type *base, uint32_t startValue,  
                                                uint32_t overValue)
```

Sets the modulo values for Quad Decoder.

The modulo values configure the minimum and maximum values that the Quad decoder counter can reach. After the counter goes over, the counter value goes to the other side and decrease/increase again.

Parameters

- base – FTM peripheral base address.
- startValue – The low limit value for Quad Decoder counter.
- overValue – The high limit value for Quad Decoder counter.

```
static inline uint32_t FTM_GetQuadDecoderCounterValue(FTM_Type *base)
```

Gets the current Quad Decoder counter value.

Parameters

- base – FTM peripheral base address.

Returns

Current quad Decoder counter value.

```
static inline void FTM_ClearQuadDecoderCounterValue(FTM_Type *base)
```

Clears the current Quad Decoder counter value.

The counter is set as the initial value.

Parameters

- base – FTM peripheral base address.

```
FSL_FTM_DRIVER_VERSION
```

FTM driver version 2.7.5.

enum _ftm_chnl

List of FTM channels.

Note: Actual number of available channels is SoC dependent

Values:

enumerator kFTM_Chnl_0
FTM channel number 0

enumerator kFTM_Chnl_1
FTM channel number 1

enumerator kFTM_Chnl_2
FTM channel number 2

enumerator kFTM_Chnl_3
FTM channel number 3

enumerator kFTM_Chnl_4
FTM channel number 4

enumerator kFTM_Chnl_5
FTM channel number 5

enumerator kFTM_Chnl_6
FTM channel number 6

enumerator kFTM_Chnl_7
FTM channel number 7

enum _ftm_fault_input

List of FTM faults.

Values:

enumerator kFTM_Fault_0
FTM fault 0 input pin

enumerator kFTM_Fault_1
FTM fault 1 input pin

enumerator kFTM_Fault_2
FTM fault 2 input pin

enumerator kFTM_Fault_3
FTM fault 3 input pin

enum _ftm_pwm_mode

FTM PWM operation modes.

Values:

enumerator kFTM_EdgeAlignedPwm
Edge-aligned PWM

enumerator kFTM_CenterAlignedPwm
Center-aligned PWM

enumerator kFTM_EdgeAlignedCombinedPwm
Edge-aligned combined PWM

enumerator kFTM_CenterAlignedCombinedPwm
Center-aligned combined PWM

enumerator kFTM_AsymmetricalCombinedPwm
Asymmetrical combined PWM

enum _ftm_pwm_level_select

FTM PWM output pulse mode: high-true, low-true or no output.

Note: kFTM_NoPwmSignal: ELSnB:ELSnA = 0:0 kFTM_LowTrue: ELSnB:ELSnA = 0:1
EPWM: Channel n output is forced low at counter overflow, forced high at channel n match.
CPWM: Channel n output is forced low at channel n match when counting down, and forced high at channel n match when counting up. Combined PWM: Channel n output is forced high at beginning of period and at channel n+1 match. It is forced low at the channel n match. kFTM_HighTrue: ELSnB:ELSnA = 1:0 EPWM: Channel n output is forced high at counter overflow, forced low at channel n match. CPWM: Channel n output is forced high at channel n match when counting down, and forced low at channel n match when counting up. Combined PWM: Channel n output is forced low at beginning of period and at channel n+1 match. It is forced high at the channel n match.

Values:

enumerator kFTM_NoPwmSignal
No PWM output on pin

enumerator kFTM_LowTrue
Low true pulses

enumerator kFTM_HighTrue
High true pulses

enum _ftm_output_compare_mode

FlexTimer output compare mode.

Values:

enumerator kFTM_NoOutputSignal
No channel output when counter reaches CnV

enumerator kFTM_ToggleOnMatch
Toggle output

enumerator kFTM_ClearOnMatch
Clear output

enumerator kFTM_SetOnMatch
Set output

enum _ftm_input_capture_edge

FlexTimer input capture edge.

Values:

enumerator kFTM_RisingEdge
Capture on rising edge only

enumerator kFTM_FallingEdge
Capture on falling edge only

enumerator kFTM_RiseAndFallEdge
Capture on rising or falling edge

enum `_ftm_dual_edge_capture_mode`
FlexTimer dual edge capture modes.

Values:

enumerator `kFTM_OneShot`
One-shot capture mode

enumerator `kFTM_Continuous`
Continuous capture mode

enum `_ftm_quad_decode_mode`
FlexTimer quadrature decode modes.

Values:

enumerator `kFTM_QuadPhaseEncode`
Phase A and Phase B encoding mode

enumerator `kFTM_QuadCountAndDir`
Count and direction encoding mode

enum `_ftm_phase_polarity`
FlexTimer quadrature phase polarities.

Values:

enumerator `kFTM_QuadPhaseNormal`
Phase input signal is not inverted

enumerator `kFTM_QuadPhaseInvert`
Phase input signal is inverted

enum `_ftm_deadtime_prescale`
FlexTimer pre-scaler factor for the dead time insertion.

Values:

enumerator `kFTM_Deadtime_Prescale_1`
Divide by 1

enumerator `kFTM_Deadtime_Prescale_4`
Divide by 4

enumerator `kFTM_Deadtime_Prescale_16`
Divide by 16

enum `_ftm_clock_source`
FlexTimer clock source selection.

Values:

enumerator `kFTM_SystemClock`
System clock selected

enumerator `kFTM_FixedClock`
Fixed frequency clock

enumerator `kFTM_ExternalClock`
External clock

enum `_ftm_clock_prescale`
FlexTimer pre-scaler factor selection for the clock source.

Values:

enumerator kFTM_Prescale_Divide_1

Divide by 1

enumerator kFTM_Prescale_Divide_2

Divide by 2

enumerator kFTM_Prescale_Divide_4

Divide by 4

enumerator kFTM_Prescale_Divide_8

Divide by 8

enumerator kFTM_Prescale_Divide_16

Divide by 16

enumerator kFTM_Prescale_Divide_32

Divide by 32

enumerator kFTM_Prescale_Divide_64

Divide by 64

enumerator kFTM_Prescale_Divide_128

Divide by 128

enum _ftm_bdm_mode

Options for the FlexTimer behaviour in BDM Mode.

Values:

enumerator kFTM_BdmMode_0

FTM counter stopped, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers

enumerator kFTM_BdmMode_1

FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are forced to their safe value , writes to MOD,CNTIN and C(n)V registers bypass the register buffers

enumerator kFTM_BdmMode_2

FTM counter stopped, CH(n)F bit is not set, FTM channels outputs are frozen when chip enters in BDM mode, writes to MOD,CNTIN and C(n)V registers bypass the register buffers

enumerator kFTM_BdmMode_3

FTM counter in functional mode, CH(n)F bit can be set, FTM channels in functional mode, writes to MOD,CNTIN and C(n)V registers is in fully functional mode

enum _ftm_fault_mode

Options for the FTM fault control mode.

Values:

enumerator kFTM_Fault_Disable

Fault control is disabled for all channels

enumerator kFTM_Fault_EvenChnls

Enabled for even channels only(0,2,4,6) with manual fault clearing

enumerator kFTM_Fault_AllChnlsMan

Enabled for all channels with manual fault clearing

enumerator kFTM_Fault_AllChnlsAuto

Enabled for all channels with automatic fault clearing

enum _ftm_external_trigger

FTM external trigger options.

Note: Actual available external trigger sources are SoC-specific

Values:

enumerator kFTM_Chnl0Trigger

Generate trigger when counter equals chnl 0 CnV reg

enumerator kFTM_Chnl1Trigger

Generate trigger when counter equals chnl 1 CnV reg

enumerator kFTM_Chnl2Trigger

Generate trigger when counter equals chnl 2 CnV reg

enumerator kFTM_Chnl3Trigger

Generate trigger when counter equals chnl 3 CnV reg

enumerator kFTM_Chnl4Trigger

Generate trigger when counter equals chnl 4 CnV reg

enumerator kFTM_Chnl5Trigger

Generate trigger when counter equals chnl 5 CnV reg

enumerator kFTM_Chnl6Trigger

Available on certain SoC's, generate trigger when counter equals chnl 6 CnV reg

enumerator kFTM_Chnl7Trigger

Available on certain SoC's, generate trigger when counter equals chnl 7 CnV reg

enumerator kFTM_InitTrigger

Generate Trigger when counter is updated with CNTIN

enumerator kFTM_ReloadInitTrigger

Available on certain SoC's, trigger on reload point

enum _ftm_pwm_sync_method

FlexTimer PWM sync options to update registers with buffer.

Values:

enumerator kFTM_SoftwareTrigger

Software triggers PWM sync

enumerator kFTM_HardwareTrigger_0

Hardware trigger 0 causes PWM sync

enumerator kFTM_HardwareTrigger_1

Hardware trigger 1 causes PWM sync

enumerator kFTM_HardwareTrigger_2

Hardware trigger 2 causes PWM sync

enum _ftm_reload_point

FTM options available as loading point for register reload.

Note: Actual available reload points are SoC-specific

Values:

enumerator kFTM_Chnl0Match

Channel 0 match included as a reload point

enumerator kFTM_Chnl1Match

Channel 1 match included as a reload point

enumerator kFTM_Chnl2Match

Channel 2 match included as a reload point

enumerator kFTM_Chnl3Match

Channel 3 match included as a reload point

enumerator kFTM_Chnl4Match

Channel 4 match included as a reload point

enumerator kFTM_Chnl5Match

Channel 5 match included as a reload point

enumerator kFTM_Chnl6Match

Channel 6 match included as a reload point

enumerator kFTM_Chnl7Match

Channel 7 match included as a reload point

enumerator kFTM_CntMax

Use in up-down count mode only, reload when counter reaches the maximum value

enumerator kFTM_CntMin

Use in up-down count mode only, reload when counter reaches the minimum value

enumerator kFTM_HalfCycMatch

Available on certain SoC's, half cycle match reload point

enum _ftm_interrupt_enable

List of FTM interrupts.

Note: Actual available interrupts are SoC-specific

Values:

enumerator kFTM_Chnl0InterruptEnable

Channel 0 interrupt

enumerator kFTM_Chnl1InterruptEnable

Channel 1 interrupt

enumerator kFTM_Chnl2InterruptEnable

Channel 2 interrupt

enumerator kFTM_Chnl3InterruptEnable

Channel 3 interrupt

enumerator kFTM_Chnl4InterruptEnable

Channel 4 interrupt

enumerator kFTM_Chnl5InterruptEnable

Channel 5 interrupt

enumerator kFTM_Chnl6InterruptEnable

Channel 6 interrupt

enumerator kFTM_Chnl7InterruptEnable
Channel 7 interrupt

enumerator kFTM_FaultInterruptEnable
Fault interrupt

enumerator kFTM_TimeOverflowInterruptEnable
Time overflow interrupt

enumerator kFTM_ReloadInterruptEnable
Reload interrupt; Available only on certain SoC's

enum _ftm_status_flags
List of FTM flags.

Note: Actual available flags are SoC-specific

Values:

enumerator kFTM_Chnl0Flag
Channel 0 Flag

enumerator kFTM_Chnl1Flag
Channel 1 Flag

enumerator kFTM_Chnl2Flag
Channel 2 Flag

enumerator kFTM_Chnl3Flag
Channel 3 Flag

enumerator kFTM_Chnl4Flag
Channel 4 Flag

enumerator kFTM_Chnl5Flag
Channel 5 Flag

enumerator kFTM_Chnl6Flag
Channel 6 Flag

enumerator kFTM_Chnl7Flag
Channel 7 Flag

enumerator kFTM_FaultFlag
Fault Flag

enumerator kFTM_TimeOverflowFlag
Time overflow Flag

enumerator kFTM_ChnlTriggerFlag
Channel trigger Flag

enumerator kFTM_ReloadFlag
Reload Flag; Available only on certain SoC's

enum _ftm_channel_index
List of FTM channel index used in logic OR.

Values:

enumerator kFTM_Chnl0_Mask
Channel 0 Mask

enumerator kFTM_Chnl1_Mask
Channel 1 Mask

enumerator kFTM_Chnl2_Mask
Channel 2 Mask

enumerator kFTM_Chnl3_Mask
Channel 3 Mask

enumerator kFTM_Chnl4_Mask
Channel 4 Mask

enumerator kFTM_Chnl5_Mask
Channel 5 Mask

enumerator kFTM_Chnl6_Mask
Channel 6 Mask

enumerator kFTM_Chnl7_Mask
Channel 7 Mask

List of FTM Quad Decoder flags.

Values:

enumerator kFTM_QuadDecoderCountingIncreaseFlag
Counting direction is increasing (FTM counter increment), or the direction is decreasing.

enumerator kFTM_QuadDecoderCountingOverflowOnTopFlag
Indicates if the TOF bit was set on the top or the bottom of counting.

typedef enum *_ftm_chnl* ftm_chnl_t
List of FTM channels.

Note: Actual number of available channels is SoC dependent

typedef enum *_ftm_fault_input* ftm_fault_input_t
List of FTM faults.

typedef enum *_ftm_pwm_mode* ftm_pwm_mode_t
FTM PWM operation modes.

typedef enum *_ftm_pwm_level_select* ftm_pwm_level_select_t
FTM PWM output pulse mode: high-true, low-true or no output.

Note: kFTM_NoPwmSignal: ELSnB:ELSnA = 0:0 kFTM_LowTrue: ELSnB:ELSnA = 0:1
EPWM: Channel n output is forced low at counter overflow, forced high at channel n match.
CPWM: Channel n output is forced low at channel n match when counting down, and forced high at channel n match when counting up. Combined PWM: Channel n output is forced high at beginning of period and at channel n+1 match. It is forced low at the channel n match. kFTM_HighTrue: ELSnB:ELSnA = 1:0 EPWM: Channel n output is forced high at counter overflow, forced low at channel n match. CPWM: Channel n output is forced high at channel n match when counting down, and forced low at channel n match when counting up. Combined PWM: Channel n output is forced low at beginning of period and at channel n+1 match. It is forced high at the channel n match.

```
typedef struct _ftm_chnl_pwm_signal_param ftm_chnl_pwm_signal_param_t
    Options to configure a FTM channel's PWM signal.
typedef struct _ftm_chnl_pwm_config_param ftm_chnl_pwm_config_param_t
    Options to configure a FTM channel using precise setting.
typedef struct _ftm_chnl_param ftm_chnl_param_t
    General options to configure a FTM channel using precise setting.
typedef enum _ftm_output_compare_mode ftm_output_compare_mode_t
    FlexTimer output compare mode.
typedef enum _ftm_input_capture_edge ftm_input_capture_edge_t
    FlexTimer input capture edge.
typedef enum _ftm_dual_edge_capture_mode ftm_dual_edge_capture_mode_t
    FlexTimer dual edge capture modes.
typedef struct _ftm_dual_edge_capture_param ftm_dual_edge_capture_param_t
    FlexTimer dual edge capture parameters.
typedef enum _ftm_quad_decode_mode ftm_quad_decode_mode_t
    FlexTimer quadrature decode modes.
typedef enum _ftm_phase_polarity ftm_phase_polarity_t
    FlexTimer quadrature phase polarities.
typedef struct _ftm_phase_param ftm_phase_params_t
    FlexTimer quadrature decode phase parameters.
typedef struct _ftm_fault_param ftm_fault_param_t
    Structure is used to hold the parameters to configure a FTM fault.
typedef enum _ftm_deadtime_prescale ftm_deadtime_prescale_t
    FlexTimer pre-scaler factor for the dead time insertion.
typedef enum _ftm_clock_source ftm_clock_source_t
    FlexTimer clock source selection.
typedef enum _ftm_clock_prescale ftm_clock_prescale_t
    FlexTimer pre-scaler factor selection for the clock source.
typedef enum _ftm_bdm_mode ftm_bdm_mode_t
    Options for the FlexTimer behaviour in BDM Mode.
typedef enum _ftm_fault_mode ftm_fault_mode_t
    Options for the FTM fault control mode.
typedef enum _ftm_external_trigger ftm_external_trigger_t
    FTM external trigger options.
```

Note: Actual available external trigger sources are SoC-specific

```
typedef enum _ftm_pwm_sync_method ftm_pwm_sync_method_t
    FlexTimer PWM sync options to update registers with buffer.
typedef enum _ftm_reload_point ftm_reload_point_t
    FTM options available as loading point for register reload.
```

Note: Actual available reload points are SoC-specific

typedef enum *_ftm_interrupt_enable* ftm_interrupt_enable_t
List of FTM interrupts.

Note: Actual available interrupts are SoC-specific

typedef enum *_ftm_status_flags* ftm_status_flags_t
List of FTM flags.

Note: Actual available flags are SoC-specific

typedef enum *_ftm_channel_index* ftm_channel_index_t
List of FTM channel index used in logic OR.

typedef struct *_ftm_config* ftm_config_t
FTM configuration structure.

This structure holds the configuration settings for the FTM peripheral. To initialize this structure to reasonable defaults, call the FTM_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 FTM_SetupFaultInput(FTM_Type *base, *ftm_fault_input_t* faultNumber, const
ftm_fault_param_t *faultParams)

Sets up the working of the FTM fault inputs protection.

FTM can have up to 4 fault inputs. This function sets up fault parameters, fault level, and input filter.

Parameters

- base – FTM peripheral base address
- faultNumber – FTM fault to configure.
- faultParams – Parameters passed in to set up the fault

static inline void FTM_SetGlobalTimeBaseOutputEnable(FTM_Type *base, bool enable)
Enables or disables the FTM global time base signal generation to other FTMs.

Parameters

- base – FTM peripheral base address
- enable – true to enable, false to disable

static inline void FTM_SetOutputMask(FTM_Type *base, *ftm_chnl_t* chnlNumber, bool mask)
Sets the FTM peripheral timer channel output mask.

Parameters

- base – FTM peripheral base address
- chnlNumber – Channel to be configured
- mask – true: masked, channel is forced to its inactive state; false: un-masked

static inline void FTM_SetPwmOutputEnable(FTM_Type *base, *ftm_chnl_t* chnlNumber, bool
value)

Allows users to enable an output on an FTM channel.

To enable the PWM channel output call this function with val=true. For input mode, call this function with val=false.

Parameters

- base – FTM peripheral base address
- chnlNumber – Channel to be configured
- value – true: enable output; false: output is disabled, used in input mode

static inline void FTM_SetSoftwareTrigger(FTM_Type *base, bool enable)

Enables or disables the FTM software trigger for PWM synchronization.

Parameters

- base – FTM peripheral base address
- enable – true: software trigger is selected, false: software trigger is not selected

static inline void FTM_SetWriteProtection(FTM_Type *base, bool enable)

Enables or disables the FTM write protection.

Parameters

- base – FTM peripheral base address
- enable – true: Write-protection is enabled, false: Write-protection is disabled

static inline void FTM_EnableDmaTransfer(FTM_Type *base, *ftm_chnl_t* chnlNumber, bool enable)

Enable DMA transfer or not.

Note: CHnIE bit needs to be set when calling this API. The channel DMA transfer request is generated and the channel interrupt is not generated if (CHnF = 1) when DMA and CHnIE bits are set.

Parameters

- base – FTM peripheral base address.
- chnlNumber – Channel to be configured
- enable – true to enable, false to disable

static inline void FTM_SetLdok(FTM_Type *base, bool value)

Enable the LDOK bit.

This function enables loading updated values.

Parameters

- base – FTM peripheral base address
- value – true: loading updated values is enabled; false: loading updated values is disabled.

static inline void FTM_SetHalfCycReloadMatchValue(FTM_Type *base, uint32_t ticks)

Sets the half cycle relade period in units of ticks.

This function can be called to set the half-cycle reload value when half-cycle matching is enabled as a reload point. Note: Need enable kFTM_HalfCycMatch as reload point, and when this API call after FTM_StartTimer(), the new HCR value will not be active until next reload point (need call FTM_SetLdok to set LDOK) or register synchronization.

Parameters

- base – FTM peripheral base address
- ticks – A timer period in units of ticks, which should be equal or greater than 1.

```
static inline void FTM_SetLoadFreq(FTM_Type *base, uint32_t loadfreq)
```

Set load frequency value.

Parameters

- base – FTM peripheral base address.
- loadfreq – PWM reload frequency, range: 0 ~ 31.

```
struct _ftm_chnl_pwm_signal_param
```

#include <fsl_ftm.h> Options to configure a FTM channel's PWM signal.

Public Members

ftm_chnl_t chnlNumber

The channel/channel pair number. In combined mode, this represents the channel pair number.

ftm_pwm_level_select_t level

PWM output active level select.

uint8_t dutyCyclePercent

PWM pulse width, value should be between 0 to 100 0 = inactive signal(0% duty cycle)... 100 = always active signal (100% duty cycle).

uint8_t firstEdgeDelayPercent

Used only in kFTM_AsymmetricalCombinedPwm mode to generate an asymmetrical PWM. Specifies the delay to the first edge in a PWM period. If unsure leave as 0; Should be specified as a percentage of the PWM period

bool enableComplementary

Used only in combined PWM mode. true: The combined channels output complementary signals; false: The combined channels output same signals;

bool enableDeadtime

Used only in combined PWM mode with enable complementary. true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

```
struct _ftm_chnl_pwm_config_param
```

#include <fsl_ftm.h> Options to configure a FTM channel using precise setting.

Public Members

ftm_chnl_t chnlNumber

The channel/channel pair number. In combined mode, this represents the channel pair number.

ftm_pwm_level_select_t level

PWM output active level select.

uint16_t dutyValue

PWM pulse width, the uint of this value is timer ticks.

uint16_t firstEdgeValue

Used only in kFTM_AsymmetricalCombinedPwm mode to generate an asymmetrical PWM. Specifies the delay to the first edge in a PWM period. If unsure leave as 0, uint of this value is timer ticks.

bool enableComplementary

Used only in combined PWM mode. true: The combined channels output complementary signals; false: The combined channels output same signals;

bool enableDeadtime

Used only in combined PWM mode with enable complementary. true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

struct `_ftm_chnl_param`

#include <fsl_ftm.h> General options to configure a FTM channel using precise setting.

Public Members

ftm_pwm_mode_t mode

PWM output mode.

ftm_pwm_level_select_t level

PWM output active level select.

uint16_t initialValue

FTM counter initial value.

uint16_t moduloValue

FTM counter modulo value.

uint16_t chnlValue

FTM channel n match value.

uint16_t combinedChnlValue

FTM combined channel n+1 match value, used only in (modified) combined PWM mode.

bool enableComplementary

Used only in combined PWM mode. true: The combined channels output complementary signals; false: The combined channels output same signals;

bool enableDeadtime

Used only in combined PWM mode with enable complementary. true: The deadtime insertion in this pair of channels is enabled; false: The deadtime insertion in this pair of channels is disabled.

struct `_ftm_dual_edge_capture_param`

#include <fsl_ftm.h> FlexTimer dual edge capture parameters.

Public Members

ftm_dual_edge_capture_mode_t mode

Dual Edge Capture mode

ftm_input_capture_edge_t currChanEdgeMode

Input capture edge select for channel n

ftm_input_capture_edge_t nextChanEdgeMode

Input capture edge select for channel n+1

struct `_ftm_phase_param`

#include <fsl_ftm.h> FlexTimer quadrature decode phase parameters.

Public Members

`bool enablePhaseFilter`
True: enable phase filter; false: disable filter

`uint32_t phaseFilterVal`
Filter value, used only if phase filter is enabled

`ftm_phase_polarity_t phasePolarity`
Phase polarity

`struct _ftm_fault_param`
#include <fsl_ftm.h> Structure is used to hold the parameters to configure a FTM fault.

Public Members

`bool enableFaultInput`
True: Fault input is enabled; false: Fault input is disabled

`bool faultLevel`
True: Fault polarity is active low; in other words, '0' indicates a fault; False: Fault polarity is active high

`bool useFaultFilter`
True: Use the filtered fault signal; False: Use the direct path from fault input

`struct _ftm_config`
#include <fsl_ftm.h> FTM configuration structure.

This structure holds the configuration settings for the FTM peripheral. To initialize this structure to reasonable defaults, call the `FTM_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

`ftm_clock_prescale_t prescale`
FTM clock prescale value

`ftm_bdm_mode_t bdmMode`
FTM behavior in BDM mode

`uint32_t pwmSyncMode`
Synchronization methods to use to update buffered registers; Multiple update modes can be used by providing an OR'ed list of options available in enumeration `ftm_pwm_sync_method_t`.

`uint32_t reloadPoints`
FTM reload points; When using this, the PWM synchronization is not required. Multiple reload points can be used by providing an OR'ed list of options available in enumeration `ftm_reload_point_t`.

`ftm_fault_mode_t faultMode`
FTM fault control mode

`uint8_t faultFilterValue`
Fault input filter value

ftm_deadtime_prescale_t deadTimePrescale

The dead time prescalar value

uint32_t deadTimeValue

The dead time value deadTimeValue's available range is 0-1023 when register has DT-VALEX, otherwise its available range is 0-63.

uint32_t extTriggers

External triggers to enable. Multiple trigger sources can be enabled by providing an OR'ed list of options available in enumeration *ftm_external_trigger_t*.

uint8_t chnlInitState

Defines the initialization value of the channels in OUTINT register

uint8_t chnlPolarity

Defines the output polarity of the channels in POL register

bool useGlobalTimeBase

True: Use of an external global time base is enabled; False: disabled

bool swTriggerResetCount

FTM counter synchronization activated by software trigger, active when (*syncMethod* & *FTM_SYNC_SWSYNC_MASK*) != 0U

bool hwTriggerResetCount

FTM counter synchronization activated by hardware trigger, active when (*syncMethod* & (*FTM_SYNC_TRIG0_MASK* | *FTM_SYNC_TRIG1_MASK* | *FTM_SYNC_TRIG2_MASK*)) != 0U

2.5 I2C: Inter-Integrated Circuit Driver

2.6 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`

2.7 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.

2.8 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_SaveDeselected`

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 <fsl_i2c.h> I2C slave transfer structure.

Public Members

`i2c_slave_handle_t *handle`
 Pointer to handle that contains this transfer.

i2c_slave_transfer_event_t event

Reason the callback is being invoked.

uint8_t receivedAddress

Matching address send by master. 7-bits plus R/nW bit0

uint32_t eventMask

Mask of enabled events.

uint8_t *rxData

Transfer buffer for receive data

const uint8_t *txData

Transfer buffer for transmit data

size_t txSize

Transfer size

size_t rxSize

Transfer size

size_t transferredCount

Number of bytes transferred during this transfer.

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for I2C_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.

2.9 I3C: I3C Driver

FSL_I3C_DRIVER_VERSION

I3C driver version.

I3C status return codes.

Values:

enumerator kStatus_I3C_Busy

The master is already performing a transfer.

enumerator kStatus_I3C_Idle

The slave driver is idle.

enumerator kStatus_I3C_Nak

The slave device sent a NAK in response to an address.

enumerator kStatus_I3C_WriteAbort

The slave device sent a NAK in response to a write.

enumerator kStatus_I3C_Term

The master terminates slave read.

enumerator kStatus_I3C_HdrParityError

Parity error from DDR read.

enumerator kStatus_I3C_CrcError

CRC error from DDR read.

enumerator kStatus_I3C_ReadFifoError

Read from M/SRDATA register when FIFO empty.

enumerator kStatus_I3C_WriteFifoError

Write to M/SWDATA register when FIFO full.

enumerator kStatus_I3C_MsgError

Message SDR/DDR mismatch or read/write message in wrong state

enumerator kStatus_I3C_InvalidReq

Invalid use of request.

enumerator kStatus_I3C_Timeout

The module has stalled too long in a frame.

enumerator kStatus_I3C_SlaveCountExceed

The I3C slave count has exceeded the definition in I3C_MAX_DEVCNT.

enumerator kStatus_I3C_IBIWon

The I3C slave event IBI or MR or HJ won the arbitration on a header address.

enumerator kStatus_I3C_OverrunError

Slave internal from-bus buffer/FIFO overrun.

enumerator kStatus_I3C_UnderrunError

Slave internal to-bus buffer/FIFO underrun

enumerator kStatus_I3C_UnderrunNak

Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kStatus_I3C_InvalidStart

Slave invalid start flag

enumerator kStatus_I3C_SdrParityError

SDR parity error

enumerator kStatus_I3C_S0S1Error
 S0 or S1 error

enum `_i3c_hdr_mode`
 I3C HDR modes.

Values:

enumerator kI3C_HDRModeNone

enumerator kI3C_HDRModeDDR

enumerator kI3C_HDRModeTSP

enumerator kI3C_HDRModeTSL

typedef enum `_i3c_hdr_mode` `i3c_hdr_mode_t`
 I3C HDR modes.

typedef struct `_i3c_device_info` `i3c_device_info_t`
 I3C device information.

I3C_RETRY_TIMES
 Max loops to wait for I3C operation status complete.

This is the maximum number of loops to wait for I3C operation status complete. If set to 0, it will wait indefinitely.

I3C_MAX_DEVCNT

I3C_IBI_BUFF_SIZE

struct `_i3c_device_info`
#include <fsl_i3c.h> I3C device information.

Public Members

uint8_t `dynamicAddr`
 Device dynamic address.

uint8_t `staticAddr`
 Static address.

uint8_t `dcr`
 Device characteristics register information.

uint8_t `bcr`
 Bus characteristics register information.

uint16_t `vendorID`
 Device vendor ID(manufacture ID).

uint32_t `partNumber`
 Device part number info

uint16_t `maxReadLength`
 Maximum read length.

uint16_t `maxWriteLength`
 Maximum write length.

uint8_t `hdrMode`
 Support hdr mode, could be OR logic in `i3c_hdr_mode`.

2.10 I3C Common Driver

```
typedef struct i3c_config i3c_config_t
```

Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

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

```
uint32_t I3C_GetInstance(I3C_Type *base)
```

Get which instance current I3C is used.

Parameters

- `base` – The I3C peripheral base address.

```
void I3C_GetDefaultConfig(i3c_config_t *config)
```

Provides a default configuration for the I3C peripheral, the configuration covers both master functionality and slave functionality.

This function provides the following default configuration for I3C:

```
config->enableMaster          = kI3C_MasterCapable;
config->disableTimeout        = false;
config->hKeep                 = kI3C_MasterHighKeeperNone;
config->enableOpenDrainStop    = true;
config->enableOpenDrainHigh    = true;
config->baudRate_Hz.i2cBaud    = 400000U;
config->baudRate_Hz.i3cPushPullBaud = 12500000U;
config->baudRate_Hz.i3cOpenDrainBaud = 2500000U;
config->masterDynamicAddress    = 0x0AU;
config->slowClock_Hz           = 1000000U;
config->enableSlave            = true;
config->vendorID                = 0x11BU;
config->enableRandomPart        = false;
config->partNumber              = 0;
config->dcr                     = 0;
config->bcr = 0;
config->hdrMode                 = (uint8_t)kI3C_HDRModeDDR;
config->nakAllRequest           = false;
config->ignoreS0S1Error        = false;
config->offline                 = false;
config->matchSlaveStartStop    = false;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the common I3C driver with `I3C_Init()`.

Parameters

- `config` – **[out]** User provided configuration structure for default values. Refer to `i3c_config_t`.

```
void I3C_Init(I3C_Type *base, const i3c_config_t *config, uint32_t sourceClock_Hz)
```

Initializes the I3C peripheral. This function enables the peripheral clock and initializes the I3C peripheral as described by the user provided configuration. This will initialize both the master peripheral and slave peripheral so that I3C module could work as pure master, pure slave or secondary master, etc. A software reset is performed prior to configuration.

Parameters

- `base` – The I3C peripheral base address.

- `config` – User provided peripheral configuration. Use `I3C_GetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

`struct __i3c_config`

`#include <fsl_i3c.h>` Structure with settings to initialize the I3C module, could both initialize master and slave functionality.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_GetDefaultConfig()` function and pass a pointer to your configuration structure instance.

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

Public Members

`i3c_master_enable_t` enableMaster

Enable master mode.

`bool` disableTimeout

Whether to disable timeout to prevent the ERRWARN.

`i3c_master_hkeep_t` hKeep

High keeper mode setting.

`bool` enableOpenDrainStop

Whether to emit open-drain speed STOP.

`bool` enableOpenDrainHigh

Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

`i3c_baudrate_hz_t` baudRate_Hz

Desired baud rate settings.

`i3c_start_scl_delay_t` startSclDelay

I3C SCL delay after START.

`i3c_start_scl_delay_t` restartSclDelay

I3C SCL delay after Repeated START.

`uint8_t` masterDynamicAddress

Main master dynamic address configuration.

`uint32_t` maxWriteLength

Maximum write length.

`uint32_t` maxReadLength

Maximum read length.

`bool` enableSlave

Whether to enable slave.

`uint8_t` staticAddr

Static address.

`uint16_t` vendorID

Device vendor ID(manufacture ID).

`uint32_t` partNumber

Device part number info

- `uint8_t dcr`
Device characteristics register information.
- `uint8_t bcr`
Bus characteristics register information.
- `uint8_t hdrMode`
Support hdr mode, could be OR logic in enumeration:`i3c_hdr_mode_t`.
- `bool nakAllRequest`
Whether to reply NAK to all requests except broadcast CCC.
- `bool ignoreS0S1Error`
Whether to ignore S0/S1 error in SDR mode.
- `bool offline`
Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.
- `bool matchSlaveStartStop`
Whether to assert start/stop status only the time slave is addressed.

2.11 I3C Master DMA Driver

```
typedef struct i3c_master_dma_handle i3c_master_dma_handle_t
```

```
typedef struct i3c_master_dma_callback i3c_master_dma_callback_t
    i3c master callback functions.
```

```
void I3C_MasterTransferCreateHandleDMA(I3C_Type *base, i3c_master_dma_handle_t *handle,
    const i3c_master_dma_callback_t *callback, void
    *userData, dma_handle_t *rxDmaHandle,
    dma_handle_t *txDmaHandle)
```

Create a new handle for the I3C master DMA APIs.

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_MasterTransferAbortDMA()` API shall be called.

For devices where the I3C send and receive DMA requests are OR'd together, the *txDmaHandle* parameter is ignored and may be set to NULL.

Parameters

- *base* – The I3C peripheral base address.
- *handle* – Pointer to the I3C master driver handle.
- *callback* – User provided pointer to the asynchronous callback function.
- *userData* – User provided pointer to the application callback data.
- *rxDmaHandle* – Handle for the DMA receive channel. Created by the user prior to calling this function.
- *txDmaHandle* – Handle for the DMA transmit channel. Created by the user prior to calling this function.

```
status_t I3C_MasterTransferDMA(I3C_Type *base, i3c_master_dma_handle_t *handle,
    i3c_master_transfer_t *transfer)
```

Performs a non-blocking DMA-based transaction on the I3C bus.

The callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

- *base* – The I3C peripheral base address.
- *handle* – Pointer to the I3C master driver handle.
- *transfer* – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or another DMA transaction is already in progress.

```
status_t I3C_MasterTransferGetCountDMA(I3C_Type *base, i3c_master_dma_handle_t *handle,  
                                       size_t *count)
```

Returns number of bytes transferred so far.

Parameters

- *base* – The I3C peripheral base address.
- *handle* – Pointer to the I3C master driver handle.
- *count* – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_Success` –
- `kStatus_NoTransferInProgress` – There is not a DMA transaction currently in progress.

```
void I3C_MasterTransferAbortDMA(I3C_Type *base, i3c_master_dma_handle_t *handle)
```

Terminates a non-blocking I3C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the DMA peripheral's IRQ priority.

Parameters

- *base* – The I3C peripheral base address.
- *handle* – Pointer to the I3C master driver handle.

```
void I3C_MasterTransferDMAHandleIRQ(I3C_Type *base, void *i3CHandle)
```

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 I3C peripheral base address.
- *handle* – Pointer to the I3C master DMA driver handle.

void (*slave2Master)(I3C_Type *base, void *userData)
 Transfer complete callback

void (*ibiCallback)(I3C_Type *base, *i3c_master_dma_handle_t* *handle, *i3c_ibi_type_t* ibiType, *i3c_ibi_state_t* ibiState)
 IBI event callback

void (*transferComplete)(I3C_Type *base, *i3c_master_dma_handle_t* *handle, *status_t* status, void *userData)
 Transfer complete callback

I3C_Type *base
 I3C base pointer.

uint8_t state
 Transfer state machine current state.

uint32_t transferCount
 Indicates progress of the transfer

uint8_t subaddressBuffer[4]
 Saving subaddress command.

uint8_t subaddressCount
 Saving command count.

i3c_master_transfer_t transfer
 Copy of the current transfer info.

i3c_master_dma_callback_t callback
 Callback function pointer.

void *userData
 Application data passed to callback.

dma_handle_t *rxDmaHandle
 Handle for receive DMA channel.

dma_handle_t *txDmaHandle
 Handle for transmit DMA channel.

uint8_t ibiAddress
 Slave address which request IBI.

uint8_t *ibiBuff
 Pointer to IBI buffer to keep ibi bytes.

size_t ibiPayloadSize
 IBI payload size.

i3c_ibi_type_t ibiType
 IBI type.

uint32_t transDataSize
 Transferred data size.

uint8_t workaroundBuff[16]
 Workaround buffer to store temporary data.

uint32_t event
 Reason the callback is being invoked.

uint8_t *txData

Transfer buffer

size_t txDataSize

Transfer size

uint8_t *rxData

Transfer buffer

size_t rxDataSize

Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for kI3C_SlaveCompletionEvent.

I3C_Type *base

I3C base pointer.

i3c_slave_dma_transfer_t transfer

I3C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

i3c_slave_dma_callback_t callback

Callback function called at transfer event.

dma_handle_t *rxDmaHandle

Handle for receive DMA channel.

dma_handle_t *txDmaHandle

Handle for transmit DMA channel.

void *userData

Callback parameter passed to callback.

struct _i3c_master_dma_callback

#include <fsl_i3c_dma.h> i3c master callback functions.

struct _i3c_master_dma_handle

#include <fsl_i3c_dma.h> Driver handle for master DMA APIs.

Note: The contents of this structure are private and subject to change.

2.12 I3C Master Driver

void I3C_MasterGetDefaultConfig(*i3c_master_config_t* *masterConfig)

Provides a default configuration for the I3C master peripheral.

This function provides the following default configuration for the I3C master peripheral:

```

masterConfig->enableMaster      = kI3C_MasterOn;
masterConfig->disableTimeout    = false;
masterConfig->hKeep             = kI3C_MasterHighKeeperNone;
masterConfig->enableOpenDrainStop = true;
masterConfig->enableOpenDrainHigh = true;
masterConfig->baudRate_Hz       = 100000U;
masterConfig->busType           = kI3C_TypeI2C;

```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the master driver with `I3C_MasterInit()`.

Parameters

- `masterConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_master_config_t`.

```
void I3C_MasterInit(I3C_Type *base, const i3c_master_config_t *masterConfig, uint32_t
sourceClock_Hz)
```

Initializes the I3C master peripheral.

This function enables the peripheral clock and initializes the I3C master peripheral as described by the user provided configuration. A software reset is performed prior to configuration.

Parameters

- `base` – The I3C peripheral base address.
- `masterConfig` – User provided peripheral configuration. Use `I3C_MasterGetDefaultConfig()` to get a set of defaults that you can override.
- `sourceClock_Hz` – Frequency in Hertz of the I3C functional clock. Used to calculate the baud rate divisors, filter widths, and timeout periods.

```
void I3C_MasterDeinit(I3C_Type *base)
```

Deinitializes the I3C master peripheral.

This function disables the I3C master peripheral and gates the clock. It also performs a software reset to restore the peripheral to reset conditions.

Parameters

- `base` – The I3C peripheral base address.

```
status_t I3C_MasterCheckAndClearError(I3C_Type *base, uint32_t status)
```

```
status_t I3C_MasterWaitForCtrlDone(I3C_Type *base, bool waitIdle)
```

```
status_t I3C_CheckForBusyBus(I3C_Type *base)
```

```
static inline void I3C_MasterEnable(I3C_Type *base, i3c_master_enable_t enable)
```

Set I3C module master mode.

Parameters

- `base` – The I3C peripheral base address.
- `enable` – Enable master mode.

```
void I3C_SlaveGetDefaultConfig(i3c_slave_config_t *slaveConfig)
```

Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableslave      = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with `I3C_SlaveInit()`.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_slave_config_t`.

```
void I3C_SlaveInit(I3C_Type *base, const i3c_slave_config_t *slaveConfig, uint32_t slowClock_Hz)
```

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The I3C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `I3C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `slowClock_Hz` – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If `FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH` defines as 1, this parameter is useless.

```
void I3C_SlaveDeinit(I3C_Type *base)
```

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- `base` – The I3C peripheral base address.

```
static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnabled)
Enable/Disable Slave.
```

Parameters

- `base` – The I3C peripheral base address.
- `isEnabled` – Enable or disable.

```
static inline uint32_t I3C_MasterGetStatusFlags(I3C_Type *base)
Gets the I3C master status flags.
```

A bit mask with the state of all I3C master status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_master_flags`

Parameters

- `base` – The I3C 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 I3C_MasterClearStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C master status flag state.

The following status register flags can be cleared:

- `ki3c_MasterSlaveStartFlag`
- `ki3c_MasterControlDoneFlag`
- `ki3c_MasterCompleteFlag`
- `ki3c_MasterArbitrationWonFlag`
- `ki3c_MasterSlave2MasterFlag`

Attempts to clear other flags has no effect.

See also:

`_i3c_master_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_master_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_MasterGetStatusFlags()`.

```
static inline uint32_t I3C_MasterGetErrorStatusFlags(I3C_Type *base)
```

Gets the I3C master error status flags.

A bit mask with the state of all I3C master error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_master_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_MasterClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C master error status flag state.

See also:

`_i3c_master_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_master_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_MasterGetStatusFlags()`.

i3c_master_state_t I3C_MasterGetState(I3C_Type *base)

Gets the I3C master state.

Parameters

- base – The I3C peripheral base address.

Returns

I3C master state.

static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_flags`

Parameters

- base – The I3C 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 I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag

Attempts to clear other flags has no effect.

See also:

`_i3c_slave_flags`.

Parameters

- base – The I3C peripheral base address.
- statusMask – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetStatusFlags()`.

static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave error status flag state.

See also:

`_i3c_slave_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_slave_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetErrorStatusFlags()`.

```
i3c_slave_activity_state_t I3C_SlaveGetActivityState(I3C_Type *base)
```

Gets the I3C slave state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C slave activity state, refer `i3c_slave_activity_state_t`.

```
status_t I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)
```

```
static inline void I3C_MasterEnableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Enables the I3C master interrupt requests.

All flags except `kI3C_MasterBetweenFlag` and `kI3C_MasterNackDetectFlag` can be enabled as interrupts.

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to enable. See `_i3c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I3C_MasterDisableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Disables the I3C master interrupt requests.

All flags except `kI3C_MasterBetweenFlag` and `kI3C_MasterNackDetectFlag` can be enabled as interrupts.

Parameters

- `base` – The I3C peripheral base address.
- `interruptMask` – Bit mask of interrupts to disable. See `_i3c_master_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I3C_MasterGetEnabledInterrupts(I3C_Type *base)
```

Returns the set of currently enabled I3C master interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_master_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_MasterGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C master interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_master_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`

- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)
```

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_MasterEnableDMA(I3C_Type *base, bool enableTx, bool enableRx,
                                       uint32_t width)
```

Enables or disables I3C master DMA requests.

Parameters

- base – The I3C peripheral base address.
- enableTx – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- enableRx – Enable flag for receive DMA request. Pass true for enable, false for disable.
- width – DMA read/write unit in bytes.

```
static inline uint32_t I3C_MasterGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C master transmit data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Transmit Data Register address.

```
static inline uint32_t I3C_MasterGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C master receive data register address for DMA transfer.

Parameters

- base – The I3C peripheral base address.
- width – DMA read/write unit in bytes.

Returns

The I3C Master Receive Data Register address.

```
static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)
```

Enables or disables I3C slave DMA requests.

Parameters

- *base* – The I3C peripheral base address.
- *enableTx* – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- *enableRx* – Enable flag for receive DMA request. Pass true for enable, false for disable.
- *width* – DMA read/write unit in bytes.

```
static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- *base* – The I3C peripheral base address.
- *width* – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

```
static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave receive data register address for DMA transfer.

Parameters

- *base* – The I3C peripheral base address.
- *width* – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

```
static inline void I3C_MasterSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl, i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)
```

Sets the watermarks for I3C master FIFOs.

Parameters

- *base* – The I3C peripheral base address.
- *txLvl* – Transmit FIFO watermark level. The `kI3C_MasterTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches *txLvl*.
- *rxLvl* – Receive FIFO watermark level. The `kI3C_MasterRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches *rxLvl*.
- *flushTx* – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- *flushRx* – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

```
static inline void I3C_MasterGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of bytes in the I3C master FIFOs.

Parameters

- *base* – The I3C peripheral base address.

- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

```
static inline void I3C_SlaveSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl,
                                         i3c_rx_trigger_level_t rxLvl, bool flushTx, bool
                                         flushRx)
```

Sets the watermarks for I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txLvl` – Transmit FIFO watermark level. The `kI3C_SlaveTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches `txLvl`.
- `rxLvl` – Receive FIFO watermark level. The `kI3C_SlaveRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches `rxLvl`.
- `flushTx` – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- `flushRx` – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

```
static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- `base` – The I3C peripheral base address.
- `txCount` – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.
- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

```
void I3C_MasterSetBaudRate(I3C_Type *base, const i3c_baudrate_hz_t *baudRate_Hz, uint32_t
                          sourceClock_Hz)
```

Sets the I3C bus frequency for master transactions.

The I3C 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 I3C peripheral base address.
- `baudRate_Hz` – Pointer to structure of requested bus frequency in Hertz.
- `sourceClock_Hz` – I3C functional clock frequency in Hertz.

```
static inline bool I3C_MasterGetBusIdleState(I3C_Type *base)
```

Returns whether the bus is idle.

Requires the master mode to be enabled.

Parameters

- `base` – The I3C peripheral base address.

Return values

- `true` – Bus is busy.
- `false` – Bus is idle.

```
status_t I3C_MasterStartWithRxSize(I3C_Type *base, i3c_bus_type_t type, uint8_t address,
                                   i3c_direction_t dir, uint8_t rxSize)
```

Sends a START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *a* address parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- *base* – The I3C peripheral base address.
- *type* – The bus type to use in this transaction.
- *address* – 7-bit slave device address, in bits [6:0].
- *dir* – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- *rxSize* – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.

```
status_t I3C_MasterStart(I3C_Type *base, i3c_bus_type_t type, uint8_t address, i3c_direction_t
                          dir)
```

Sends a START signal and slave address on the I2C/I3C bus.

This function is used to initiate a new master mode transfer. First, the bus state is checked to ensure that another master is not occupying the bus. Then a START signal is transmitted, followed by the 7-bit address specified in the *address* parameter. Note that this function does not actually wait until the START and address are successfully sent on the bus before returning.

Parameters

- *base* – The I3C peripheral base address.
- *type* – The bus type to use in this transaction.
- *address* – 7-bit slave device address, in bits [6:0].
- *dir* – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

- `kStatus_Success` – START signal and address were successfully enqueued in the transmit FIFO.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.

```
status_t I3C_MasterRepeatedStartWithRxSize(I3C_Type *base, i3c_bus_type_t type, uint8_t
                                             address, i3c_direction_t dir, uint8_t rxSize)
```

Sends a repeated START signal and slave address on the I2C/I3C bus, receive size is also specified in the call.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `I3C_MasterStart()`, it also sends the specified 7-bit address. Call this API also configures the read terminate size for the following read transfer. For example, set the `rxSize = 2`, the

following read transfer will be terminated after two bytes of data received. Write transfer will not be affected by the rxSize configuration.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.
- `rxSize` – Read terminate size for the followed read transfer, limit to 255 bytes.

Return values

`kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.

```
static inline status_t I3C_MasterRepeatedStart(I3C_Type *base, i3c_bus_type_t type, uint8_t address, i3c_direction_t dir)
```

Sends a repeated START signal and slave address on the I2C/I3C bus.

This function is used to send a Repeated START signal when a transfer is already in progress. Like `I3C_MasterStart()`, it also sends the specified 7-bit address.

Note: This function exists primarily to maintain compatible APIs between I3C and I2C drivers, as well as to better document the intent of code that uses these APIs.

Parameters

- `base` – The I3C peripheral base address.
- `type` – The bus type to use in this transaction.
- `address` – 7-bit slave device address, in bits [6:0].
- `dir` – Master transfer direction, either `kI3C_Read` or `kI3C_Write`. This parameter is used to set the R/w bit (bit 0) in the transmitted slave address.

Return values

`kStatus_Success` – Repeated START signal and address were successfully enqueued in the transmit FIFO.

```
status_t I3C_MasterSend(I3C_Type *base, const void *txBuff, size_t txSize, uint32_t flags)
```

Performs a polling send transfer on the I2C/I3C 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_I3C_Nak`.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

- `flags` – Bit mask of options for the transfer. See enumeration `_i3c_master_transfer_flags` for available options.

Return values

- `kStatus_Success` – Data was sent successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Nak` – The slave device sent a NAK in response to an address.
- `kStatus_I3C_WriteAbort` – The slave device sent a NAK in response to a write.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_WriteFifoError` – Write to M/SWDATAB register when FIFO full.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

`status_t` I3C_MasterReceive(I3C_Type *base, void *rxBuff, size_t rxSize, uint32_t flags)

Performs a polling receive transfer on the I2C/I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.
- `flags` – Bit mask of options for the transfer. See enumeration `_i3c_master_transfer_flags` for available options.

Return values

- `kStatus_Success` – Data was received successfully.
- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_Term` – The master terminates slave read.
- `kStatus_I3C_HdrParityError` – Parity error from DDR read.
- `kStatus_I3C_CrcError` – CRC error from DDR read.
- `kStatus_I3C_MsgError` – Message SDR/DDR mismatch or read/write message in wrong state.
- `kStatus_I3C_ReadFifoError` – Read from M/SRDATAB register when FIFO empty.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

`status_t` I3C_MasterStop(I3C_Type *base)

Sends a STOP signal on the I2C/I3C bus.

This function does not return until the STOP signal is seen on the bus, or an error occurs.

Parameters

- `base` – The I3C peripheral base address.

Return values

- `kStatus_Success` – The STOP signal was successfully sent on the bus and the transaction terminated.

- `kStatus_I3C_Busy` – Another master is currently utilizing the bus.
- `kStatus_I3C_Timeout` – The module has stalled too long in a frame.
- `kStatus_I3C_InvalidReq` – Invalid use of request.

`void I3C_MasterEmitRequest(I3C_Type *base, i3c_bus_request_t masterReq)`
I3C master emit request.

Parameters

- `base` – The I3C peripheral base address.
- `masterReq` – I3C master request of type `i3c_bus_request_t`

`static inline void I3C_MasterEmitIBIResponse(I3C_Type *base, i3c_ibi_response_t ibiResponse)`
I3C master emit request.

Parameters

- `base` – The I3C peripheral base address.
- `ibiResponse` – I3C master emit IBI response of type `i3c_ibi_response_t`

`void I3C_MasterRegisterIBI(I3C_Type *base, i3c_register_ibi_addr_t *ibiRule)`
I3C master register IBI rule.

Parameters

- `base` – The I3C peripheral base address.
- `ibiRule` – Pointer to ibi rule description of type `i3c_register_ibi_addr_t`

`void I3C_MasterGetIBIRules(I3C_Type *base, i3c_register_ibi_addr_t *ibiRule)`
I3C master get IBI rule.

Parameters

- `base` – The I3C peripheral base address.
- `ibiRule` – Pointer to store the read out ibi rule description.

`i3c_ibi_type_t I3C_GetIBIType(I3C_Type *base)`
I3C master get IBI Type.

Parameters

- `base` – The I3C peripheral base address.

Return values

`i3c_ibi_type_t` – Type of `i3c_ibi_type_t`.

`static inline uint8_t I3C_GetIBIAddress(I3C_Type *base)`
I3C master get IBI Address.

Parameters

- `base` – The I3C peripheral base address.

Return values

The – 8-bit IBI address.

`status_t I3C_MasterProcessDAASpecifiedBaudrate(I3C_Type *base, uint8_t *addressList, uint32_t count, i3c_master_daa_baudrate_t *daaBaudRate)`

Performs a DAA in the i3c bus with specified temporary baud rate.

Parameters

- `base` – The I3C peripheral base address.

- `addressList` – The pointer for address list which is used to do DAA.
- `count` – The address count in the address list.
- `daaBaudRate` – The temporary baud rate in DAA process, NULL for using initial setting. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- `kStatus_I3C_SlaveCountExceed` – The I3C slave count has exceed the definition in `I3C_MAX_DEVCNT`.

```
static inline status_t I3C_MasterProcessDAA(I3C_Type *base, uint8_t *addressList, uint32_t count)
```

Performs a DAA in the i3c bus.

Parameters

- `base` – The I3C peripheral base address.
- `addressList` – The pointer for address list which is used to do DAA.
- `count` – The address count in the address list. The initial setting is set back between the completion of the DAA and the return of this function.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.
- `kStatus_I3C_SlaveCountExceed` – The I3C slave count has exceed the definition in `I3C_MAX_DEVCNT`.

```
i3c_device_info_t *I3C_MasterGetDeviceListAfterDAA(I3C_Type *base, uint8_t *count)
```

Get device information list after DAA process is done.

Parameters

- `base` – The I3C peripheral base address.
- `count` – **[out]** The pointer to store the available device count.

Returns

Pointer to the `i3c_device_info_t` array.

```
void I3C_MasterClearDeviceCount(I3C_Type *base)
```

Clear the global device count which represents current devices number on the bus. When user resets all dynamic addresses on the bus, should call this API.

Parameters

- `base` – The I3C peripheral base address.

```
status_t I3C_MasterTransferBlocking(I3C_Type *base, i3c_master_transfer_t *transfer)
```

Performs a master polling transfer on the I2C/I3C bus.

Note: The API does not return until the transfer succeeds or fails due to error happens during transfer.

Parameters

- base – The I3C peripheral base address.
- transfer – Pointer to the transfer structure.

Return values

- kStatus_Success – Data was received successfully.
- kStatus_I3C_Busy – Another master is currently utilizing the bus.
- kStatus_I3C_IBIWon – The I3C slave event IBI or MR or HJ won the arbitration on a header address.
- kStatus_I3C_Timeout – The module has stalled too long in a frame.
- kStatus_I3C_Nak – The slave device sent a NAK in response to an address.
- kStatus_I3C_WriteAbort – The slave device sent a NAK in response to a write.
- kStatus_I3C_Term – The master terminates slave read.
- kStatus_I3C_HdrParityError – Parity error from DDR read.
- kStatus_I3C_CrcError – CRC error from DDR read.
- kStatus_I3C_MsgError – Message SDR/DDR mismatch or read/write message in wrong state.
- kStatus_I3C_ReadFifoError – Read from M/SRDATAB register when FIFO empty.
- kStatus_I3C_WriteFifoError – Write to M/SWDATAB register when FIFO full.
- kStatus_I3C_InvalidReq – Invalid use of request.

status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)

Performs a polling send transfer on the I3C bus.

Parameters

- base – The I3C peripheral base address.
- txBuff – The pointer to the data to be transferred.
- txSize – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)

Performs a polling receive transfer on the I3C bus.

Parameters

- base – The I3C peripheral base address.
- rxBuff – The pointer to the data to be transferred.
- rxSize – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

void I3C_MasterTransferCreateHandle(I3C_Type *base, *i3c_master_handle_t* *handle, const *i3c_master_transfer_callback_t* *callback, void *userData)

Creates a new handle for the I3C 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 I3C_MasterTransferAbort() API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – **[out]** Pointer to the I3C master driver handle.
- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I3C_MasterTransferNonBlocking(I3C_Type *base, i3c_master_handle_t *handle, i3c_master_transfer_t *transfer)`

Performs a non-blocking transaction on the I2C/I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C master driver handle.
- `transfer` – The pointer to the transfer descriptor.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or a non-blocking transaction is already in progress.

`status_t I3C_MasterTransferGetCount(I3C_Type *base, i3c_master_handle_t *handle, size_t *count)`

Returns number of bytes transferred so far.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C master driver handle.
- `count` – **[out]** Number of bytes transferred so far by the non-blocking transaction.

Return values

- `kStatus_Success` –
- `kStatus_NoTransferInProgress` – There is not a non-blocking transaction currently in progress.

`void I3C_MasterTransferAbort(I3C_Type *base, i3c_master_handle_t *handle)`

Terminates a non-blocking I3C master transmission early.

Note: It is not safe to call this function from an IRQ handler that has a higher priority than the I3C peripheral's IRQ priority.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C master driver handle.

```
void I3C_MasterTransferHandleIRQ(I3C_Type *base, void *intHandle)
```

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 I3C peripheral base address.
- intHandle – Pointer to the I3C master driver handle.

```
enum _i3c_master_flags
```

I3C master peripheral flags.

The following status register flags can be cleared:

- kI3C_MasterSlaveStartFlag
- kI3C_MasterControlDoneFlag
- kI3C_MasterCompleteFlag
- kI3C_MasterArbitrationWonFlag
- kI3C_MasterSlave2MasterFlag

All flags except kI3C_MasterBetweenFlag and kI3C_MasterNackDetectFlag can be enabled as interrupts.

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

Values:

```
enumerator kI3C_MasterBetweenFlag
```

Between messages/DAA's flag

```
enumerator kI3C_MasterNackDetectFlag
```

NACK detected flag

```
enumerator kI3C_MasterSlaveStartFlag
```

Slave request start flag

```
enumerator kI3C_MasterControlDoneFlag
```

Master request complete flag

```
enumerator kI3C_MasterCompleteFlag
```

Transfer complete flag

```
enumerator kI3C_MasterRxReadyFlag
```

Rx data ready in Rx buffer flag

```
enumerator kI3C_MasterTxReadyFlag
```

Tx buffer ready for Tx data flag

```
enumerator kI3C_MasterArbitrationWonFlag
```

Header address won arbitration flag

```
enumerator kI3C_MasterErrorFlag
```

Error occurred flag

enumerator kI3C_MasterSlave2MasterFlag

Switch from slave to master flag

enumerator kI3C_MasterClearFlags

enum _i3c_master_error_flags

I3C master error flags to indicate the causes.

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

Values:

enumerator kI3C_MasterErrorNackFlag

Slave NACKed the last address

enumerator kI3C_MasterErrorWriteAbortFlag

Slave NACKed the write data

enumerator kI3C_MasterErrorParityFlag

Parity error from DDR read

enumerator kI3C_MasterErrorCrcFlag

CRC error from DDR read

enumerator kI3C_MasterErrorReadFlag

Read from MRDATAB register when FIFO empty

enumerator kI3C_MasterErrorWriteFlag

Write to MWDATAB register when FIFO full

enumerator kI3C_MasterErrorMsgFlag

Message SDR/DDR mismatch or read/write message in wrong state

enumerator kI3C_MasterErrorInvalidReqFlag

Invalid use of request

enumerator kI3C_MasterErrorTimeoutFlag

The module has stalled too long in a frame

enumerator kI3C_MasterAllErrorFlags

All error flags

enum _i3c_master_state

I3C working master state.

Values:

enumerator kI3C_MasterStateIdle

Bus stopped.

enumerator kI3C_MasterStateSlvReq

Bus stopped but slave holding SDA low.

enumerator kI3C_MasterStateMsgSdr

In SDR Message mode from using MWMSG_SDR.

enumerator kI3C_MasterStateNormAct

In normal active SDR mode.

enumerator kI3C_MasterStateDdr

In DDR Message mode.

enumerator kI3C_MasterStateDaa

In ENTDAAs mode.

enumerator kI3C_MasterStateIbiAck

Waiting on IBI ACK/NACK decision.

enumerator kI3C_MasterStateIbiRcv

Receiving IBI.

enum _i3c_master_enable

I3C master enable configuration.

Values:

enumerator kI3C_MasterOff

Master off.

enumerator kI3C_MasterOn

Master on.

enumerator kI3C_MasterCapable

Master capable.

enum _i3c_master_hkeep

I3C high keeper configuration.

Values:

enumerator kI3C_MasterHighKeeperNone

Use PUR to hold SCL high.

enumerator kI3C_MasterHighKeeperWiredIn

Use pin_HK controls.

enumerator kI3C_MasterPassiveSDA

Hi-Z for Bus Free and hold SDA.

enumerator kI3C_MasterPassiveSDASCL

Hi-Z both for Bus Free, and can Hi-Z SDA for hold.

enum _i3c_bus_request

Emits the requested operation when doing in pieces vs. by message.

Values:

enumerator kI3C_RequestNone

No request.

enumerator kI3C_RequestEmitStartAddr

Request to emit start and address on bus.

enumerator kI3C_RequestEmitStop

Request to emit stop on bus.

enumerator kI3C_RequestIbiAckNack

Manual IBI ACK or NACK.

enumerator kI3C_RequestProcessDAA

Process DAA.

enumerator kI3C_RequestForceExit

Request to force exit.

enumerator kI3C_RequestAutoIbi
Hold in stopped state, but Auto-emit START,7E.

enum _i3c_bus_type
Bus type with EmitStartAddr.

Values:

enumerator kI3C_TypeI3CSdr
SDR mode of I3C.

enumerator kI3C_TypeI2C
Standard i2c protocol.

enumerator kI3C_TypeI3CDdr
HDR-DDR mode of I3C.

enum _i3c_ibi_response
IBI response.

Values:

enumerator kI3C_IbiRespAck
ACK with no mandatory byte.

enumerator kI3C_IbiRespNack
NACK.

enumerator kI3C_IbiRespAckMandatory
ACK with mandatory byte.

enumerator kI3C_IbiRespManual
Reserved.

enum _i3c_ibi_type
IBI type.

Values:

enumerator kI3C_IbiNormal
In-band interrupt.

enumerator kI3C_IbiHotJoin
slave hot join.

enumerator kI3C_IbiMasterRequest
slave master ship request.

enum _i3c_ibi_state
IBI state.

Values:

enumerator kI3C_IbiReady
In-band interrupt ready state, ready for user to handle.

enumerator kI3C_IbiDataBuffNeed
In-band interrupt need data buffer for data receive.

enumerator kI3C_IbiAckNackPending
In-band interrupt Ack/Nack pending for decision.

enum `_i3c_direction`

Direction of master and slave transfers.

Values:

enumerator `kI3C_Write`

Master transmit.

enumerator `kI3C_Read`

Master receive.

enum `_i3c_tx_trigger_level`

Watermark of TX int/dma trigger level.

Values:

enumerator `kI3C_TxTriggerOnEmpty`

Trigger on empty.

enumerator `kI3C_TxTriggerUntilOneQuarterOrLess`

Trigger on 1/4 full or less.

enumerator `kI3C_TxTriggerUntilOneHalfOrLess`

Trigger on 1/2 full or less.

enumerator `kI3C_TxTriggerUntilOneLessThanFull`

Trigger on 1 less than full or less.

enum `_i3c_rx_trigger_level`

Watermark of RX int/dma trigger level.

Values:

enumerator `kI3C_RxTriggerOnNotEmpty`

Trigger on not empty.

enumerator `kI3C_RxTriggerUntilOneQuarterOrMore`

Trigger on 1/4 full or more.

enumerator `kI3C_RxTriggerUntilOneHalfOrMore`

Trigger on 1/2 full or more.

enumerator `kI3C_RxTriggerUntilThreeQuarterOrMore`

Trigger on 3/4 full or more.

enum `_i3c_rx_term_ops`

I3C master read termination operations.

Values:

enumerator `kI3C_RxTermDisable`

Master doesn't terminate read, used for CCC transfer.

enumerator `kI3C_RxAutoTerm`

Master auto terminate read after receiving specified bytes(<=255).

enumerator `kI3C_RxTermLastByte`

Master terminates read at any time after START, no length limitation.

enum `_i3c_start_scl_delay`

I3C start SCL delay options.

Values:

enumerator `kI3C_NoDelay`

No delay.

enumerator `kI3C_IncreaseSclHalfPeriod`

Increases SCL clock period by 1/2.

enumerator `kI3C_IncreaseSclOnePeriod`

Increases SCL clock period by 1.

enumerator `kI3C_IncreaseSclOneAndHalfPeriod`

Increases SCL clock period by 1 1/2

enum `_i3c_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 `_i3c_master_transfer::flags` field.

Values:

enumerator `kI3C_TransferDefaultFlag`

Transfer starts with a start signal, stops with a stop signal.

enumerator `kI3C_TransferNoStartFlag`

Don't send a start condition, address, and sub address

enumerator `kI3C_TransferRepeatedStartFlag`

Send a repeated start condition

enumerator `kI3C_TransferNoStopFlag`

Don't send a stop condition.

enumerator `kI3C_TransferWordsFlag`

Transfer in words, else transfer in bytes.

enumerator `kI3C_TransferDisableRxTermFlag`

Disable Rx termination. Note: It's for I3C CCC transfer.

enumerator `kI3C_TransferRxAutoTermFlag`

Set Rx auto-termination. Note: It's adaptive based on Rx size(<=255 bytes) except in `I3C_MasterReceive`.

enumerator `kI3C_TransferStartWithBroadcastAddr`

Start transfer with 0x7E, then read/write data with device address.

typedef enum `_i3c_master_state` `i3c_master_state_t`

I3C working master state.

typedef enum `_i3c_master_enable` `i3c_master_enable_t`

I3C master enable configuration.

typedef enum `_i3c_master_hkeep` `i3c_master_hkeep_t`

I3C high keeper configuration.

typedef enum `_i3c_bus_request` `i3c_bus_request_t`

Emits the requested operation when doing in pieces vs. by message.

typedef enum `_i3c_bus_type` `i3c_bus_type_t`

Bus type with `EmitStartAddr`.

```

typedef enum _i3c_ibi_response i3c_ibi_response_t
    IBI response.

typedef enum _i3c_ibi_type i3c_ibi_type_t
    IBI type.

typedef enum _i3c_ibi_state i3c_ibi_state_t
    IBI state.

typedef enum _i3c_direction i3c_direction_t
    Direction of master and slave transfers.

typedef enum _i3c_tx_trigger_level i3c_tx_trigger_level_t
    Watermark of TX int/dma trigger level.

typedef enum _i3c_rx_trigger_level i3c_rx_trigger_level_t
    Watermark of RX int/dma trigger level.

typedef enum _i3c_rx_term_ops i3c_rx_term_ops_t
    I3C master read termination operations.

typedef enum _i3c_start_scl_delay i3c_start_scl_delay_t
    I3C start SCL delay options.

typedef struct _i3c_register_ibi_addr i3c_register_ibi_addr_t
    Structure with setting master IBI rules and slave registry.

typedef struct _i3c_baudrate i3c_baudrate_hz_t
    Structure with I3C baudrate settings.

typedef struct _i3c_master_daa_baudrate i3c_master_daa_baudrate_t
    I3C DAA baud rate configuration.

typedef struct _i3c_master_config i3c_master_config_t
    Structure with settings to initialize the I3C master module.

    This structure holds configuration settings for the I3C peripheral. To initialize this structure
    to reasonable defaults, call the I3C_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 _i3c_master_transfer i3c_master_transfer_t

typedef struct _i3c_master_handle i3c_master_handle_t

typedef struct _i3c_master_transfer_callback i3c_master_transfer_callback_t
    i3c master callback functions.

typedef void (*i3c_master_isr_t)(I3C_Type *base, void *handle)
    Typedef for master interrupt handler.

struct _i3c_register_ibi_addr
    #include <fsl_i3c.h> Structure with setting master IBI rules and slave registry.

```

Public Members

```

uint8_t address[5]
    Address array for registry.

bool i3cFastStart
    Allow the START header to run as push-pull speed if all dynamic addresses take MSB
    0.

```

bool ibiHasPayload

Whether the address array has mandatory IBI byte.

struct _i3c_baudrate

#include <fsl_i3c.h> Structure with I3C baudrate settings.

Public Members

uint32_t i2cBaud

Desired I2C baud rate in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct _i3c_master_daa_baudrate

#include <fsl_i3c.h> I3C DAA baud rate configuration.

Public Members

uint32_t sourceClock_Hz

FCLK, function clock in Hertz.

uint32_t i3cPushPullBaud

Desired I3C push-pull baud rate in Hertz.

uint32_t i3cOpenDrainBaud

Desired I3C open-drain baud rate in Hertz.

struct _i3c_master_config

#include <fsl_i3c.h> Structure with settings to initialize the I3C master module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_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

i3c_master_enable_t enableMaster

Enable master mode.

bool disableTimeout

Whether to disable timeout to prevent the ERRWARN.

i3c_master_hkeep_t hKeep

High keeper mode setting.

bool enableOpenDrainStop

Whether to emit open-drain speed STOP.

bool enableOpenDrainHigh

Enable Open-Drain High to be 1 PPBAUD count for i3c messages, or 1 ODBAUD.

i3c_baudrate_hz_t baudRate_Hz

Desired baud rate settings.

i3c_start_scl_delay_t startSclDelay

I3C SCL delay after START.

i3c_start_scl_delay_t restartSclDelay

I3C SCL delay after Repeated START.

struct *_i3c_master_transfer_callback*

#include <fsl_i3c.h> i3c master callback functions.

Public Members

void (*slave2Master)(I3C_Type *base, void *userData)

Transfer complete callback

void (*ibiCallback)(I3C_Type *base, *i3c_master_handle_t* *handle, *i3c_ibi_type_t* ibiType, *i3c_ibi_state_t* ibiState)

IBI event callback

void (*transferComplete)(I3C_Type *base, *i3c_master_handle_t* *handle, *status_t* completionStatus, void *userData)

Transfer complete callback

struct *_i3c_master_transfer*

#include <fsl_i3c.h> Non-blocking transfer descriptor structure.

This structure is used to pass transaction parameters to the *I3C_MasterTransferNonBlocking()* API.

Public Members

uint32_t flags

Bit mask of options for the transfer. See enumeration *_i3c_master_transfer_flags* for available options. Set to 0 or *kI3C_TransferDefaultFlag* for normal transfers.

uint8_t slaveAddress

The 7-bit slave address.

i3c_direction_t direction

Either *kI3C_Read* or *kI3C_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.

i3c_bus_type_t busType

bus type.

i3c_ibi_response_t ibiResponse

ibi response during transfer.

```
struct _i3c_master_handle
#include <fsl_i3c.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` remainingBytes
Remaining byte count in current state.

`i3c_rx_term_ops_t` rxTermOps
Read termination operation.

`i3c_master_transfer_t` transfer
Copy of the current transfer info.

`uint8_t` ibiAddress
Slave address which request IBI.

`uint8_t *`ibiBuff
Pointer to IBI buffer to keep ibi bytes.

`size_t` ibiPayloadSize
IBI payload size.

`i3c_ibi_type_t` ibiType
IBI type.

`i3c_master_transfer_callback_t` callback
Callback functions pointer.

`void *`userData
Application data passed to callback.

2.13 I3C Slave DMA Driver

```
void I3C_SlaveTransferCreateHandleDMA(I3C_Type *base, i3c_slave_dma_handle_t *handle,
                                     i3c_slave_dma_callback_t callback, void *userData,
                                     dma_handle_t *rxDmaHandle, dma_handle_t
                                     *txDmaHandle)
```

Create a new handle for the I3C slave DMA APIs.

The creation of a handle is for use with the DMA APIs. Once a handle is created, there is not a corresponding destroy handle. If the user wants to terminate a transfer, the `I3C_SlaveTransferAbortDMA()` API shall be called.

For devices where the I3C send and receive DMA requests are OR'd together, the `txDmaHandle` parameter is ignored and may be set to NULL.

Parameters

- base – The I3C peripheral base address.
- handle – Pointer to the I3C slave driver handle.
- callback – User provided pointer to the asynchronous callback function.

- `userData` – User provided pointer to the application callback data.
- `rxDmaHandle` – Handle for the DMA receive channel. Created by the user prior to calling this function.
- `txDmaHandle` – Handle for the DMA transmit channel. Created by the user prior to calling this function.

```
status_t I3C_SlaveTransferDMA(I3C_Type *base, i3c_slave_dma_handle_t *handle,
                             i3c_slave_dma_transfer_t *transfer, uint32_t eventMask)
```

Prepares for a non-blocking DMA-based transaction on the I3C bus.

The API will do DMA configuration according to the input transfer descriptor, and the data will be transferred when there's bus master requesting transfer from/to this slave. So the timing of call to this API need be aligned with master application to ensure the transfer is executed as expected. Callback specified when the *handle* was created is invoked when the transaction has completed.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to the I3C slave driver handle.
- `transfer` – The pointer to the transfer descriptor.
- `eventMask` – Bit mask formed by OR'ing together `i3c_slave_transfer_event_t` enumerators to specify which events to send to the callback. The transmit and receive events is not allowed to be enabled.

Return values

- `kStatus_Success` – The transaction was started successfully.
- `kStatus_I3C_Busy` – Either another master is currently utilizing the bus, or another DMA transaction is already in progress.

```
void I3C_SlaveTransferAbortDMA(I3C_Type *base, i3c_slave_dma_handle_t *handle)
```

Abort a slave dma non-blocking transfer in a early time.

Parameters

- `base` – I3C peripheral base address
- `handle` – pointer to `i3c_slave_dma_handle_t` structure

```
void I3C_SlaveTransferDMAHandleIRQ(I3C_Type *base, void *i3CHandle)
```

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 I3C peripheral base address.
- `handle` – Pointer to the I3C slave DMA driver handle.

```
typedef struct _i3c_slave_dma_handle i3c_slave_dma_handle_t
```

```
typedef struct _i3c_slave_dma_transfer i3c_slave_dma_transfer_t
I3C slave transfer structure.
```

```
typedef void (*i3c_slave_dma_callback_t)(I3C_Type *base, i3c_slave_dma_transfer_t *transfer, void *userData)
```

Slave event callback function pointer type.

This callback is used only for the slave DMA transfer API.

Param base

Base address for the I3C instance on which the event occurred.

Param handle

Pointer to slave DMA transfer handle.

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.

```
struct _i3c_slave_dma_transfer
```

```
    #include <fsl_i3c_dma.h> I3C slave transfer structure.
```

```
struct _i3c_slave_dma_handle
```

```
    #include <fsl_i3c_dma.h> I3C slave dma handle structure.
```

Note: The contents of this structure are private and subject to change.

2.14 I3C Slave Driver

```
void I3C_SlaveGetDefaultConfig(i3c_slave_config_t *slaveConfig)
```

Provides a default configuration for the I3C slave peripheral.

This function provides the following default configuration for the I3C slave peripheral:

```
slaveConfig->enableSlave = true;
```

After calling this function, you can override any settings in order to customize the configuration, prior to initializing the slave driver with `I3C_SlaveInit()`.

Parameters

- `slaveConfig` – **[out]** User provided configuration structure for default values. Refer to `i3c_slave_config_t`.

```
void I3C_SlaveInit(I3C_Type *base, const i3c_slave_config_t *slaveConfig, uint32_t slowClock_Hz)
```

Initializes the I3C slave peripheral.

This function enables the peripheral clock and initializes the I3C slave peripheral as described by the user provided configuration.

Parameters

- `base` – The I3C peripheral base address.
- `slaveConfig` – User provided peripheral configuration. Use `I3C_SlaveGetDefaultConfig()` to get a set of defaults that you can override.
- `slowClock_Hz` – Frequency in Hertz of the I3C slow clock. Used to calculate the bus match condition values. If `FSL_FEATURE_I3C_HAS_NO_SCONFIG_BAMATCH` defines as 1, this parameter is useless.

`void I3C_SlaveDeinit(I3C_Type *base)`

Deinitializes the I3C slave peripheral.

This function disables the I3C slave peripheral and gates the clock.

Parameters

- `base` – The I3C peripheral base address.

`static inline void I3C_SlaveEnable(I3C_Type *base, bool isEnabled)`

Enable/Disable Slave.

Parameters

- `base` – The I3C peripheral base address.
- `isEnabled` – Enable or disable.

`static inline uint32_t I3C_SlaveGetStatusFlags(I3C_Type *base)`

Gets the I3C slave status flags.

A bit mask with the state of all I3C slave status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_flags`

Parameters

- `base` – The I3C 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 I3C_SlaveClearStatusFlags(I3C_Type *base, uint32_t statusMask)`

Clears the I3C slave status flag state.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Attempts to clear other flags has no effect.

See also:

`_i3c_slave_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of status flags that are to be cleared. The mask is composed of `_i3c_slave_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetStatusFlags()`.

```
static inline uint32_t I3C_SlaveGetErrorStatusFlags(I3C_Type *base)
```

Gets the I3C slave error status flags.

A bit mask with the state of all I3C slave error status flags is returned. For each flag, the corresponding bit in the return value is set if the flag is asserted.

See also:

`_i3c_slave_error_flags`

Parameters

- `base` – The I3C peripheral base address.

Returns

State of the error status flags:

- 1: related status flag is set.
- 0: related status flag is not set.

```
static inline void I3C_SlaveClearErrorStatusFlags(I3C_Type *base, uint32_t statusMask)
```

Clears the I3C slave error status flag state.

See also:

`_i3c_slave_error_flags`.

Parameters

- `base` – The I3C peripheral base address.
- `statusMask` – A bitmask of error status flags that are to be cleared. The mask is composed of `_i3c_slave_error_flags` enumerators OR'd together. You may pass the result of a previous call to `I3C_SlaveGetErrorStatusFlags()`.

```
i3c_slave_activity_state_t I3C_SlaveGetActivityState(I3C_Type *base)
```

Gets the I3C slave state.

Parameters

- `base` – The I3C peripheral base address.

Returns

I3C slave activity state, refer `i3c_slave_activity_state_t`.

```
status_t I3C_SlaveCheckAndClearError(I3C_Type *base, uint32_t status)
```

```
static inline void I3C_SlaveEnableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Enables the I3C slave interrupt requests.

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`

- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to enable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline void I3C_SlaveDisableInterrupts(I3C_Type *base, uint32_t interruptMask)
```

Disables the I3C slave interrupt requests.

Only below flags can be disabled as interrupts.

- kI3C_SlaveBusStartFlag
- kI3C_SlaveMatchedFlag
- kI3C_SlaveBusStopFlag
- kI3C_SlaveRxReadyFlag
- kI3C_SlaveTxReadyFlag
- kI3C_SlaveDynamicAddrChangedFlag
- kI3C_SlaveReceivedCCCFlag
- kI3C_SlaveErrorFlag
- kI3C_SlaveHDRCommandMatchFlag
- kI3C_SlaveCCCHandledFlag
- kI3C_SlaveEventSentFlag

Parameters

- base – The I3C peripheral base address.
- interruptMask – Bit mask of interrupts to disable. See `_i3c_slave_flags` for the set of constants that should be OR'd together to form the bit mask.

```
static inline uint32_t I3C_SlaveGetEnabledInterrupts(I3C_Type *base)
```

Returns the set of currently enabled I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of enabled interrupts.

```
static inline uint32_t I3C_SlaveGetPendingInterrupts(I3C_Type *base)
```

Returns the set of pending I3C slave interrupt requests.

Parameters

- base – The I3C peripheral base address.

Returns

A bitmask composed of `_i3c_slave_flags` enumerators OR'd together to indicate the set of pending interrupts.

```
static inline void I3C_SlaveEnableDMA(I3C_Type *base, bool enableTx, bool enableRx, uint32_t width)
```

Enables or disables I3C slave DMA requests.

Parameters

- *base* – The I3C peripheral base address.
- *enableTx* – Enable flag for transmit DMA request. Pass true for enable, false for disable.
- *enableRx* – Enable flag for receive DMA request. Pass true for enable, false for disable.
- *width* – DMA read/write unit in bytes.

```
static inline uint32_t I3C_SlaveGetTxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave transmit data register address for DMA transfer.

Parameters

- *base* – The I3C peripheral base address.
- *width* – DMA read/write unit in bytes.

Returns

The I3C Slave Transmit Data Register address.

```
static inline uint32_t I3C_SlaveGetRxFifoAddress(I3C_Type *base, uint32_t width)
```

Gets I3C slave receive data register address for DMA transfer.

Parameters

- *base* – The I3C peripheral base address.
- *width* – DMA read/write unit in bytes.

Returns

The I3C Slave Receive Data Register address.

```
static inline void I3C_SlaveSetWatermarks(I3C_Type *base, i3c_tx_trigger_level_t txLvl, i3c_rx_trigger_level_t rxLvl, bool flushTx, bool flushRx)
```

Sets the watermarks for I3C slave FIFOs.

Parameters

- *base* – The I3C peripheral base address.
- *txLvl* – Transmit FIFO watermark level. The `kI3C_SlaveTxReadyFlag` flag is set whenever the number of words in the transmit FIFO reaches *txLvl*.
- *rxLvl* – Receive FIFO watermark level. The `kI3C_SlaveRxReadyFlag` flag is set whenever the number of words in the receive FIFO reaches *rxLvl*.
- *flushTx* – true if TX FIFO is to be cleared, otherwise TX FIFO remains unchanged.
- *flushRx* – true if RX FIFO is to be cleared, otherwise RX FIFO remains unchanged.

```
static inline void I3C_SlaveGetFifoCounts(I3C_Type *base, size_t *rxCount, size_t *txCount)
```

Gets the current number of bytes in the I3C slave FIFOs.

Parameters

- *base* – The I3C peripheral base address.
- *txCount* – **[out]** Pointer through which the current number of bytes in the transmit FIFO is returned. Pass NULL if this value is not required.

- `rxCount` – **[out]** Pointer through which the current number of bytes in the receive FIFO is returned. Pass NULL if this value is not required.

`status_t I3C_SlaveSend(I3C_Type *base, const void *txBuff, size_t txSize)`

Performs a polling send transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `txBuff` – The pointer to the data to be transferred.
- `txSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`status_t I3C_SlaveReceive(I3C_Type *base, void *rxBuff, size_t rxSize)`

Performs a polling receive transfer on the I3C bus.

Parameters

- `base` – The I3C peripheral base address.
- `rxBuff` – The pointer to the data to be transferred.
- `rxSize` – The length in bytes of the data to be transferred.

Returns

Error or success status returned by API.

`void I3C_SlaveTransferCreateHandle(I3C_Type *base, i3c_slave_handle_t *handle, i3c_slave_transfer_callback_t callback, void *userData)`

Creates a new handle for the I3C 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 `I3C_SlaveTransferAbort()` API shall be called.

Note: The function also enables the NVIC IRQ for the input I3C. Need to notice that on some SoCs the I3C IRQ is connected to INTMUX, in this case user needs to enable the associated INTMUX IRQ in application.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – **[out]** Pointer to the I3C slave driver handle.
- `callback` – User provided pointer to the asynchronous callback function.
- `userData` – User provided pointer to the application callback data.

`status_t I3C_SlaveTransferNonBlocking(I3C_Type *base, i3c_slave_handle_t *handle, uint32_t eventMask)`

Starts accepting slave transfers.

Call this API after calling `I2C_SlaveInit()` and `I3C_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 `I3C_SlaveTransferCreateHandle()`. The callback is always invoked from the interrupt context.

The set of events received by the callback is customizable. To do so, set the `eventMask` parameter to the OR'd combination of `i3c_slave_transfer_event_t` enumerators for the events you wish to receive. The `kI3C_SlaveTransmitEvent` and `kI3C_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 `kI3C_SlaveAllEvents` constant is provided as a convenient way to enable all events.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.
- `eventMask` – Bit mask formed by OR'ing together `i3c_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 `kI3C_SlaveAllEvents` to enable all events.

Return values

- `kStatus_Success` – Slave transfers were successfully started.
- `kStatus_I3C_Busy` – Slave transfers have already been started on this handle.

`status_t I3C_SlaveTransferGetCount(I3C_Type *base, i3c_slave_handle_t *handle, size_t *count)`
Gets the slave transfer status during a non-blocking transfer.

Parameters

- `base` – The I3C peripheral base address.
- `handle` – Pointer to `i2c_slave_handle_t` structure.
- `count` – **[out]** Pointer to a value to hold the number of bytes transferred. May be NULL if the count is not required.

Return values

- `kStatus_Success` –
- `kStatus_NoTransferInProgress` –

`void I3C_SlaveTransferAbort(I3C_Type *base, i3c_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 I3C peripheral base address.
- `handle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

`void I3C_SlaveTransferHandleIRQ(I3C_Type *base, void *intHandle)`
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 I3C peripheral base address.
- `intHandle` – Pointer to struct: `_i3c_slave_handle` structure which stores the transfer state.

enum `_i3c_slave_flags`

I3C slave peripheral flags.

The following status register flags can be cleared:

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`

Only below flags can be enabled as interrupts.

- `kI3C_SlaveBusStartFlag`
- `kI3C_SlaveMatchedFlag`
- `kI3C_SlaveBusStopFlag`
- `kI3C_SlaveRxReadyFlag`
- `kI3C_SlaveTxReadyFlag`
- `kI3C_SlaveDynamicAddrChangedFlag`
- `kI3C_SlaveReceivedCCCFlag`
- `kI3C_SlaveErrorFlag`
- `kI3C_SlaveHDRCommandMatchFlag`
- `kI3C_SlaveCCCHandledFlag`
- `kI3C_SlaveEventSentFlag`

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

Values:

enumerator `kI3C_SlaveNotStopFlag`

Slave status not stop flag

enumerator `kI3C_SlaveMessageFlag`

Slave status message, indicating slave is listening to the bus traffic or responding

enumerator `kI3C_SlaveRequiredReadFlag`

Slave status required, either is master doing SDR read from slave, or is IBI pushing out.

enumerator `kI3C_SlaveRequiredWriteFlag`

Slave status request write, master is doing SDR write to slave, except slave in ENTDAAMode

enumerator `kI3C_SlaveBusDAAModeFlag`

I3C bus is in ENTDAAMode

enumerator `kI3C_SlaveBusHDRModeFlag`

I3C bus is in HDR mode

enumerator `kI3C_SlaveBusStartFlag`

Start/Re-start event is seen since the bus was last cleared

enumerator `kI3C_SlaveMatchedFlag`

Slave address(dynamic/static) matched since last cleared

enumerator `kI3C_SlaveBusStopFlag`

Stop event is seen since the bus was last cleared

enumerator kI3C_SlaveRxReadyFlag
Rx data ready in rx buffer flag

enumerator kI3C_SlaveTxReadyFlag
Tx buffer ready for Tx data flag

enumerator kI3C_SlaveDynamicAddrChangedFlag
Slave dynamic address has been assigned, re-assigned, or lost

enumerator kI3C_SlaveReceivedCCCFlag
Slave received Common command code

enumerator kI3C_SlaveErrorFlag
Error occurred flag

enumerator kI3C_SlaveHDRCommandMatchFlag
High data rate command match

enumerator kI3C_SlaveCCCHandledFlag
Slave received Common command code is handled by I3C module

enumerator kI3C_SlaveEventSentFlag
Slave IBI/P2P/MR/HJ event has been sent

enumerator kI3C_SlaveIbiDisableFlag
Slave in band interrupt is disabled.

enumerator kI3C_SlaveMasterRequestDisabledFlag
Slave master request is disabled.

enumerator kI3C_SlaveHotJoinDisabledFlag
Slave Hot-Join is disabled.

enumerator kI3C_SlaveClearFlags
All flags which are cleared by the driver upon starting a transfer.

enumerator kI3C_SlaveAllIrqFlags

enum _i3c_slave_error_flags
I3C slave error flags to indicate the causes.

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

Values:

enumerator kI3C_SlaveErrorOverrunFlag
Slave internal from-bus buffer/FIFO overrun.

enumerator kI3C_SlaveErrorUnderrunFlag
Slave internal to-bus buffer/FIFO underrun

enumerator kI3C_SlaveErrorUnderrunNakFlag
Slave internal from-bus buffer/FIFO underrun and NACK error

enumerator kI3C_SlaveErrorTermFlag
Terminate error from master

enumerator kI3C_SlaveErrorInvalidStartFlag
Slave invalid start flag

enumerator kI3C_SlaveErrorSdrParityFlag
SDR parity error

enumerator kI3C_SlaveErrorHdrParityFlag
HDR parity error

enumerator kI3C_SlaveErrorHdrCRCFlag
HDR-DDR CRC error

enumerator kI3C_SlaveErrorS0S1Flag
S0 or S1 error

enumerator kI3C_SlaveErrorOverreadFlag
Over-read error

enumerator kI3C_SlaveErrorOverwriteFlag
Over-write error

enum _i3c_slave_event
I3C slave.event.

Values:

enumerator kI3C_SlaveEventNormal
Normal mode.

enumerator kI3C_SlaveEventIBI
In band interrupt event.

enumerator kI3C_SlaveEventMasterReq
Master request event.

enumerator kI3C_SlaveEventHotJoinReq
Hot-join event.

enum _i3c_slave_activity_state
I3C slave.activity state.

Values:

enumerator kI3C_SlaveNoLatency
Normal bus operation

enumerator kI3C_SlaveLatency1Ms
1ms of latency.

enumerator kI3C_SlaveLatency100Ms
100ms of latency.

enumerator kI3C_SlaveLatency10S
10s latency.

enum _i3c_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 `I3C_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 `kI3C_SlaveAddressMatchEvent`

Received the slave address after a start or repeated start.

enumerator `kI3C_SlaveTransmitEvent`

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

enumerator `kI3C_SlaveReceiveEvent`

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

enumerator `kI3C_SlaveRequiredTransmitEvent`

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

enumerator `kI3C_SlaveStartEvent`

A start/repeated start was detected.

enumerator `kI3C_SlaveHDRCommandMatchEvent`

Slave Match HDR Command.

enumerator `kI3C_SlaveCompletionEvent`

A stop was detected, completing the transfer.

enumerator `kI3C_SlaveRequestSentEvent`

Slave request event sent.

enumerator `kI3C_SlaveReceivedCCCEvent`

Slave received CCC event, need to handle by application.

enumerator `kI3C_SlaveAllEvents`

Bit mask of all available events.

typedef enum `_i3c_slave_event` `i3c_slave_event_t`

I3C slave.event.

typedef enum `_i3c_slave_activity_state` `i3c_slave_activity_state_t`

I3C slave.activity state.

typedef struct `_i3c_slave_config` `i3c_slave_config_t`

Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the `I3C_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 `_i3c_slave_transfer_event` `i3c_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 `I3C_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 `_i3c_slave_transfer` `i3c_slave_transfer_t`

I3C slave transfer structure.

typedef struct `_i3c_slave_handle` `i3c_slave_handle_t`

```
typedef void (*i3c_slave_transfer_callback_t)(I3C_Type *base, i3c_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 I3C_SlaveSetCallback() function after you have created a handle.

Param base

Base address for the I3C 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 void (*i3c_slave_isr_t)(I3C_Type *base, void *handle)
```

Typedef for slave interrupt handler.

```
struct i3c_slave_config
```

#include <fsl_i3c.h> Structure with settings to initialize the I3C slave module.

This structure holds configuration settings for the I3C peripheral. To initialize this structure to reasonable defaults, call the I3C_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

bool enableSlave

Whether to enable slave.

uint8_t staticAddr

Static address.

uint16_t vendorID

Device vendor ID(manufacture ID).

uint32_t partNumber

Device part number info

uint8_t dcr

Device characteristics register information.

uint8_t bcr

Bus characteristics register information.

uint8_t hdrMode

Support hdr mode, could be OR logic in enumeration:i3c_hdr_mode_t.

bool nakAllRequest

Whether to reply NAK to all requests except broadcast CCC.

bool ignoreS0S1Error

Whether to ignore S0/S1 error in SDR mode.

bool offline

Whether to wait 60 us of bus quiet or HDR request to ensure slave track SDR mode safely.

bool matchSlaveStartStop

Whether to assert start/stop status only the time slave is addressed.

uint32_t maxWriteLength

Maximum write length.

uint32_t maxReadLength

Maximum read length.

struct `_i3c_slave_transfer`

#include <fsl_i3c.h> I3C slave transfer structure.

Public Members

uint32_t event

Reason the callback is being invoked.

uint8_t *txData

Transfer buffer

size_t txDataSize

Transfer size

uint8_t *rxData

Transfer buffer

size_t rxDataSize

Transfer size

status_t completionStatus

Success or error code describing how the transfer completed. Only applies for `kI3C_SlaveCompletionEvent`.

size_t transferredCount

Number of bytes actually transferred since start or last repeated start.

struct `_i3c_slave_handle`

#include <fsl_i3c.h> I3C slave handle structure.

Note: The contents of this structure are private and subject to change.

Public Members

i3c_slave_transfer_t transfer

I3C slave transfer copy.

bool isBusy

Whether transfer is busy.

bool wasTransmit

Whether the last transfer was a transmit.

uint32_t eventMask

Mask of enabled events.

uint32_t transferredCount

Count of bytes transferred.

i3c_slave_transfer_callback_t callback
 Callback function called at transfer event.

void *userData
 Callback parameter passed to callback.

size_t txFifoSize
 Tx Fifo size

2.15 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 numOfBytes, 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.

`status_t` IAP_ExtendedFlashSignatureRead(`uint32_t` startPage, `uint32_t` endPage, `uint32_t` numOfStates, `uint32_t` *signature)

Extended Read signature.

This function calculates the signature value for one or more pages of on-chip flash memory.

Parameters

- `startPage` – Start page number.
- `endPage` – End page number.

- numOfStates – Number of wait states.
- signature – Address to store the signature value.

Return values

kStatus_IAP_Success – Api has been executed successfully.

status_t IAP_ReadFlashSignature(uint32_t *signature)

Read flash signature.

This function is used to obtain a 32-bit signature value of the entire flash memory.

Parameters

- signature – Address to store the 32-bit generated signature value.

Return values

kStatus_IAP_Success – Api has been executed successfully.

status_t IAP_ReadFAIMPage(uint32_t pageNumber, uint32_t *dstAddr)

Read FAIM page.

This function is used to read given page of FAIM into the memory provided.

Parameters

- pageNumber – FAIM page number.
- dstAddr – Memory address to store the value read from FAIM.

Return values

- kStatus_IAP_Success – Api has been executed successfully.
- kStatus_IAP_DstAddrNotMapped – Destination address is not mapped in the memory map.

status_t IAP_WriteFAIMPage(uint32_t pageNumber, uint32_t *srcAddr)

Write FAIM page.

This function is used to write given data in the provided memory to a page of G.

Parameters

- pageNumber – FAIM page number.
- srcAddr – Memory address holding data to be stored on to FAIM page.

Return values

- kStatus_IAP_Success – Api has been executed successfully.
- kStatus_IAP_SrcAddrNotMapped – Source address is not mapped in the memory map.

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_ReadFAIMPage
Read FAIM page

enumerator kIapCmd_IAP_WriteFAIMPage
Write FAIM 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

2.16 INPUTMUX: Input Multiplexing Driver

enum _inputmux_connection_t
INPUTMUX connections type.
Values:

enumerator kINPUTMUX_DmaChannel0TrigoutToTriginChannels
DMA OTRIG.

enumerator kINPUTMUX_DmaChannel1TrigoutToTriginChannels

enumerator kINPUTMUX_DmaChannel2TrigoutToTriginChannels

enumerator kINPUTMUX_DmaChannel3TrigoutToTriginChannels

enumerator kINPUTMUX_DmaChannel4TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel5TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel6TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel7TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel8TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel9TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel10TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel11TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel12TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel13TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel14TrigoutToTriginChannels
enumerator kINPUTMUX_DmaChannel15TrigoutToTriginChannels

DMA ITRIG.

enumerator kINPUTMUX_GpioInt4ToDma
enumerator kINPUTMUX_GpioInt5ToDma
enumerator kINPUTMUX_GpioInt6ToDma
enumerator kINPUTMUX_GpioInt7ToDma
enumerator kINPUTMUX_Adc0SeqaIrqToDma
enumerator kINPUTMUX_Adc0SeqbIrqToDma
enumerator kINPUTMUX_Comp0OutToDma
enumerator kINPUTMUX_Ftm0InitTrigToDma
enumerator kINPUTMUX_Ftm1InitTrigToDma
enumerator kINPUTMUX_Ftm0Ch0ToCh5ToDma
enumerator kINPUTMUX_Ftm1Ch0ToCh3ToDma
enumerator kINPUTMUX_SdmaTrigoutAToDma
enumerator kINPUTMUX_SdmaTrigoutBToDma

typedef enum *inputmux_connection_t* inputmux_connection_t
INPUTMUX connections type.

DMA_OTRIG_PMUX_ID

Periphinmux IDs.

DMA_TRIG0_PMUX_ID

PMUX_SHIFT

FSL_INPUTMUX_DRIVER_VERSION

Group interrupt driver version for SDK.

```
void INPUTMUX_Init(void *base)
```

Initialize INPUTMUX peripheral.

This function enables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

```
void INPUTMUX_AttachSignal(void *base, uint16_t index, inputmux_connection_t connection)
```

Attaches a signal.

This function attaches multiplexed signals from INPUTMUX to target signals. For example, to attach GPIO PORT0 Pin 5 to PINT peripheral, do the following:

```
INPUTMUX_AttachSignal(INPUTMUX, 2, kINPUTMUX_GpioPort0Pin5ToPintsel);
```

In this example, INTMUX has 8 registers for PINT, PINT_SEL0~PINT_SEL7. With parameter index specified as 2, this function configures register PINT_SEL2.

Parameters

- base – Base address of the INPUTMUX peripheral.
- index – The serial number of destination register in the group of INPUTMUX registers with same name.
- connection – Applies signal from source signals collection to target signal.

Return values

None. –

```
void INPUTMUX_EnableSignal(void *base, inputmux_signal_t signal, bool enable)
```

Enable/disable a signal.

This function gates the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.
- signal – Enable signal register id and bit offset.
- enable – Selects enable or disable.

Return values

None. –

```
void INPUTMUX_Deinit(void *base)
```

Deinitialize INPUTMUX peripheral.

This function disables the INPUTMUX clock.

Parameters

- base – Base address of the INPUTMUX peripheral.

Return values

None. –

2.17 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 I2C status codes.

enumerator *kStatusGroup_LPSCI*

Group number for LPSCI status codes.

enumerator *kStatusGroup_LPUART*

Group number for LPUART status codes.

- enumerator `kStatusGroup_SPI`
Group number for SPI status code.
- enumerator `kStatusGroup_XRDC`
Group number for XRDC status code.
- enumerator `kStatusGroup_SEMA42`
Group number for SEMA42 status code.
- enumerator `kStatusGroup_SDHC`
Group number for SDHC status code
- enumerator `kStatusGroup_SDMMC`
Group number for SDMMC status code
- enumerator `kStatusGroup_SAI`
Group number for SAI status code
- enumerator `kStatusGroup_MCG`
Group number for MCG status codes.
- enumerator `kStatusGroup_SCG`
Group number for SCG status codes.
- enumerator `kStatusGroup_SDSPI`
Group number for SDSPI status codes.
- enumerator `kStatusGroup_FLEXIO_I2S`
Group number for FLEXIO I2S status codes
- enumerator `kStatusGroup_FLEXIO_MCULCD`
Group number for FLEXIO LCD status codes
- enumerator `kStatusGroup_FLASHIAP`
Group number for FLASHIAP status codes
- enumerator `kStatusGroup_FLEXCOMM_I2C`
Group number for FLEXCOMM I2C status codes
- enumerator `kStatusGroup_I2S`
Group number for I2S status codes
- enumerator `kStatusGroup_IUART`
Group number for IUART status codes
- enumerator `kStatusGroup_CSI`
Group number for CSI status codes
- enumerator `kStatusGroup_MIPI_DSI`
Group number for MIPI DSI status codes
- enumerator `kStatusGroup_SDRAMC`
Group number for SDRAMC status codes.
- enumerator `kStatusGroup_POWER`
Group number for POWER status codes.
- enumerator `kStatusGroup_ENET`
Group number for ENET status codes.
- enumerator `kStatusGroup_PHY`
Group number for PHY status codes.

- enumerator kStatusGroup_TRGMUX
Group number for TRGMUX status codes.
- enumerator kStatusGroup_SMARTCARD
Group number for SMARTCARD status codes.
- enumerator kStatusGroup_LMEM
Group number for LMEM status codes.
- enumerator kStatusGroup_QSPI
Group number for QSPI status codes.
- enumerator kStatusGroup_DMA
Group number for DMA status codes.
- enumerator kStatusGroup_EDMA
Group number for EDMA status codes.
- enumerator kStatusGroup_DMAMGR
Group number for DMAMGR status codes.
- enumerator kStatusGroup_FLEXCAN
Group number for FlexCAN status codes.
- enumerator kStatusGroup_LTC
Group number for LTC status codes.
- enumerator kStatusGroup_FLEXIO_CAMERA
Group number for FLEXIO CAMERA status codes.
- enumerator kStatusGroup_LPC_SPI
Group number for LPC_SPI status codes.
- enumerator kStatusGroup_LPC_USART
Group number for LPC_USART status codes.
- enumerator kStatusGroup_DMIC
Group number for DMIC status codes.
- enumerator kStatusGroup_SDIF
Group number for SDIF status codes.
- enumerator kStatusGroup_SPIFI
Group number for SPIFI status codes.
- enumerator kStatusGroup_OTP
Group number for OTP status codes.
- enumerator kStatusGroup_MCAN
Group number for MCAN status codes.
- enumerator kStatusGroup_CAAM
Group number for CAAM status codes.
- enumerator kStatusGroup_ECSPi
Group number for ECSPi status codes.
- enumerator kStatusGroup_USDHC
Group number for USDHC status codes.
- enumerator kStatusGroup_LPC_I2C
Group number for LPC_I2C status codes.

- enumerator `kStatusGroup_DCP`
Group number for DCP status codes.
- enumerator `kStatusGroup_MSCAN`
Group number for MSCAN status codes.
- enumerator `kStatusGroup_ESAI`
Group number for ESAI status codes.
- enumerator `kStatusGroup_FLEXSPI`
Group number for FLEXSPI status codes.
- enumerator `kStatusGroup_MMDC`
Group number for MMDC status codes.
- enumerator `kStatusGroup_PDM`
Group number for MIC status codes.
- enumerator `kStatusGroup_SDMA`
Group number for SDMA status codes.
- enumerator `kStatusGroup_ICS`
Group number for ICS status codes.
- enumerator `kStatusGroup_SPDIF`
Group number for SPDIF status codes.
- enumerator `kStatusGroup_LPC_MINISPI`
Group number for LPC_MINISPI status codes.
- enumerator `kStatusGroup_HASHCRYPT`
Group number for Hashcrypt status codes
- enumerator `kStatusGroup_LPC_SPI_SSP`
Group number for LPC_SPI_SSP status codes.
- enumerator `kStatusGroup_I3C`
Group number for I3C status codes
- enumerator `kStatusGroup_LPC_I2C_1`
Group number for LPC_I2C_1 status codes.
- enumerator `kStatusGroup_NOTIFIER`
Group number for NOTIFIER status codes.
- enumerator `kStatusGroup_DebugConsole`
Group number for debug console status codes.
- enumerator `kStatusGroup_SEMC`
Group number for SEMC status codes.
- enumerator `kStatusGroup_ApplicationRangeStart`
Starting number for application groups.
- enumerator `kStatusGroup_IAP`
Group number for IAP status codes
- enumerator `kStatusGroup_SFA`
Group number for SFA status codes
- enumerator `kStatusGroup_SPC`
Group number for SPC status codes.

- enumerator kStatusGroup_PUF
Group number for PUF status codes.
- enumerator kStatusGroup_TOUCH_PANEL
Group number for touch panel status codes
- enumerator kStatusGroup_VBAT
Group number for VBAT status codes
- enumerator kStatusGroup_XSPI
Group number for XSPI status codes
- enumerator kStatusGroup_PNGDEC
Group number for PNGDEC status codes
- enumerator kStatusGroup_JPEGDEC
Group number for JPEGDEC status codes
- enumerator kStatusGroup_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.

2.18 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 postiveInputChannel,
                                         uint32_t negativeInputChannel)
```

Set the ACOMP postive and negative input channel.

Parameters

- base – ACOMP peripheral base address.
- postiveInputChannel – The index of postive 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 diabled, 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.

2.19 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.

bool ADC_DoSelfCalibration(ADC_Type *base)

Do the hardware self-calibration.

Deprecated:

Do not use this function. It has been superceded by ADC_DoOffsetCalibration.

To calibrate the ADC, set the ADC clock to 500 kHz. In order to achieve the specified ADC accuracy, the A/D converter must be recalibrated, at a minimum, following every chip reset before initiating normal ADC operation.

Parameters

- base – ADC peripheral base address.

Return values

- true – Calibration succeed.
- false – Calibration failed.

bool ADC_DoOffsetCalibration(ADC_Type *base, uint32_t frequency)

Do the hardware offset-calibration.

To calibrate the ADC, set the ADC clock to no more then 30 MHz. In order to achieve the specified ADC accuracy, the A/D converter must be recalibrated, at a minimum, following every chip reset before initiating normal ADC operation.

Parameters

- base – ADC peripheral base address.
- frequency – The clock frequency that ADC operates at.

Return values

- true – Calibration succeed.
- false – Calibration failed.

```
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_inforeresultshift
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 interrpt/DMA trigger mode.

uint8_t seqSampleTimeNumber

Conversion sequence sampling time.

```
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.
```

2.20 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

2.21 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.

2.22 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

2.23 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, pint_pin_int_t 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, pin_t pintr)
```

Clear Selected pin interrupt rise flag.

This function clears the selected pin interrupt rise flag.

Parameters

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

Return values

None. –

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

Get selected pin interrupt rise flag.

This function returns the selected pin interrupt rise flag.

Parameters

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

Return values

flag – = 0 Rising edge has not been detected. = 1 Rising edge has been detected.

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

Clear all pin interrupt rise flags.

This function clears the rise flag for all pin interrupts.

Parameters

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

Return values

None. –

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

Get all pin interrupt rise flags.

This function returns the rise flag of all pin interrupts.

Parameters

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

Return values

flags – Each bit position indicates the rising edge detection of the corresponding pin interrupt. 0 Rising edge has not been detected. = 1 Rising edge has been detected.

```
void PINT_PatternMatchConfig(PINT_Type *base, pin_t bslice, pin_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, *pin_int_t* pinIdx)
enable callback by pin index.

This function enables callback by pin index instead of enabling all pins.

Parameters

- base – Base address of the peripheral.
- pinIdx – pin index.

Return values

None. –

void PINT_EnableInterruptByIndex(PINT_Type *base, *pin_int_t* pinIdx)
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, *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, *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>
```

2.24 Power

```
enum pd_bits
    power down configurations mask
    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_SYSOSC
    enumerator kPDRUNCFG_PD_LPO_OSC
    enumerator kPDRUNCFG_PD_SYSPLL
    enumerator kPDRUNCFG_PD_ACOMP
    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_SYSOSC`

enumerator `kPDAWAKECFG_Wakeup_LP_OSC`

enumerator `kPDAWAKECFG_Wakeup_ACMP`

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`
DPDCTRL bit 31-8

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_ResetLevelReserved

BOD Reset Level reserved.

enumerator kBod_ResetLevel1

BOD Reset Level1: 2.05V

enumerator kBod_ResetLevel2

BOD Reset Level2: 2.35V

enumerator kBod_ResetLevel3

BOD Reset Level3: 2.63V

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: 2.25V.

enumerator kBod_InterruptLevel2

BOD interrupt level2: 2.55V.

enumerator kBod_InterruptLevel3

BOD interrupt level3: 2.84V.

typedef enum *pd_bits* pd_bit_t

power down configurations mask

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

POWER_EnbaleLPOInDeepPowerDownMode

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_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_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_EnableLPO(bool enable)`

API to enable LPO.

Parameters

- enable – true to enable LPO, false to disable LPO.

`static inline void POWER_EnableLPOInDeepPowerDownMode(bool enable)`

API to enable LPO in deep power down mode.

Parameters

- enable – true to enable LPO, false to disable LPO.

`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. Note the kPMU_GenReg4 can restore 3 byte data only, so the general purpose register can store 19 bytes data.

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. Note the kPMU_GenReg4 can restore 3 byte data only, so the general purpose register can store 19 bytes data.

Parameters

- index – general purpose data register index.

Returns

data stored in the general purpose register.

```
static inline void POWER_EnableWktClkIn(bool enable, bool enHysteresis)
```

API to enable external clock input for self wake up timer.

Parameters

- enable – true is enable external clock input for self-wake-up timer, otherwise disable.
- enHysteresis – true is enable Hysteresis for the pin, otherwise disable.

```
static inline void POWER_EnableWakeupPinForDeepPowerDown(bool enable, bool enHysteresis)
```

API to enable wake up pin for deep power down mode.

Parameters

- enable – true is enable, otherwise disable.
- enHysteresis – true is enable Hysteresis for the pin, otherwise disable.

```
static inline void POWER_EnableResetPinForDeepPowerDown(bool enable, bool enHysteresis)
```

API to enable external clock input for self wake up timer.

Parameters

- enable – true is enable, otherwise disable.
- enHysteresis – true is enable Hysteresis for the pin, otherwise disable.

```
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.

2.25 Reset

```
enum _SYSCON_RSTn
```

Enumeration for peripheral reset control bits.

Defines the enumeration for peripheral reset control bits in PRESETCTRL/ASYNCPRESETCTRL registers

Values:

```
enumerator kFLASH_RST_N_SHIFT_RSTn
```

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 kSPI1_RST_N_SHIFT_RSTn
SPI1 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 kUART2_RST_N_SHIFT_RSTn
UART2 reset control

enumerator kIOCON_RST_N_SHIFT_RSTn
IOCON reset control

enumerator kACMP_RST_N_SHIFT_RSTn
Analog comparator reset control

enumerator kGPIO1_RST_N_SHIFT_RSTn
GPIO1 reset control

enumerator kFTM0_RST_N_SHIFT_RSTn
FTM0 reset control

enumerator kFTM1_RST_N_SHIFT_RSTn
FTM1 reset control

enumerator kI3C_RST_N_SHIFT_RSTn
I3C reset control

enumerator kADC_RST_N_SHIFT_RSTn
ADC reset control

enumerator kGPIOINT_RST_N_SHIFT_RSTn
GPIOINT reset control

enumerator kDMA_RST_N_SHIFT_RSTn
DMA reset control

enumerator kFRG0_RST_N_SHIFT_RSTn
Fractional baud rate generator 0 reset control

enumerator kFRG1_RST_N_SHIFT_RSTn
Fractional baud rate generator 1 reset control

```
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

FTM_RSTS_N

I3C_RSTS

ADC_RSTS_N

GPIPOINT_RSTS_N

DMA_RSTS_N

FRG_RSTS_N

2.26 SPI: Serial Peripheral Interface Driver

2.27 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 irqs)
```

Enables 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 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.

2.28 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_P0_18
port_pin number P0_18.

enumerator kSWM_PortPin_P0_19
port_pin number P0_19.

enumerator kSWM_PortPin_P0_20
port_pin number P0_20.

enumerator kSWM_PortPin_P0_21
port_pin number P0_21.

enumerator kSWM_PortPin_P0_22
port_pin number P0_22.

enumerator kSWM_PortPin_P0_23
port_pin number P0_23.

enumerator kSWM_PortPin_P0_24
port_pin number P0_24.

enumerator kSWM_PortPin_P0_25
port_pin number P0_25.

enumerator kSWM_PortPin_P0_26
port_pin number P0_26.

enumerator kSWM_PortPin_P0_27
port_pin number P0_27.

enumerator kSWM_PortPin_P0_28
port_pin number P0_28.

enumerator kSWM_PortPin_P0_29
port_pin number P0_29.

enumerator kSWM_PortPin_P0_30
port_pin number P0_30.

enumerator kSWM_PortPin_P0_31
port_pin number P0_31.

enumerator kSWM_PortPin_P1_0
port_pin number P1_0.

enumerator kSWM_PortPin_P1_1
port_pin number P1_1.

enumerator kSWM_PortPin_P1_2
port_pin number P1_2.

enumerator kSWM_PortPin_P1_3
port_pin number P1_3.

enumerator kSWM_PortPin_P1_4
port_pin number P1_4.

enumerator kSWM_PortPin_P1_5
port_pin number P1_5.

enumerator kSWM_PortPin_P1_6
port_pin number P1_6.

enumerator kSWM_PortPin_P1_7
port_pin number P1_7.

enumerator kSWM_PortPin_P1_8
port_pin number P1_8.

enumerator kSWM_PortPin_P1_9
port_pin number P1_9.

enumerator kSWM_PortPin_P1_10
port_pin number P1_10.

enumerator kSWM_PortPin_P1_11
port_pin number P1_11.

enumerator kSWM_PortPin_P1_12
port_pin number P1_12.

enumerator kSWM_PortPin_P1_13
port_pin number P1_13.

enumerator kSWM_PortPin_P1_14
port_pin number P1_14.

enumerator kSWM_PortPin_P1_15
port_pin number P1_15.

enumerator kSWM_PortPin_P1_16
port_pin number P1_16.

enumerator kSWM_PortPin_P1_17
port_pin number P1_17.

enumerator kSWM_PortPin_P1_18
port_pin number P1_18.

enumerator kSWM_PortPin_P1_19
port_pin number P1_19.

enumerator kSWM_PortPin_P1_20
port_pin number P1_20.

enumerator kSWM_PortPin_P1_21
port_pin number P1_21.

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_RTS
Movable function as USART1_RTS.

enumerator kSWM_USART1_CTS
Movable function as USART1_CTS.

enumerator kSWM_USART1_SCLK
Movable function as USART1_SCLK.

enumerator kSWM_USART2_TXD
Movable function as USART2_TXD.

enumerator kSWM_USART2_RXD
Movable function as USART2_RXD.

enumerator kSWM_USART2_RTS
Movable function as USART2_RTS.

enumerator kSWM_USART2_CTS
Movable function as USART2_CTS.

enumerator kSWM_USART2_SCLK
Movable function as USART2_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_SPI0_SSEL2
Movable function as SPI0_SSEL2.

enumerator kSWM_SPI0_SSEL3
Movable function as SPI0_SSEL3.

enumerator kSWM_SPI1_SCK
Movable function as SPI1_SCK.

enumerator kSWM_SPI1_MOSI
Movable function as SPI1_MOSI.

enumerator kSWM_SPI1_MISO
Movable function as SPI1_MISO.

enumerator kSWM_SPI1_SSEL0
Movable function as SPI1_SSEL0.

enumerator kSWM_SPI1_SSEL1
Movable function as SPI1_SSEL1.

enumerator kSWM_I2C0_SDA
Movable function as I2C0_SDA.

enumerator kSWM_I2C0_SCL
Movable function as I2C0_SCL.

enumerator kSWM_I3C0_SDA
Movable function as I3C0_SDA.

enumerator kSWM_I3C0_SCL
Movable function as I3C0_SCL.

enumerator kSWM_I3C0_PUR
Movable function as I3C0_PUR.

enumerator kSWM_ACMP_O
Movable function as COMP0_OUT.

enumerator kSWM_CLKOUT
Movable function as CLKOUT.

enumerator kSWM_GPIO_INT_BMAT
Movable function as GPIO_INT_BMAT.

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_ACMP_INPUT5
Fixed-pin function as ACMP_INPUT5.

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_CMPVREF
Fixed-pin function as CMPVREF.

enumerator kSWM_XTALIN
Fixed-pin function as XTALIN.

enumerator kSWM_XTALOUT
Fixed-pin function as XTALOUT.

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.

enum _swm_flextimer_pin_func_t

SWM flextimer pin function.

Values:

enumerator kSWM_FTM0_EXTCLK

Flextimer function as FTM0_EXTCLK.

enumerator kSWM_FTM0_CH0

Flextimer function as FTM0_CH0.

enumerator kSWM_FTM0_CH1

Flextimer function as FTM0_CH1.

enumerator kSWM_FTM0_CH2

Flextimer function as FTM0_CH2.

enumerator kSWM_FTM0_CH3

Flextimer function as FTM0_CH3.

enumerator kSWM_FTM0_CH4

Flextimer function as FTM0_CH4.

enumerator kSWM_FTM0_CH5

Flextimer function as FTM0_CH5.

enumerator kSWM_FTM0_FAULT0

Flextimer function as FTM0_FAULT0.

enumerator kSWM_FTM0_FAULT1

Flextimer function as FTM0_FAULT1.

enumerator kSWM_FTM0_FAULT2

Flextimer function as FTM0_FAULT2.

enumerator kSWM_FTM0_FAULT3

Flextimer function as FTM0_FAULT3.

enumerator kSWM_FTM1_EXTCLK
Flextimer function as FTM1_EXTCLK.

enumerator kSWM_FTM1_CH0
Flextimer function as FTM1_CH0.

enumerator kSWM_FTM1_CH1
Flextimer function as FTM1_CH1.

enumerator kSWM_FTM1_CH2
Flextimer function as FTM1_CH2.

enumerator kSWM_FTM1_CH3
Flextimer function as FTM1_CH3.

enumerator kSWM_FTM1_QD_PHA
Flextimer function as FTM1_QD_PHA.

enumerator kSWM_FTM1_QD_PHB
Flextimer function as FTM1_QD_PHB.

enumerator kSWM_FTM_NUM_FUNCS
Flextimer function function number.

enum `_swm_select_flextimer_pin_sel_t`
SWM flextimer pin selection.

Values:

enumerator kSWM_FTM_Selection0
Flextimer pin selection0.

enumerator kSWM_FTM_Selection1
Flextimer pin selection1.

enumerator kSWM_FTM_Selection2
Flextimer pin selection2.

enumerator kSWM_FTM_Selection3
Flextimer pin selection3, which is not connected by default on LPC86x.

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.

typedef enum `_swm_flextimer_pin_func_t` `swm_flextimer_pin_func_t`
SWM flextimer pin function.

typedef enum `_swm_select_flextimer_pin_sel_t` `swm_flextimer_pin_sel_t`
SWM flextimer 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.

```
void SWM_SetFlextimerPinSelect(SWM_Type *base, swm_flextimer_pin_func_t func,  
                               swm_flextimer_pin_sel_t selection)
```

Enable the flextimer function.

This function will enables a flextimer function in FTM_PINASSIGN0 or FTM_PINASSIGN1.

Parameters

- base – SWM peripheral base address.
- func – any function name that is flextimer.
- selection – flextimer pin selection.

2.29 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_GpioPort0Pin6ToPintsel
```

```
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
enumerator kSYSCON_GpioPort0Pin18ToPintsel
enumerator kSYSCON_GpioPort0Pin19ToPintsel
enumerator kSYSCON_GpioPort0Pin20ToPintsel
enumerator kSYSCON_GpioPort0Pin21ToPintsel
enumerator kSYSCON_GpioPort0Pin22ToPintsel
enumerator kSYSCON_GpioPort0Pin23ToPintsel
enumerator kSYSCON_GpioPort0Pin24ToPintsel
enumerator kSYSCON_GpioPort0Pin25ToPintsel
enumerator kSYSCON_GpioPort0Pin26ToPintsel
enumerator kSYSCON_GpioPort0Pin27ToPintsel
enumerator kSYSCON_GpioPort0Pin28ToPintsel
enumerator kSYSCON_GpioPort0Pin29ToPintsel
enumerator kSYSCON_GpioPort0Pin30ToPintsel
enumerator kSYSCON_GpioPort0Pin31ToPintsel
enumerator kSYSCON_GpioPort1Pin0ToPintsel
enumerator kSYSCON_GpioPort1Pin1ToPintsel
enumerator kSYSCON_GpioPort1Pin2ToPintsel
enumerator kSYSCON_GpioPort1Pin3ToPintsel
enumerator kSYSCON_GpioPort1Pin4ToPintsel
enumerator kSYSCON_GpioPort1Pin5ToPintsel
enumerator kSYSCON_GpioPort1Pin6ToPintsel
enumerator kSYSCON_GpioPort1Pin7ToPintsel
enumerator kSYSCON_GpioPort1Pin8ToPintsel
enumerator kSYSCON_GpioPort1Pin9ToPintsel
enumerator kSYSCON_GpioPort1Pin10ToPintsel
enumerator kSYSCON_GpioPort1Pin11ToPintsel
enumerator kSYSCON_GpioPort1Pin12ToPintsel

enumerator kSYSCON_GpioPort1Pin13ToPintsel
 enumerator kSYSCON_GpioPort1Pin14ToPintsel
 enumerator kSYSCON_GpioPort1Pin15ToPintsel
 enumerator kSYSCON_GpioPort1Pin16ToPintsel
 enumerator kSYSCON_GpioPort1Pin17ToPintsel
 enumerator kSYSCON_GpioPort1Pin18ToPintsel
 enumerator kSYSCON_GpioPort1Pin19ToPintsel
 enumerator kSYSCON_GpioPort1Pin20ToPintsel
 enumerator kSYSCON_GpioPort1Pin21ToPintsel
 enumerator kSYSCON_GpioPort1Pin22ToPintsel
 enumerator kSYSCON_GpioPort1Pin23ToPintsel
 enumerator kSYSCON_GpioPort1Pin24ToPintsel
 enumerator kSYSCON_GpioPort1Pin25ToPintsel
 enumerator kSYSCON_GpioPort1Pin26ToPintsel
 enumerator kSYSCON_GpioPort1Pin27ToPintsel
 enumerator kSYSCON_GpioPort1Pin28ToPintsel
 enumerator kSYSCON_GpioPort1Pin29ToPintsel
 enumerator kSYSCON_GpioPort1Pin30ToPintsel
 enumerator kSYSCON_GpioPort1Pin31ToPintsel

typedef enum *syscon_connection_t* syscon_connection_t
 SYSCON connections type.

PINTSEL_ID

Periphinmux IDs.

SYSCON_SHIFT

FSL_SYSON_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. –

2.30 USART: Universal Asynchronous Receiver/Transmitter Driver

2.31 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, 20000000U);
```

Parameters

- `base` – USART peripheral base address.
- `baudrate_Bps` – USART baudrate to be set.
- `srcClock_Hz` – USART clock source frequency in HZ.

Return values

- `kStatus_USART_BaudrateNotSupport` – Baudrate is not support in current clock source.
- `kStatus_Success` – Set baudrate succeed.
- `kStatus_InvalidArgument` – One or more arguments are invalid.

`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 void USART__SetRxIdleTimeout(USART_Type *base, uint8_t rxIdleTimeout)
```

Config the USART instance rx idle timeout.

This function configures the number idle character of USART rx idle. For 115200,8n1, 1 character timing is $86.81\mu\text{s} = 1 / (115200 / (1\text{start} + 8\text{data} + 0\text{parity} + 1\text{stop}))$

```
USART__SetRxIdleTimeout(USART1, 1);
```

Parameters

- base – USART peripheral base address.
- rxIdleTimeout – The configuration of UART rx idle .

```
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 function, the Continuous Clock bit is automatically cleared when a complete character has been received. This bit is cleared at the same time.

Parameters

- base – USART peripheral base address.
- enable – Enable auto clear or not, true for enable and false for disable.

static inline void USART_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 `remainBtyes` to find out how many bytes are still not sent out.

Parameters

- `base` – USART peripheral base address.
- `handle` – USART handle pointer.

```
status_t USART_TransferGetSendCount(USART_Type *base, usart_handle_t *handle, uint32_t *count)
```

Get the number of bytes that have been 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.

enumerator kStatus_USART_RxIdleTimeout

USART receive idle 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_RxIdleTimeoutInterruptEnable`

Receive idle timeout 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.

enumerator `kUSART_RxIdleTimeoutFlag`
Receive idle timeout 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.

`uint8_t rxIdleTimeout`

Receive idle bytes. Value [0,7]. Set the value to n then the idle byte count will be the (n)th power of 2.

`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 __unnamed12__

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.

2.32 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

2.33 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 = 0xFFFFFU;
config->timeoutValue = 0xFFFFFU;
config->warningValue = 0;
```

See also:

wwdt_config_t

Parameters

- *config* – Pointer to WWDT config structure.

```
void WWDT_Init(WWDT_Type *base, const wwdt_config_t *config)
```

Initializes the WWDT.

This function initializes the WWDT. When called, the WWDT runs according to the configuration.

Example:

```
wwdt_config_t config;
WWDT_GetDefaultConfig(&config);
config.timeoutValue = 0x7ffU;
WWDT_Init(wwdt_base,&config);
```

Parameters

- *base* – WWDT peripheral base address
- *config* – The configuration of WWDT

```
void WWDT_Deinit(WWDT_Type *base)
```

Shuts down the WWDT.

This function shuts down the WWDT.

Parameters

- base – WWDT peripheral base address

```
static inline void WWDT_Enable(WWDT_Type *base)
```

Enables the WWDT module.

This function write value into WWDT_MOD register to enable the WWDT, it is a write-once bit; once this bit is set to one and a watchdog feed is performed, the watchdog timer will run permanently.

Parameters

- base – WWDT peripheral base address

```
static inline void WWDT_Disable(WWDT_Type *base)
```

Disables the WWDT module.

Deprecated:

Do not use this function. It will be deleted in next release version, for once the bit field of WDEN written with a 1, it can not be re-written with a 0.

This function write value into WWDT_MOD register to disable the WWDT.

Parameters

- base – WWDT peripheral base address

```
static inline uint32_t WWDT_GetStatusFlags(WWDT_Type *base)
```

Gets all WWDT status flags.

This function gets all status flags.

Example for getting Timeout Flag:

```
uint32_t status;
status = WWDT_GetStatusFlags(wwdt_base) & kWWDT_TimeoutFlag;
```

Parameters

- base – WWDT peripheral base address

Returns

The status flags. This is the logical OR of members of the enumeration `_wwdt_status_flags_t`

```
void WWDT_ClearStatusFlags(WWDT_Type *base, uint32_t mask)
```

Clear WWDT flag.

This function clears WWDT status flag.

Example for clearing warning flag:

```
WWDT_ClearStatusFlags(wwdt_base, kWWDT_WarningFlag);
```

Parameters

- base – WWDT peripheral base address
- mask – The status flags to clear. This is a logical OR of members of the enumeration `_wwdt_status_flags_t`

```
static inline void WWDT_SetWarningValue(WWDT_Type *base, uint32_t warningValue)
```

Set the WWDT warning value.

The WDWARNINT register determines the watchdog timer counter value that will generate a watchdog interrupt. When the watchdog timer counter is no longer greater than the value defined by WARNINT, an interrupt will be generated after the subsequent WDCLK.

Parameters

- base – WWDT peripheral base address
- warningValue – WWDT warning value.

```
static inline void WWDT_SetTimeoutValue(WWDT_Type *base, uint32_t timeoutCount)
```

Set the WWDT timeout value.

This function sets the timeout value. Every time a feed sequence occurs the value in the TC register is loaded into the Watchdog timer. Writing a value below 0xFF will cause 0xFF to be loaded into the TC register. Thus the minimum time-out interval is $TWDCLK * 256 * 4$. If enableWatchdogProtect flag is true in `wwdt_config_t` config structure, any attempt to change the timeout value before the watchdog counter is below the warning and window values will cause a watchdog reset and set the WDTOF flag.

Parameters

- base – WWDT peripheral base address
- timeoutCount – WWDT timeout value, count of WWDT clock tick.

```
static inline void WWDT_SetWindowValue(WWDT_Type *base, uint32_t windowValue)
```

Sets the WWDT window value.

The WINDOW register determines the highest TV value allowed when a watchdog feed is performed. If a feed sequence occurs when timer value is greater than the value in WINDOW, a watchdog event will occur. To disable windowing, set windowValue to 0xFFFFFFFF (maximum possible timer value) so windowing is not in effect.

Parameters

- base – WWDT peripheral base address
- windowValue – WWDT window value.

```
void WWDT_Refresh(WWDT_Type *base)
```

Refreshes the WWDT timer.

This function feeds the WWDT. This function should be called before WWDT timer is in timeout. Otherwise, a reset is asserted.

Parameters

- base – WWDT peripheral base address

```
FSL_WWDT_DRIVER_VERSION
```

Defines WWDT driver version.

```
WWDT_FIRST_WORD_OF_REFRESH
```

First word of refresh sequence

```
WWDT_SECOND_WORD_OF_REFRESH
```

Second word of refresh sequence

```
enum _wwdt_status_flags_t
```

WWDT status flags.

This structure contains the WWDT status flags for use in the WWDT functions.

Values:

enumerator kWWDT_TimeoutFlag

Time-out flag, set when the timer times out

enumerator kWWDT_WarningFlag

Warning interrupt flag, set when timer is below the value WDWARNINT

typedef struct *_wwdt_config* wwdt_config_t

Describes WWDT configuration structure.

struct *_wwdt_config*

#include <fsl_wwdt.h> Describes WWDT configuration structure.

Public Members

bool enableWwdt

Enables or disables WWDT

bool enableWatchdogReset

true: Watchdog timeout will cause a chip reset false: Watchdog timeout will not cause a chip reset

bool enableWatchdogProtect

true: Enable watchdog protect i.e timeout value can only be changed after counter is below warning & window values false: Disable watchdog protect; timeout value can be changed at any time

uint32_t windowValue

Window value, set this to 0xFFFFFFFF if windowing is not in effect

uint32_t timeoutValue

Timeout value

uint32_t warningValue

Watchdog time counter value that will generate a warning interrupt. Set this to 0 for no warning

uint32_t clockFreq_Hz

Watchdog clock source frequency.

Chapter 3

Middleware

3.1 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_pdbdm** uses JTAG or BDM debugging interface to access the target RAM directly while the CPU is running. Note that such approach can be used with any MCU application, even without any special driver code. The computer reads from and writes into the RAM directly without CPU intervention. The Packet-Driven BDM (PD-BDM) communication uses the same memory access to exchange command and response frames. With PD-BDM, the FreeMASTER tool is able to go beyond basic memory read/write operations and accesses also advanced features like Recorder, TSA, or Pipes. The typical transmission speed is around 10 kbps. A PD-BDM communication plug-in must be used in FreeMASTER and configured properly for the selected debugging interface. Note that this communication cannot be used while a debugging interface is used by a debugger session.
- **fmstr_any** is a special example application which demonstrates how the NXP MCUXpresso Config Tools can be used to configure pins, clocks, peripherals, interrupts, and even the FreeMASTER "middleware" driver features in a graphical and user friendly way. The user can switch between the Serial, CAN, and other ways of communication and generate the required initialization code automatically.

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

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

Description

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

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

- Read/write access to any memory location on the target.
- Optional password protection of the read, read/write, and read/write/flash access levels.

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

The FreeMASTER driver features:

- Full FreeMASTER protocol V4 implementation with a new V4 style of CRC used.
- Layered approach supporting Serial, CAN, Network, PD-BDM, and other transports.
- Layered low-level Serial transport driver architecture enabling to select UART, LPUART, USART, and other physical implementations of serial interfaces, including USB-CDC.
- Layered low-level CAN transport driver architecture enabling to select FlexCAN, msCAN, MCAN, and other physical implementations of the CAN interface.
- Layered low-level Networking transport enabling to select TCP, UDP or J-Link RTT communication.
- TSA support to write-protect memory regions or individual variables and to deny the access to the unsafe memory.
- The pipe callback handlers are invoked whenever new data is available for reading from the pipe.
- Two Serial Single-Wire modes of operation are enabled. The “external” mode has the RX and TX shorted on-board. The “true” single-wire mode interconnects internally when the MCU or UART modules support it.

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

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

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

- Number of the Recorder and Oscilloscope instances.
- RAM Base Address for optimized memory access commands.

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

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

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

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

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

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

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

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

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

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

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

The information is encoded as so-called TSA tables which become direct part of the application code. The TSA tables contain descriptors of variables that shall be visible to the host tool. The descriptors can describe the memory areas by specifying the address and size of the memory

block or more conveniently using the C variable names directly. Different set of TSA descriptors can be used to encode information about the structure types, unions, enumerations, or arrays.

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

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

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

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

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

- “External” single-wire operation where the Receiver and Transmitter pins are shorted on the board. This mode is supported by default in the MCU driver because the Receiver and Transmitter units are enabled or disabled whenever needed. It is also easy to extend this operation for the RS485 communication.
- “True” single-wire mode which uses only a single pin and the direction switching is made by the UART module. This mode of operation must be enabled by defining the FMSTR_SERIAL_SINGLEWIRE configuration option.

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

Zephyr-specific

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

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

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

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

- **src/platforms** platform-specific folder—one folder exists for each supported processor platform (for example; 32-bit Little Endian platform). Each such folder contains a platform header file with data types and a code which implements the potentially platform-specific operations, such as aligned memory access.
- **src/common** folder—contains the common driver source files shared by the driver for all supported platforms. All the .c files must be added to the project, compiled, and linked together with the application.
 - *freemaster.h* - master driver header file, which declares the common data types, macros, and prototypes of the FreeMASTER driver API functions.
 - *freemaster_cfg.h.example* - this file can serve as an example of the FreeMASTER driver configuration file. Save this file into a project source code folder and rename it to *freemaster_cfg.h*. The FreeMASTER driver code includes this file to get the project-specific configuration options and to optimize the compilation of the driver.
 - *freemaster_defcfg.h* - defines the default values for each FreeMASTER configuration option if the option is not set in the *freemaster_cfg.h* file.
 - *freemaster_protocol.h* - defines the FreeMASTER protocol constants used internally by the driver.
 - *freemaster_protocol.c* - implements the FreeMASTER protocol decoder and handles the basic Get Configuration Value, Memory Read, and Memory Write commands.
 - *freemaster_rec.c* - handles the Recorder-specific commands and implements the Recorder sampling and triggering routines. When the Recorder is disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.
 - *freemaster_scope.c* - handles the Oscilloscope-specific commands. If the Oscilloscope is disabled by the FreeMASTER driver configuration file, this file compiles as void.
 - *freemaster_pipes.c* - implements the Pipes functionality when the Pipes feature is enabled.
 - *freemaster_appcmd.c* - handles the communication commands used to deliver and execute the Application Commands within the context of the embedded application. When the Application Commands are disabled by the FreeMASTER driver configuration file, this file only compiles to empty API functions.

- *freemaster_tsa.c* - handles the commands specific to the TSA feature. This feature enables the FreeMASTER host tool to obtain the TSA memory descriptors declared in the embedded application. If the TSA is disabled by the FreeMASTER driver configuration file, this file compiles as void.
- *freemaster_tsa.h* - contains the declaration of the macros used to define the TSA memory descriptors. This file is indirectly included into the user application code (via *freemaster.h*).
- *freemaster_sha.c* - implements the SHA-1 hash code used in the password authentication algorithm.
- *freemaster_private.h* - contains the declarations of functions and data types used internally in the driver. It also contains the C pre-processor statements to perform the compile-time verification of the user configuration provided in the *freemaster_cfg.h* file.
- *freemaster_serial.c* - implements the serial protocol logic including the CRC, FIFO queuing, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a character-oriented API exported by the specific low-level driver.
- *freemaster_serial.h* - defines the low-level character-oriented Serial API.
- *freemaster_can.c* - implements the CAN protocol logic including the CAN message preparation, signalling using the first data byte in the CAN frame, and other communication-related operations. This code calls the functions of the low-level communication driver indirectly via a message-oriented API exported by the specific low-level driver.
- *freemaster_can.h* - defines the low-level message-oriented CAN API.
- *freemaster_net.c* - implements the Network protocol transport logic including multiple session management code.
- *freemaster_net.h* - definitions related to the Network transport.
- *freemaster_pdbdm.c* - implements the packet-driven BDM communication buffer and other communication-related operations.
- *freemaster_utils.c* - aligned memory copy routines, circular buffer management and other utility functions
- *freemaster_utils.h* - definitions related to utility code.
- **src/drivers/[sdk]/serial** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_serial_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the UART, LPUART, USART, and other kinds of Serial communication modules.
- **src/drivers/[sdk]/can** - contains the code related to the serial communication implemented using one of the supported SDK frameworks.
 - *freemaster_XXX.c* and *.h* - implement low-level access to the communication peripheral registers. Different files exist for the FlexCAN, msCAN, MCAN, and other kinds of CAN communication modules.
- **src/drivers/[sdk]/network** - contains low-level code adapting the FreeMASTER Network transport to an underlying TCP/IP or RTT stack.
 - *freemaster_net_lwip_tcp.c* and *_udp.c* - default networking implementation of TCP and UDP transports using lwIP stack.
 - *freemaster_net_segger_rtt.c* - implementation of network transport using Segger J-Link RTT interface

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

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

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

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

Interrupt modes

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

Value Type boolean (0 or 1)

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

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

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

Protocol transport

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

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

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

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

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

```
#define FMSTR_TRANSPORT FMSTR_SERIAL
```

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

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

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

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

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

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

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

FMSTR_SERIAL_BASE

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

Value Type Optional address value (numeric or symbolic)

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

FMSTR_COMM_BUFFER_SIZE

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

Value Type 0 or a value in range 32...255

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

FMSTR_COMM_QUEUE_SIZE

```
#define FMSTR_COMM_QUEUE_SIZE [number]
```

Value Type Value in range 0...255

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

FMSTR_SERIAL_SINGLEWIRE

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

Value Type Boolean 0 or 1.

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

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

```
#define FMSTR_TRANSPORT FMSTR_CAN
```

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

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

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

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

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

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

FMSTR_CAN_BASE

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

Value Type Optional address value (numeric or symbolic)

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

FMSTR_CAN_CMDID

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

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

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

FMSTR_CAN_RSPID

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

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

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

FMSTR_FLEXCAN_TXMB

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

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

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

FMSTR_FLEXCAN_RXMB

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

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

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

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

```
#define FMSTR_TRANSPORT FMSTR_NET
```

FMSTR_NET_DRV Select network interface implementation.

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

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

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

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

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

Add another row below:

FMSTR_NET_PORT

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

Value Type TCP or UDP port number (short integer)

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

FMSTR_NET_BLOCKING_TIMEOUT

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

Value Type Timeout as number of milliseconds

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

FMSTR_NET_AUTODISCOVERY

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

Value Type Boolean 0 or 1.

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

Debugging options

FMSTR_DISABLE

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

Value Type boolean (0 or 1)

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

FMSTR_DEBUG_TX

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

Value Type Boolean 0 or 1.

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

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

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

FMSTR_APPLICATION_STR

```
#define FMSTR_APPLICATION_STR
```

Value Type String.

Description Name of the application visible in FreeMASTER host application.

Memory access

FMSTR_USE_READMEM

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_WRITEMEM

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

Value Type Boolean 0 or 1.

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

Oscilloscope options

FMSTR_USE_SCOPE

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

Value Type Integer number.

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

FMSTR_MAX_SCOPE_VARS

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

Value Type Integer number larger than 2.

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

Recorder options

FMSTR_USE_RECORDER

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

Value Type Integer number.

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

Default value is 0.

FMSTR_REC_BUFF_SIZE

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

Value Type Integer number larger than 2.

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

FMSTR_REC_TIMEBASE

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

Value Type Number (nanoseconds time).

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

Use one of the following macros:

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

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

FMSTR_REC_FLOAT_TRIG

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

Value Type Boolean 0 or 1.

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

Default value is 0 (false).

Application Commands options

FMSTR_USE_APPCMD

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

Value Type Boolean 0 or 1.

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

FMSTR_APPCMD_BUFF_SIZE

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

Value Type Numeric buffer size in range 1..255

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

FMSTR_MAX_APPCMD_CALLS

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

Value Type Number in range 0..255

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

TSA options

FMSTR_USE_TSA

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_TSA_SAFETY

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_TSA_INROM

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

Value Type Boolean 0 or 1.

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

FMSTR_USE_TSA_DYNAMIC

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

Value Type Boolean 0 or 1.

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

Pipes options

FMSTR_USE_PIPES

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

Value Type Boolean 0 or 1.

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

FMSTR_MAX_PIPES_COUNT

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

Value Type Number in range 1..63.

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

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

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

Completely Interrupt-Driven operation Activated using:

```
#define FMSTR_LONG_INTR 1
```

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

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

Mixed Interrupt and Polling Modes Activated using:

```
#define FMSTR_SHORT_INTR 1
```

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

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

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

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

Completely Poll-driven

```
#define FMSTR_POLL_DRIVEN 1
```

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

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

In the latter two modes (*FMSTR_SHORT_INTR* and *FMSTR_POLL_DRIVEN*), the protocol handling takes place in the *FMSTR_Poll* routine. An application interrupt can occur in the middle of the

Read Memory or Write Memory commands' execution and corrupt the variable being accessed by the FreeMASTER driver. In these two modes, some issues or glitches may occur when using FreeMASTER to visualize or monitor volatile variables modified in interrupt servicing code.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Driver API

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

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

FMSTR_Init

Prototype

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

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

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

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

FMSTR_Poll

Prototype

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

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

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

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

$$N * Tchar$$

where:

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

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

FMSTR_SerialIsr / FMSTR_CanIsr

Prototype

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

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

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

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

Recorder API

FMSTR_RecorderCreate

Prototype

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

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

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

For more information, see [Configurable items](#).

FMSTR_Recorder

Prototype

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

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

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

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

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

FMSTR_RecorderTrigger

Prototype

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

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

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

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

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

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

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

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

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

```
FMSTR_TSA_TABLE_BEGIN(table_id)
```

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

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

/* Description of complex data types */
FMSTR_TSA_STRUCT(struct_name) /* structure or union type entry */
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 *FM-STR_SIZE*.

<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 *FM-STR_SIZE*.

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

<i>FM-STR_HPIPE</i>	Pipe handle that identifies the open-pipe object.
---------------------	---

Generally, this is a pointer to a void type.

<i>FM-STR_PIPE_PC</i>	Integer type required to hold at least 7 bits of data.
-----------------------	--

Generally, this is an unsigned 8-bit or 16-bit type.

<i>FM-STR_PIPE_SI</i>	Integer type required to hold at least 16 bits of data.
-----------------------	---

This is used to store the data buffer sizes.

<i>FM-STR_PPIPEF</i>	Pointer to the pipe handler function.
----------------------	---------------------------------------

See [FM-STR_PipeOpen](#) for more details.

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

<i>FMSTR_U8</i>	The smallest memory entity.
On the vast majority of platforms, this is an unsigned 8-bit integer.	
On the 56F8xx DSP platform, this is defined as an unsigned 16-bit integer.	
<i>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
- FreeMASTER community area: community.nxp.com/community/freemaster
- FreeMASTER GitHub code repo: <https://github.com/nxp-mcuxpresso/mcux-freemaster>
- MCUXpresso SDK home: www.nxp.com/mcuxpresso
- MCUXpresso SDK builder: mcuxpresso.nxp.com/en

Documents

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

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

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

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